## Politecnico di Torino

Computer Engineering

A.a. 2025/2026

Graduation Session March 2026

# Orchestrating Network Security Borders in the Computing Continuum with Liqo

Supervisors:

Fulvio Risso
Stefano Galantino
Attilio Oliva

Candidate:

Riccardo Tornesello

## Abstract

The transition from centralized cloud computing models to the computing continuum paradigm has redefined IT infrastructure management requirements, introducing unprecedented complexity in the areas of orchestration and network security. Typical continuum scenarios are inherently characterized by a plurality of heterogeneous providers operating within a framework of mutual distrust. In this context, traditional security models are obsolete: executing workloads in shared environments requires the definition of granular isolation perimeters, not based on nodes as was previously the case. Moving beyond the now outdated logic based on physical nodes, this approach aims to mitigate the risk of compromise and inhibit unauthorized access.

Although Kubernetes has established itself as the de facto standard for resource abstraction, it lacks the concept of federation natively, making it necessary to use external tools to enable peering between separate clusters. This is where Liqo comes in, a project aimed at extending the Kubernetes control plane beyond administrative boundaries through dynamic cluster interconnection. However, Liqo's current network model presents critical issues in multi-tenant scenarios, as its basic configuration allows indiscriminate connectivity between clusters, without adequate isolation mechanisms.

This thesis aims to design and implement an advanced Network Manager, integrated within the Liqo multi-cluster orchestration platform. The technological core of the solution lies in a dynamic controller capable of interpreting high-level intents—such as the definition of security boundaries between federated resources—and translating them into low-level network primitives. The fundamental objective is to ensure compatibility with different Container Network Interfaces (CNI) plugins, with a specific focus on the coexistence between the Netfilter framework and emerging eBPF-based technologies.

# Table of Contents


**1 Introduction** 1

**2 Background** 3

2.1 Programmable Networking in Linux Environments . . . . . . . . . . 3
2.1.1 From Iptables to Nftables . . . . . . . . . . . . . . . . . . . . . 4
2.1.2 Scalability through Sets and Maps . . . . . . . . . . . . . . . . 5
2.1.3 eBPF . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.1.4 Coexistence between the two models . . . . . . . . . . . . . . . 6
2.1.5 Technology summary . . . . . . . . . . . . . . . . . . . . . . . . 6
2.2 Introduction to Kubernetes . . . . . . . . . . . . . . . . . . . . . . . 6
2.2.1 Cluster architecture . . . . . . . . . . . . . . . . . . . . . . . . 7
2.2.2 Cloud-Native Networking: the CNI . . . . . . . . . . . . . . . . 9
2.2.3 Network Policies . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.2.4 Evolution of Data Planes: From iptables to eBPF . . . . . . 11
2.2.5 CNIs comparison . . . . . . . . . . . . . . . . . . . . . . . . . . 12
2.3 Computing Continuum . . . . . . . . . . . . . . . . . . . . . . . . . . 14
2.3.1 Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
2.3.2 Kubernetes in the Computing Continuum . . . . . . . . . . . 15
2.3.3 Orchestration . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
2.3.4 Applications . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
2.4 Liqo . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
2.4.1 The network fabric . . . . . . . . . . . . . . . . . . . . . . . . . 18
2.4.2 The external network . . . . . . . . . . . . . . . . . . . . . . . . 19
2.4.3 FirewallConfiguration . . . . . . . . . . . . . . . . . . . . . . . . 20

**3 Problem analysis** 23

3.1 Use cases . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
3.1.1 Provider Protection . . . . . . . . . . . . . . . . . . . . . . . . . 24
3.1.2 Consumer protection . . . . . . . . . . . . . . . . . . . . . . . . 25
3.1.3 Leaf-to-Leaf Protection . . . . . . . . . . . . . . . . . . . . . . . 28
3.1.4 Multiconsumer Protection . . . . . . . . . . . . . . . . . . . . . 28

3.2 Firewalling solutions . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
3.3 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

**4 Implementation** 35
4.1 High level design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
4.2 CRD . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
4.2.1 Groups . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
4.2.2 Rules definition . . . . . . . . . . . . . . . . . . . . . . . . . . 41
4.3 Extending FirewallConfiguration . . . . . . . . . . . . . . . . . . . . . 44
4.3.1 Support for Connection Tracking State . . . . . . . . . . . . . 44
4.3.2 Named Sets management . . . . . . . . . . . . . . . . . . . . . 46
4.4 Rules translation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
4.4.1 Generation Logic . . . . . . . . . . . . . . . . . . . . . . . . . 49
4.4.2 Groups conversion . . . . . . . . . . . . . . . . . . . . . . . . 51
4.4.3 Conversion Flow for FirewallConfiguration . . . . . . . . . . . 51
4.4.4 Conversion Flow for NetworkPolicy . . . . . . . . . . . . . . . 52
4.5 Controller . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
4.5.1 The reconciliation loop . . . . . . . . . . . . . . . . . . . . . . 54
4.5.2 Triggers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
4.5.3 Systemic Resilience . . . . . . . . . . . . . . . . . . . . . . . . 56

**5 Tests** 58
5.1 Experimental Methodology and Infrastructure Automation . . . . . 58
5.2 Validation Methodology: The Connectivity Verification Framework 59
5.3 Test Suite . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
5.3.1 Policy resources . . . . . . . . . . . . . . . . . . . . . . . . . . 61
5.4 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
5.4.1 Single peering . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
5.4.2 Multiconsumer . . . . . . . . . . . . . . . . . . . . . . . . . . 65
5.4.3 Multiprovider . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
5.4.4 Differences with Cilium . . . . . . . . . . . . . . . . . . . . . . 70
5.4.5 Comparison with the FirewallConfiguration-only model . . . 71
5.4.6 Performance impact . . . . . . . . . . . . . . . . . . . . . . . 72

**6 Conclusion and future work** 75

**Bibliography** 77

# Chapter 1

# Introduction

The widespread adoption of microservices-based architectures and the transition to cloud-native paradigms have radically redefined IT infrastructure management. While centralized cloud computing has historically offered elastic scalability and operational efficiency, the explosion of Internet of Things (IoT) devices and massive data generation at the network edge have exposed its physical and logical limitations. Strict latency constraints, bandwidth bottlenecks, and stringent data sovereignty requirements are currently catalyzing a paradigm shift away from centralized models and toward the Computing Continuum.

Within this highly distributed landscape, Kubernetes has emerged as the de facto operating system, abstracting underlying heterogeneous resources into a unified control plane. However, extending Kubernetes beyond single geographic and administrative boundaries introduces profound orchestration and network security challenges. Typical continuum scenarios are inherently characterized by a plurality of heterogeneous providers operating within a framework of mutual distrust.

In this context, traditional security models are obsolete. The network is no longer just a means of transport, but a critical component responsible for isolation, encryption, and observability. Executing workloads in shared, distributed environments requires the definition of granular isolation perimeters. Moving beyond outdated physical node-based logic, this approach is essential to mitigate the risk of compromise and inhibit unauthorized access.

Although Kubernetes excels at resource abstraction, it lacks native federation concepts, necessitating external tools to enable peering between separate clusters. Liqo addresses this gap by extending the Kubernetes control plane beyond administrative boundaries through dynamic cluster interconnection, allowing for seamless computation offloading. However, Liqo's current network model presents critical vulnerabilities in multi-tenant scenarios. Its baseline configuration allows indiscriminate connectivity between clusters without adequate isolation mechanisms,

currently requiring low-level manual intervention to secure.

Initial steps to address these vulnerabilities were undertaken in the previous thesis work by Giulio Brazzo[1]. His research established a foundational security layer, developing the basic network functionalities necessary to protect a provider cluster from unauthorized access during peering. While his solution effectively mitigated primary exposure risks, it served as a generalized baseline defense that lacked the fine-grained control required for complex, multi-tenant deployments.

Building directly upon Brazzo's foundational architecture, the primary objective of this thesis is to design and implement a more complete and granular Network Manager integrated within the Liqo multi-cluster orchestration platform. This work aims to engineer a sophisticated traffic control and security mechanism by creating a dynamic Kubernetes controller that natively interfaces with Liqo.

The technological core of this solution lies in bridging the gap between high-level administrative intents (e.g., defining security boundaries to "isolate peering A") and low-level network enforcement (e.g., automatically generating nftables rules). A fundamental requirement of this implementation is ensuring compatibility with various Container Network Interface (CNI) plugins, with a specific focus on managing the coexistence and resolving potential conflicts between the traditional Netfilter framework and emerging eBPF-based technologies.

### The IPCEI-CIS AVANT Project

This thesis has been developed in collaboration with the Engineering Ingegneria Informatica S.p.A.[2], as part of the European project IPCEI-CIS AVANT[3], funded by the European Union with program NextGenerationEU.

IPCEI stands for Important Project of Common European Interest. These are largescale, strategic projects co-funded by multiple EU Member States and approved under EU state-aid rules, to support innovation, infrastructure, or technologies of major EU importance. CIS means Cloud Infrastructure and Services. IPCEI-CIS is the IPCEI focused on developing next-generation cloud and edge computing infrastructures and services in Europe. AVANT (an acronym from dAta and infrastructural serVices for the digitAl coNTinuum) is one of the projects under the umbrella of IPCEI-CIS. It is led by Engineering and aims to deliver advanced cloud-to-edge technologies, flexible infrastructures, interoperability, and open source components.

This thesis contributes to the project with the design and implementation of an advanced Network Manager integrated into the Liqo platform, which provides the granular isolation perimeters necessary to secure multi-tenant workloads across the distributed cloud-to-edge continuum.

# Chapter 2

# Background

To achieve the goal of building a secure, multi-tenant Kubernetes controller for Liqo, a deep understanding of the underlying network stacks and orchestration tools is imperative. This chapter is dedicated to exploring the fundamental technological background required for this work. It provides a comprehensive analysis of the concepts and tools that form the basis of the proposed implementation, specifically covering:

- Programmable Networking in Linux Environments: An in-depth examination of the tools utilized for network routing and security enforcement, including iptables, nftables, eBPF, and their underlying data structures (sets and maps).
- The Computing Continuum and Liqo: An overview of the continuum paradigm and Liqo's specific architecture and dynamic peering mechanisms.
- Kubernetes Architecture: A detailed look at the control plane, CNI plugins, network policies, and a comparison of dataplanes (nftables vs. eBPF).

## 2.1 Programmable Networking in Linux Environments

The evolution of cloud-native architectures and the growing density of microservices in Kubernetes clusters have forced a radical transformation of traffic management paradigms in the Linux kernel. In this context, the Linux network stack no longer acts as a simple paper pusher, but has evolved into the beating heart of Software Defined Networks (SDNs), providing the necessary hooks to abstract the hardware and implement dynamic routing logic via software.

To understand the connectivity mechanisms proposed in this thesis, it is essential to analyze the state of the art of networking in the kernel, focusing on the transition

from legacy models based on static chains to truly programmable and scalable execution engines: `nftables` and `eBPF`. These technologies allow the intelligence of the network to be moved directly into the data plane, ensuring the flexibility and performance required by modern virtualized infrastructures.

### 2.1.1 From Iptables to Nftables

For over two decades, `iptables` has been the mainstay of packet filtering in Linux. However, its internal architecture, designed for less dynamic scenarios, has obvious structural limitations when applied to orchestrators such as Kubernetes, where the number of endpoints can vary by the thousands in a matter of seconds.

The main limitation of `iptables` lies in its sequential execution model. Each packet must pass through a series of chains and rules in a linear fashion ($O(n)$), which introduces latency proportional to the number of services defined in the cluster.

`Nftables`, introduced as a modern successor, breaks this paradigm by adopting an architecture based on a kernel-internal virtual machine (VM). Instead of evaluating hard-coded rules, the `nftables` framework compiles user-supplied expressions into optimised `bytecode`, which is then executed by the kernel VM. This approach ensures:

- Reduced code duplication: A single rule can perform multiple actions on different protocols without having to replicate the entire matching logic.

- Fetching efficiency: The VM operates on registers, minimising memory access during packet inspection.

Another key evolution regards atomicity and transactional updates, essential in multi-tenant and dynamic contexts such as those enabled by Liqo where network state consistency is critical. `iptables` suffers from atomicity issues: to update a single rule, the entire set must be read, modified, and rewritten (read-modify-commit), a process that can lead to race conditions and packet loss during massive rule reloading.

`nftables` introduces the concept of transaction-based atomic operations. Using the `netlink` API, it is possible to send an entire batch of changes (additions, removals, replacements) that the kernel applies as a single atomic operation. This ensures that the data plane is never in an inconsistent state, a fundamental requirement for ensuring high service reliability during network reconfiguration phases.

### 2.1.2 Scalability through Sets and Maps

The most significant feature of `nftables` in terms of scalability is the native integration of `sets` and `maps`.

While in `iptables` matching a source IP against a list of 1,000 addresses would require 1,000 separate rules evaluated sequentially, in `nftables` this operation is delegated to an optimised data structure (usually a hash set).

In computational terms, we move from linear complexity $O(n)$ to constant complexity $O(1)$ (or logarithmic complexity $O(\log n)$ for ranges). For a network architecture that has to manage peering between distributed clusters, the use of maps allows NAT and routing logic to be implemented with almost zero overhead, regardless of the size of the data set.

### 2.1.3 eBPF

While `nftables`represents the evolution of the Netfilter framework, `eBPF` introduces an even more profound paradigm shift: the transformation of the kernel from a fixed-function system to an event-driven programmable platform.

eBPF allows sandboxed programs to be executed within the kernel without the need to recompile the kernel itself or load potentially unstable kernel modules. Each `eBPF` program undergoes a rigorous verification process (verifier) that ensures there are no infinite loops, unauthorised memory accesses or system crashes, guaranteeing operational security superior to any traditional kernel module.

In the context of networking, `eBPF` can be connected to different "hooks" (attach points) in the packet path:

- `XDP` (eXpress Data Path): This is the earliest point of interception, located directly in the network interface card (NIC) driver. `XDP` allows packets to be processed (or discarded) even before the kernel's `sk_buff` (socket buffer) data structure is allocated, enabling near wire-speed performance and resistance to traffic spikes (e.g., DDoS mitigation) that is unattainable by Netfilter.
- `TC` (Traffic Control): Operating after the allocation of the `sk_buff`, `eBPF` programs connected to TC have access to all packet metadata and can handle more complex ingress/egress logic, which is essential for monitoring and manipulating traffic at the container level.

Although the implementations discussed in this thesis make extensive use of Netfilter for reasons of universal compatibility across different Linux distributions, understanding `eBPF` is crucial since many modern CNIs (Container Network Interfaces), such as `Cilium`, use `eBPF` to completely bypass the traditional network stack, improving throughput and reducing inter-pod traffic latency. This aspect will be further analyzed in the following chapters.

### 2.1.4 Coexistence between the two models

A critical aspect analysed in this work concerns the coexistence of these technologies. Extreme attention must be paid to the order of execution: a packet handled by an `eBPF` program connected to an `XDP` hook may never reach the `Netfilter` hooks.

If an underlying CNI (such as Cilium in "full BPF" mode) is handling traffic, the `nftables` rules defined by other components may be bypassed. This thesis explores how to ensure that network policies and tunnelling logic remain consistent even in hybrid environments, where the data plane is fragmented between legacy and programmable execution engines.

### 2.1.5 Technology summary

| Feature | Iptables | Nftables | eBPF |
|---|---|---|---|
| **Execution Model** | Linear chains | Virtual Machine (Bytecode) | JIT-compiled programs |
| **Matching Complexity** | $O(n)$ | $O(1)$ / $O(\log n)$ | $O(1)$ / $O(\log n)$ |
| **Atomicity** | No (Read-Modify-Write) | Yes (Netlink Transactions) | Yes (Map updates) |
| **Intervention Point** | Netfilter Hooks | Netfilter Hooks | XDP, TC, Sockets, Kprobes |
| **Flexibility** | Low (Predefined actions) | Medium (Combinable expressions) | Maximum (C-like language) |

**Table 2.1:** Comparison of Linux packet filtering technologies.

This foundation in Linux networking now allows us to analyse how these tools are orchestrated to overcome the boundaries of a single Kubernetes cluster, which is the central objective of the Liqo project described in the following chapters.

## 2.2 Introduction to Kubernetes

Kubernetes, colloquially abbreviated as K8s, stands as the paramount platform for automating the deployment, scaling, and operations of application containers across clusters of hosts. Originally architected by Google based on over a decade of experience running production workloads at scale—most notably the Borg and Omega systems—it was subsequently open-sourced in 2014 and is currently maintained by the Cloud Native Computing Foundation (CNCF).

At its core, Kubernetes acts as a container orchestrator: a system designed to abstract the underlying hardware infrastructure, presenting a unified Application Programming Interface (API) to the user. This abstraction decouples the application logic from the physical or virtual machines upon which it executes, thereby facilitating a paradigm shift from machine-centric operations to application-centric management.

### 2.2.1 Cluster architecture

The architecture of Kubernetes is designed as a highly resilient and scalable distributed system, structured according to a microservices model that logically separates management and control functions from the actual execution of workloads. Basically, a Kubernetes cluster is divided into two interdependent macro-areas: the Control Plane, which acts as the decision-making “brain” of the system, and the Worker Nodes, which are the physical or virtual computational units responsible for hosting the containers.

The design is based on a declarative approach and a continuous reconciliation cycle (control loop), where the system constantly monitors the current state of the infrastructure and intervenes autonomously to align it with the desired state defined by the user. This abstraction allows the application to be decoupled from the underlying hardware, ensuring portability, high availability, and automated resource management in cloud-native environments.

The main components that make up the backbone of the cluster are summarized below:

- **kube-apiserver:** Represents the single point of entry for all administrative and management operations within the cluster, acting as the front-end for the Control Plane. It exposes a RESTful interface through which users, internal components, and external services communicate; every request for modification or status consultation must pass through this component, which authenticates, authorizes, and validates calls before persisting the data in the cluster database. Its horizontally scalable architecture allows it to handle high traffic volumes, ensuring the operational consistency of the entire ecosystem.

- **etcd:** This is a distributed, high-availability key-value datastore used as persistent storage for all configuration data and cluster status. In Kubernetes, etcd acts as the “single source of truth,” storing crucial information such as network topology, node status, and running workload specifications. Given its critical importance, it is typically implemented in a quorum configuration to prevent data loss and ensure operational continuity even in the event of partial infrastructure failures.

- **kube-scheduler:** This component is responsible for monitoring newly created objects (Pods) that do not yet have an assigned node and selecting the most suitable computational unit for their execution. The selection process is carried out by evaluating multiple factors, including hardware resource requirements, affinity or anti-affinity constraints, security policies, and local storage availability. The primary goal of the scheduler is to optimize resource allocation in the cluster, avoiding bottlenecks and ensuring that each workload operates in an environment compatible with its specifications.

- **kube-controller-manager:** This is the daemon responsible for managing the control loops that regulate the state of the cluster, grouping several logical functions into a single process to reduce complexity. It contains various controllers, such as the Node Controller, which monitors the health of the nodes, or the Deployment Controller, which ensures that the number of replicas of an application matches the declared number. Acting as an automatic adjustment mechanism, the controller manager detects discrepancies between the actual and desired states and activates the necessary procedures to correct anomalies without human intervention.

- **kubelet:** This is the fundamental agent installed on each node of the cluster and is responsible for ensuring that the containers described in the PodSpecs are actually running and healthy. The kubelet communicates directly with the Control Plane to receive instructions and constantly sends reports on the status of containers and the resources used by the node; it does not manage containers that were not created by Kubernetes, acting exclusively as an executor of centralized directives and interfacing with the container runtime to start or stop the necessary processes.

- **kube-proxy:** It handles networking management at the node level, implementing network rules that allow communication between Pods and to the outside world. This component keeps routing tables up to date and manages TCP, UDP, and SCTP traffic load balancing, translating the logical abstractions of Kubernetes "Services" into actual IP rules (often using iptables or IPVS). its presence is essential to ensure that each service is consistently reachable, regardless of the physical location of the Pods that compose it within the cluster.

- **Container Runtime:** This is the software responsible for the physical execution of containers on the node, managing their lifecycle from the extraction of images from repositories to their isolated execution. Kubernetes supports several runtimes that implement the Container Runtime Interface (CRI), such as containerd or CRI-O, allowing for technological flexibility that adapts to

different security and performance requirements. This layer acts as an intermediary between the logical orchestration of Kubernetes and the isolation features offered by the underlying operating system kernel.

### 2.2.2 Cloud-Native Networking: the CNI

The pervasive adoption of microservices architectures and the transition to cloud computing have redefined the fundamental paradigms of IT infrastructure management. Orchestration platforms such as Kubernetes have emerged as the standard for automation, scalability, and resilience of containerised workloads. However, the ephemeral nature of containers and the density of distributed workloads introduce significant challenges in terms of connectivity, security, and observability, rendering traditional approaches to network security often inadequate or ineffective.

In this context, the network subsystem is no longer a mere transport medium, but a critical component that must ensure isolation, encryption, and granular visibility without compromising performance. The management of these features is delegated to the Container Network Interface (CNI), a standard that abstracts the complexity of the underlying networking, allowing the integration of different solutions (plugins) within the Kubernetes ecosystem.

The Kubernetes network model is designed to eliminate the complexity of port mapping between hosts and containers by enforcing a flat network architecture. The key principles of this model are:

1. Each Pod must have its own unique IP address within the cluster.
2. All Pods must be able to communicate with each other across nodes without the use of Network Address Translation (NAT).
3. The IP address that a Pod “sees” on itself must match the IP address that other Pods see it with.

Although Kubernetes provides built-in plugins such as `Kubenet` or `Bridge` for basic functionality, these often lack the advanced capabilities needed in complex production environments, such as support for advanced network policies, in-transit encryption, and deep observability. As a result, choosing a third-party CNI becomes a fundamental architectural decision that directly affects the scalability and security posture of the entire cluster.

### 2.2.3 Network Policies

A fundamental characteristic of the native Kubernetes networking model is its default "allow-all" policy, which permits unrestricted communication among all Pods

within a cluster. In multi-tenant or security-critical environments, this inherently permissive architecture introduces significant vulnerabilities, as the compromise of a single service can facilitate unimpeded lateral movement across the entire infrastructure.

To mitigate these risks, Network Policies are used, which are software-defined constructs that act as distributed firewalls. They allow declarative rules to be defined to isolate workloads, specifying which traffic flows are authorized. In practice, network policies are applied using label selectors to group pods and define filters based on:

- L3/L4 (IP and Ports): Restriction of ingress and egress traffic based on IP addresses, namespaces, or pod labels.
- L7 (Application Layer): In more advanced implementations (often enabled by `eBPF` or Service Mesh), granular control over protocols such as HTTP or gRPC.

The use of these policies transforms the network from a static entity to a "Zero Trust" system, where every interaction must be explicitly allowed, ensuring granular traffic segmentation without sacrificing the flexibility typical of cloud-native environments.

Although Kubernetes defines a standard, declarative syntax for Network Policies[4], their actual implementation and enforcement of rules in the data path are delegated to the Container Network Interface (CNI) installed in the cluster. This implies that, while behavior remains consistent and predictable for all use cases formally defined by Kubernetes specifications, discrepancies may arise in non-standardized scenarios. A prime example concerns supported protocols: the Kubernetes standard guarantees control of TCP, UDP, and SCTP traffic. As a result, some CNIs strictly adhere to this triad, allowing other protocols such as ICMP (ping) to pass through, while other more advanced CNIs extend these capabilities by allowing the entire network stack, including ICMP, to be blocked to ensure total isolation, by working at network level (L3). This variability underscores how the choice of network plug-in is not just a matter of performance, but defines the effective security perimeter of the infrastructure.

The following example shows a policy that applies a Zero Trust model to a database, allowing access only on port 5432 (PostgreSQL) and only from Pods labeled as frontend.

```
1 apiVersion: networking.k8s.io/v1
2 kind: NetworkPolicy
3 metadata:
4   name: db-allow-frontend
5   namespace: prod
```

```
6  spec:
7    podSelector:
8      matchLabels:
9        app: database  # Target: the pods with this
     label
10   policyTypes:
11   - Ingress
12   ingress:
13   - from:
14     - podSelector:
15         matchLabels:
16           role: frontend  # Only pods with this
     label can connect
17     ports:
18     - protocol: TCP
19       port: 5432  # Access limited to this single
     port
```

### 2.2.4 Evolution of Data Planes: From iptables to eBPF

The efficiency with which a CNI implements the Kubernetes network model depends heavily on the technology used in the data plane. Historically, most implementations have relied on the Linux kernel's `Netfilter` framework, configured via `iptables`. Although reliable, the `iptables` architecture has structural limitations in large-scale environments. Significant differences in terms of flexibility and performance were addressed in the previous chapter.

`Nftables` was introduced as a modern successor, offering $O(1)$ or $O(\log n)$ matching complexity through the use of verdict maps and indexed sets, solving many of the scalability issues of iptables. Starting with Kubernetes 1.29, an `nftables` mode was introduced for `kube-proxy`, marking a step towards abandoning legacy technologies.

Parallel to the evolution of Netfilter, `eBPF` (Extended Berkeley Packet Filter) technology has introduced a radical paradigm shift. `eBPF` allows sandboxed programs to run directly in the Linux kernel, enabling packet processing at very low levels of the stack, such as the `XDP` (eXpress Data Path) hook at the network driver level.

eBPF-based CNIs, such as `Cilium`, eliminate the need to traverse the entire Netfilter and `iptables` stack, drastically reducing context switching and CPU overhead[5]. This approach not only enables superior performance in terms of

throughput and latency, but also enables advanced features such as identity-aware filtering and sidecar-free observability. Benchmarks indicate that `eBPF` can maintain high throughput even with complex L3/L4 policies, while solutions based on `iptables` suffer significant degradation (60-70% reduction under load).

Currently, the major CNIs use `nftables` or `eBPF` for obvious reasons of flexibility in writing rules, often allowing the data plane to be implemented with one tool rather than another. This is another very important aspect that is taken into consideration when choosing which CNI to install in your Kubernetes cluster.

### 2.2.5 CNIs comparison

The CNI landscape offers several solutions, each with specific trade-offs between architecture, security, and performance. The following are the ones that will be analyzed in this thesis.

#### Flannel

Flannel is one of the longest-standing and most minimalist solutions in the Kubernetes landscape. Originally designed by CoreOS, its primary goal is to provide a Layer 3 (L3) connectivity layer by configuring a network fabric common to all nodes in the cluster.

Architecture and Data Plane: Flannel primarily operates as a binary agent (flanneld) installed on each node, responsible for allocating an IP subnet for each host. It supports several encapsulation mechanisms (called backends):

- VXLAN: The default and most widely used backend, which creates a Layer 2 overlay network on top of the L3 infrastructure.
- host-gw: A high-performance option that avoids encapsulation by using direct IP routes, but requires all nodes to be on the same L2 network.
- UDP: Used exclusively for debugging purposes or on legacy architectures.

Although it stands out for its ease of use and low memory requirements, Flannel does not natively support Kubernetes Network Policies. To implement traffic isolation, it must often be paired with external solutions such as Calico (in "Canal" mode).

#### Calico

Calico (developed by Tigera) is considered one of the main choices for production environments that require high scalability and granular security management.

Unlike Flannel, Calico does not just provide connectivity, but implements an advanced policy engine.

Its defining characteristic is a multi-dataplane architecture, which enables the selection of a forwarding mechanism that best aligns with the specific operational context:

- Standard Linux (iptables): The classic dataplane based on the kernel's bridge module and iptables rules for filtering.
- eBPF: A modern dataplane that bypasses the limitations of iptables, offering superior performance, lower CPU overhead, and preservation of the source IP address in load balancing.
- VPP (Vector Packet Processing): Ideal for very high throughput scenarios, where packet processing takes place entirely in user space via the DPDK library.
- nftables: The modern successor to iptables that provides a more efficient classification engine and reduced rule-set complexity by using a single framework for IPv4, IPv6, and ARP.

Calico uses the BGP (Border Gateway Protocol) to distribute routes between nodes. This allows for easy integration with physical network infrastructure (Top-of-Rack switches), enabling Pods to be reachable without the use of overlay networks (encapsulation), thus reducing latency.

#### Cilium

Cilium is a cloud-native networking and security solution that places eBPF (extended Berkeley Packet Filter) technology at the center of its architecture. It is designed to overcome the scalability limitations of iptables-based technologies, especially in large clusters.

It uses eBPF to insert logic programs directly into strategic points in the Linux kernel (such as TC or XDP hooks). This allows networking, load balancing, and security to be managed without ever leaving the kernel context, ensuring near wire-speed performance.

This CNI can completely replace kube-proxy, managing the load balancing service via eBPF maps instead of long chains of iptables or nftables.

Unlike traditional IP address-based firewalls (which are ephemeral in Kubernetes), Cilium enforces policies based on logical identities (labels). This ensures that security rules remain constant even if Pods are restarted with new IP addresses. It also offers deep visibility at Layer 7 (HTTP/gRPC/Kafka) through integration with the Envoy proxy.

**Summary**

| **Feature** | Flannel | Calico | Cilium |
|---|---|---|---|
| **Data Plane** | VXLAN, host-gw | iptables, eBPF, VPP | eBPF |
| **Routing Protocol** | Static / Overlay | BGP / Overlay | Overlay / Direct Routing |
| **Network Policies** | No | Yes (L3/L4) | Yes (L3/L4/L7) |
| **Observability** | Low | Medium | High (Hubble) |
| **Use cases** | Simple clusters / Testing | Enterprise / Multi-cloud | Performance / Security / Telco |

**Table 2.2:** CNIs comparison summary

## 2.3 Computing Continuum

The evolution of ICT infrastructure over the last decade has been characterised by a fundamental paradigm shift: the move away from a purely cloud-centric model in favour of a distributed and pervasive approach known as Computing Continuum (or Cloud-to-Edge Continuum). While the traditional Cloud Computing model has offered undeniable advantages in terms of elastic scalability, resource consolidation and operational efficiency (OpEx), the explosion of IoT (Internet of Things) devices and the massive generation of data at the edge of the network have highlighted the physical and logical limitations of centralisation. Even with the advent of the massive use of AI, there has been a need to perform complex calculations in a central infrastructure, leaving lighter calculations to the edge, for example to save energy on the edge devices.

Stringent latency constraints, bandwidth limitations, data sovereignty requirements and the need for real-time inference make the indiscriminate transfer of every byte to Hyperscale Data Centres impractical. The Computing Continuum therefore emerges as the infrastructural convergence that unifies Cloud and Edge into a single logical topology, allowing workloads to be executed at the most appropriate point based on business requirements and performance metrics.

### 2.3.1 Architecture

The Computing Continuum should not be understood as a simple juxtaposition of hierarchical tiers, but rather as a continuous computational fabric. In it, computing,

storage, and networking resources are abstracted and presented as a unified pool, despite their underlying heterogeneity.

We can segment the Continuum into three logical macro-levels, although the boundaries are becoming increasingly fluid:

1. Central Cloud (Core): Characterized by virtually "infinite" resources, high availability and massive processing capacity. It is the ideal place for training complex Machine Learning models, data warehousing and global orchestration.

2. Near Edge / Regional Edge: Typically located at ISP points of presence (PoPs) or regional data centers (Multi-access Edge Computing - MEC). It offers a compromise between the computational capacity of the cloud and proximity to the end user, reducing network latency (RTT).

3. Far Edge / Extreme Edge: Includes on-premise gateways, industrial servers, embedded devices, and smart sensors. Here, resources are severely constrained (CPU, RAM, power), but latency is deterministic and close to zero. This is the domain of data acquisition and critical real-time inference.

The primary engineering challenge in realizing the Continuum lies in managing heterogeneity. Unlike a homogeneous data center, the Continuum is composed of mixed CPU architectures (x86_64, ARM64, RISC-V), variable network interfaces (Fiber, 5G, LPWAN, industrial Ethernet) and uncertain levels of reliability.

### 2.3.2 Kubernetes in the Computing Continuum

To manage this complexity, the need for a unified control plane emerged. Kubernetes has established itself as the main player for container orchestration, evolving from its original role as a cloud-native cluster manager to one of the leading solutions for managing the Computing Continuum control plane.

The choice of Kubernetes as the technological substrate for the Continuum is motivated by characteristics intrinsic to its architecture:

- Declarative Abstraction: The Desired State-based management model allows the user to define "what" needs to be done, leaving it to the control plane to figure out "how" and "where", a crucial feature in dynamic and fallible environments such as the Edge.

- Immutability and Containerisation: The use of containers ensures that the application and its dependencies are packaged atomically, eliminating the problems of "environment drift" between the development data centre and the Edge gateway in production.

- Extensibility (CRD and Operator Pattern): The ability to extend Kubernetes APIs via Custom Resource Definitions (CRDs) allows non-native resources (e.g., physical sensors, industrial protocols such as Modbus or OPC-UA) to be modelled as standard Kubernetes objects.

Despite its theoretical suitability, the "vanilla" implementation of Kubernetes presents significant overhead for Edge Computing (e.g., etcd resource consumption, verbosity of communications between kubelet and API server). To achieve an effective Continuum, specific architectural strategies must be adopted:

- Lightweight Distributions: The use of optimised distributions such as `K3s` or `MicroK8s`, which reduce memory footprint by removing legacy drivers and unifying control plane processes.

- Edge-Native Architectures (e.g. `KubeEdge`): Solutions that decouple the control plane (in the cloud) from the data plane (at the edge) using optimised protocols (e.g. `WebSocket`) to handle intermittent connections and tolerate prolonged network partitions without evicting pods (tolerance to `NotReady` state).

- Centralized Control Plane: Use a single cluster or a single control plane for multiple tenants, thereby reducing the use of resources linked to the orchestrator and dedicating available resources to truly important computational loads.

### 2.3.3 Orchestration

Achieving the Computing Continuum means implementing advanced scheduling logic that goes beyond simple resource bin-packing. The scheduler must be aware of the network topology and geographical characteristics.

In a Continuum scenario, a single geographically distributed cluster is rarely used due to the latency of the etcd consensus algorithm (Raft). The prevailing approach is Multi-Cluster, where independent clusters (Core, Regional, Edge) are coordinated by a higher management plan (e.g. Liqo, Karmada, Red Hat ACM, Google Anthos).

This meta-level of orchestration allows security policies to be propagated and applications to be deployed across fleets of thousands of edge clusters, treating them as grouped logical entities.

The value of Continuum is realized when the system is able to dynamically decide where to place the load. Decision metrics for the extended Kubernetes scheduler include:

- Network Latency: Place latency-sensitive pods (e.g. robotic control) in the node closest to the data source.
- Data Gravity: Move computation to the data (Edge) to avoid egress costs and bandwidth saturation, or move data to computation (Cloud) if historical aggregation is required.
- Specialized Hardware: Leverage nodes with specific hardware accelerators (TPU, GPU, NPU) available only in certain tiers of the continuum for AI inference tasks.

### 2.3.4 Applications

The application of the Continuum paradigm enabled by Kubernetes is reflected in critical sectors where IT/OT (Information Technology/Operational Technology) convergence is a priority.

- Smart Manufacturing (Industry 4.0): Kubernetes orchestrates containers that perform predictive analytics on machinery directly in the factory (Far Edge), ensuring millisecond reaction times for emergency stops, while logs are sent asynchronously to the Cloud for model re-training.
- Telco Cloud and 5G: Modern 5G network architectures (Open RAN) are based on virtualised network functions (CNF - Cloud Native Network Functions) running on distributed Kubernetes clusters. Here, Continuum allows the User Plane Function (UPF) to be moved extremely close to the user to enable ultra-low latency services.
- Automotive and V2X: Autonomous vehicles act as mobile clusters, synchronising critical data with roadside infrastructure (Roadside Units) managed via Kubernetes edge nodes, creating a dynamic continuum between vehicle, infrastructure and central cloud for traffic management.

In conclusion, the Computing Continuum is not just a physical extension of infrastructure, but a new operating model. Kubernetes, thanks to its modular architecture and large open source community, stands as the main enabler of this transformation. However, as we will see in the following chapters, the use of Kubernetes on a multi-tenant environment introduces new complexities in terms of security and connectivity management.

## 2.4 Liqo

Liqo is an open-source solution designed to enable dynamic and seamless multi-cluster Kubernetes topologies, supporting heterogeneous infrastructures ranging

from on-premises to cloud and edge computing. The primary goal of the platform is to extend the control plane and data plane of a Kubernetes cluster beyond its physical boundaries, allowing workloads to be executed on remote clusters transparently and without requiring any changes to applications or standard Kubernetes APIs.

Liqo's architecture is based on the concept of peering[6], a (typically unidirectional) resource consumption relationship between a "consumer" cluster, which requires computational capacity, and a "provider" cluster, which offers it. To support the offloading of computations[7], Liqo implements a network fabric that extends the Kubernetes network model, ensuring pod-to-pod communication and resource reflection (such as Services, ConfigMaps and Secrets) through secure VPN tunnels, automatically managing the complexities associated with IP address overlap and inter-cluster routing[8].

### 2.4.1 The network fabric

Liqo's inter-cluster connectivity solution is based on the concept of Network Fabric, a subsystem designed to transparently extend the Kubernetes network model across independent and heterogeneous clusters. The primary goal is to ensure that pods, regardless of their physical location (local or offloaded to a remote cluster), can communicate with each other as if they belonged to the same logical network, while preserving compatibility with the underlying CNI (Container Network Interface) plugins.

The clusters are interconnected by creating a secure VPN tunnel between them, usually Wireguard, but Liqo allows any type of tunnel to be created. A gateway is created in each cluster, which is the point through which all traffic directed to the other cluster passes. This gateway is also connected to each node in the cluster on which it is installed via tunnels through which traffic arriving from the VPN tunnel passes.

Liqo's Network Fabric addresses the complexity of cross-cluster communication by dividing the architecture into two distinct but interconnected logical segments: the internal network, which manages intra-cluster traffic to the gateway, and the external network, which is responsible for the secure transport of packets between federated clusters. This separation allows for abstraction of local network specifics, facilitating the management of IP addressing conflicts (PodCIDR) through dynamically managed NAT (Network Address Translation) mechanisms.

To enable traffic arriving from the remote cluster to reach the correct destination pod, when this traffic arrives at the gateway from the remote cluster, it must be encapsulated with tunnels connected to each node. Liqo uses the GENEVE (Generic Network Virtualization Encapsulation) protocol to do this.

In this architecture, each node in the cluster establishes a dedicated Geneve

tunnel to the gateway pod. As a result, the number of active tunnels is directly proportional to the number of physical nodes in the cluster, ensuring scalability independent of the number of pods running. The Geneve interface encapsulates Ethernet (L2) traffic within UDP datagrams.

This approach is essential to ensure transparent traversal of the CNI. The outer packet has the node's IP as its source address and the gateway's IP as its destination. To the CNI, this appears as legitimate traffic generated by the node itself, allowing the encapsulated packet to be routed correctly to the gateway without undergoing unwanted alterations.

A significant technical challenge in using UDP tunnels such as Geneve lies in the interaction with Kubernetes services and NAT mechanisms. As specified in RFC 7348, Linux calculates the UDP source port by hashing the fields of the inner packet to facilitate load balancing (ECMP). This means that the 5-tuple (protocol, source/destination IP, source/destination port) varies dynamically for each connection, making it impossible to traverse intermediate NATs that rely on static connection tracking tables.

To overcome this limitation, Liqo's Geneve tunnels cannot terminate on virtual endpoints such as ClusterIPs or LoadBalancers, but must be established directly on the real IP addresses of Pods or Nodes, which the CNI is able to manage without applying NAT. This implies the need for dedicated controllers that dynamically monitor the actual IP address of the Gateway to update the routes on the nodes.

### 2.4.2 The external network

Traffic between separate clusters (external network) is transported using a VPN tunnel, by default with `WireGuard` protocol (chosen for its high performance and encryption capabilities), but the administrator can choose and configure any tunnel.

For each active peering, a dedicated Gateway pod is instantiated, ensuring connection isolation. Gateways operate in two distinct roles:

- Gateway-Client: Hosted on the consumer, initiates the connection to the remote cluster.
- Gateway-Server: Hosted on the provider, accepts incoming connections and is exposed via a Kubernetes Service (LoadBalancer or NodePort).

The Gateway pod acts as a junction point between the internal and external networks, managing three types of interfaces: the default interface (for external connectivity), the Geneve interfaces (one for each local node) and the WireGuard interface (tunnel to the remote cluster).

Traffic from local pods arrives at the Gateway encapsulated in the Geneve tunnel. Inside the Gateway pod, the packet is decapsulated and reintroduced into

the network stack. At this point, the system applies policy routing rules to direct packets destined for the remote cluster to the WireGuard interface. WireGuard, operating at Layer 3 (L3), re-encapsulates the IP packet, encrypts it, and transmits it to the remote Gateway endpoint through the pod's default interface.

In the event of address space overlap (identical PodCIDRs between clusters), the Gateway performs DNAT (Destination NAT) and SNAT (Source NAT) operations to remap the original IP addresses to virtual CIDRs negotiated during peering, allowing conflict-free communication.

Figure 2.1 shows the flow of a packet from one cluster to another one:

1. The packet leaves the pod, and the routing table on the node route it through the GENEVE tunnel
2. The packet is encapsulated with GENEVE and reaches the other end of the tunnel at the gateway
3. The gateway extracts the packet arriving from the GENEVE tunnel and encapsulates it to send it through the VPN tunnel to reach the gateway of the other cluster
4. The other gateway receives the packet and analyzes the destination IP to route it to the GENEVE tunnel connected to the node hosting the destination Pod
5. The packet reaches the destination node and enters the Pod

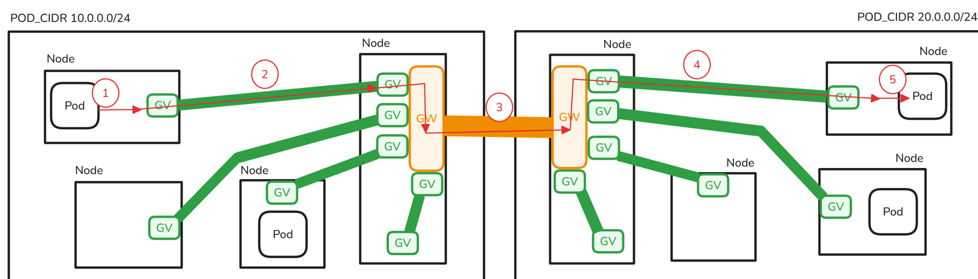


**Figure 2.1:** Example liqo traffic flow

### 2.4.3 FirewallConfiguration

Advanced traffic management, including packet manipulation to overcome CNI limitations and security, is orchestrated through the Custom Resource Definition (CRD) called `FirewallConfiguration`. These configurations define sets of `nftables` rules that are reconciled by dedicated controllers.

To fully understand the application of these rules, it is essential to introduce the concept of Linux Network Namespace. In Kubernetes, each Pod has its own isolated network namespace, while processes running on the node (including system network components) typically operate in the root network namespace. Liqo applies its FirewallConfigurations in different contexts depending on the function: some rules must be applied within the Gateway pod namespace, while others must operate at the node (fabric) level, in the root namespace.

The configurations applied on the nodes are critical to ensure the correct delivery of packets to the Gateway. A key example is the Masquerade Bypass configuration. Many CNIs (e.g. Azure CNI, Calico, KindNet) indiscriminately apply masquerading (SNAT) to traffic leaving the pod towards the node, altering the source IP address and replacing it with that of the node itself.

Since the Geneve tunnel requires hosts to communicate using the IPs assigned to their network interfaces, masquerading prevents the tunnel from being established correctly. FirewallConfiguration solves this problem by applying rules in the node's root namespace that intercept traffic directed to Geneve interfaces or coming from the Gateway, preventing the source IP from being altered through double SNAT techniques or specific exclusions.

Another fabric-level use case concerns the routing of NodePort services. Specific rules use connection tracking marks (ctmark) to associate each connection with the source node. This ensures that return traffic, once decapsulated by the Gateway, is correctly redirected to the node that originally received the external request, preserving routing symmetry.

A crucial consideration is that CNI plugins also enforce network policies within the root namespace, necessitating careful management to mitigate potential rule conflicts. For instance, CNI plugins that implement their data plane using extended Berkeley Packet Filter (eBPF) technology may circumvent these policies, as eBPF enables specific network traffic to bypass the standard Linux network stack entirely

The configurations applied within the Gateway pod namespace focus primarily on address translation and inter-cluster traffic management.

- CIDR Remapping: DNAT and SNAT rules are applied to translate "remapped" IP addresses (used locally to avoid conflicts) into the actual IP addresses of the remote cluster, and vice versa. For example, outbound traffic to the WireGuard tunnel undergoes post-routing SNAT, while inbound traffic destined for local pods undergoes pre-routing DNAT.

- External IP Remapping: IP resources generate firewall configurations to map local IP addresses to addresses belonging to the External CIDR, making specific services reachable from the peered cluster through static NAT rules.

In summary, the distinction between applying rules on the fabric (node) and on the gateway reflects the hybrid nature of Liqo's architecture: the fabric manages the

overlay and compatibility with the host infrastructure, while the gateway acts as an intelligent edge router, managing the business logic of multi-cluster connectivity.

# Chapter 3

# Problem analysis

The transition from monolithic cluster architectures to the computing continuum introduces profound security challenges, particularly regarding the maintenance of trust boundaries across administrative domains. As established in the preceding chapters, the Liqo framework enables the seamless peering of Kubernetes clusters, effectively flattening the network topology to allow unrestricted pod-to-pod communication. While this architectural model facilitates resource offloading and flexibility, it inherently dissolves the network perimeter, creating potential vulnerabilities in multi-tenant environments.

This chapter provides a rigorous analysis of the critical use cases addressed by the proposed *Liqo Network Manager*. Specifically, it examines the necessity for fine-grained traffic filtering in two primary directions: *Provider Protection*, ensuring the integrity of the hosting cluster against untrusted consumers, and *Consumer Protection*, safeguarding the originating cluster from potential compromises within the provider's infrastructure. These scenarios necessitate a hybrid enforcement strategy that transcends standard Kubernetes `NetworkPolicies`: this approach leverages low-level packet filtering via `nftables` to overcome the inherent lack of flexibility in `NetworkPolicy` specifications, which often struggle to address complex traffic patterns.

In this first phase of the project, the primary focus is on the firewalling component: the Network Manager must act as a programmable guardian, capable of intercepting network flows and applying filtering rules based on the offloading context. The ambition is not only to provide a static solution, but to create a flexible and adaptive framework that can be applied to different security scenarios depending on the specific needs of the players involved.

## 3.1 Use cases

In a federation of clusters, traffic no longer follows traditional linear north-south (client-server) or east-west (pod-to-pod within the same cluster) paths. A third dimension of inter-cluster connectivity is introduced, requiring granular control mechanisms capable of operating at the kernel level but instructed by Kubernetes logical definitions, even in multi-tenant scenarios.

The following analysis explores the main real-world use cases in which the application of firewalling rules becomes a necessary condition for system operation. These scenarios are not just theoretical possibilities, but constitute the fundamental requirements on which the Network Manager, the subject of this thesis, was designed. Although the proposed implementation focuses on these primary needs, the architecture has been designed to be inherently extensible, allowing it to accommodate custom policy models.

The two fundamental pillars of this analysis are Provider Protection and Consumer Protection, which reflect the symmetrical and bidirectional nature of trust in multi-clusters.

### 3.1.1 Provider Protection

In the context of the computing continuum, the “Provider” is defined as the cluster offering resources (CPU, RAM, storage) to external entities (“Consumers”) for the execution of offloaded workloads. If the provider operates in a trusted environment and consumers all operate in the same administrative environment, trusting each other, the default network policy (allow-all) is fine. However, if the provider supplies resources to consumers from different domains or owners, it must be able to set firewall rules to regulate traffic.

By default, the peering mechanism in Liqo establishes a Virtual Private Network (VPN) tunnel—typically using WireGuard—between the clusters and then encapsulates traffic within Geneve tunnels to facilitate cross-cluster communication. Without intervention, this architecture permits a pod in the Consumer cluster to route packets to any IP address within the Provider’s Pod CIDR.

This unrestricted connectivity poses a severe security risk. A malicious or compromised Consumer cluster could attempt to do:

- **Lateral Movement**: Initiate connections to the Provider’s system namespaces (e.g., `kube-system`), potentially accessing sensitive services such as the API server or internal databases.

- **Cross-Tenant Attacks**: In a multi-tenant environment, where a single Provider hosts workloads from multiple Consumers (e.g., Cluster A and Cluster

B), the default flat network would allow Cluster A to communicate with the offloaded namespaces of Cluster B, violating tenant isolation requirements.

The primary security objective in this scenario is to enforce a strict principle of least privilege, ensuring that a Consumer can access exclusively the resources it has offloaded, without obtaining network visibility into the Provider's internal infrastructure or the workloads of other tenants.

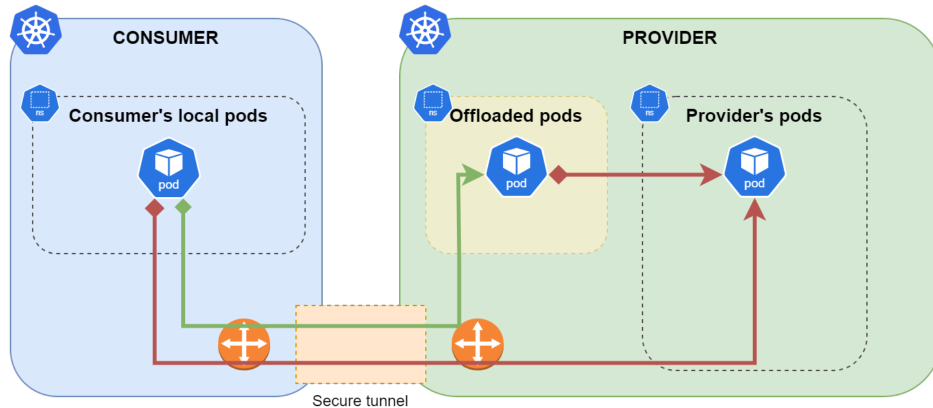


**Figure 3.1:** Provider Protection traffic rules

### 3.1.2 Consumer protection

While Provider Protection focuses on the integrity of the hosting infrastructure, the symmetrical use case, *Consumer Protection*, addresses the security of the originating cluster that initiates the offloading process. In this scenario, the Consumer cluster extends its logical boundaries to incorporate resources hosted on the Provider. Although the Provider is trusted computationally to execute delegated workloads, it must be treated as an untrusted environment from a network security perspective. The fundamental security imperative is to prevent the Provider infrastructure—or any compromised workloads residing within it—from initiating unauthorized connections back to the Consumer's internal network.

In a standard Liqo peering configuration, the network tunnel operates bidirectionally. Consequently, just as the Consumer can reach its offloaded pods, the Provider cluster gains routing visibility into the Consumer's Pod CIDR to facilitate return traffic and service reachability. This inherent bi-directionality introduces significant threat vectors:

- **Provider Compromise**: Should the Provider cluster be compromised, an attacker could exploit the established VPN tunnel to probe the Consumer's

internal network, actively scanning for vulnerabilities in local services or databases.

- **Malicious Workloads**: An offloaded pod, if subverted, could function as a jump host to launch lateral attacks against the Consumer's local infrastructure.

Consequently, the Consumer Protection policy dictates that the Provider must be strictly prohibited from establishing arbitrary connections to the Consumer's network. Incoming connections must be rigorously whitelisted based on the identity of the source workload.

The architectural requirement for Consumer Protection is nuanced: a rigid "deny-all" ingress policy is untenable, as legitimate communication flows are essential for the operation of distributed applications. For instance, an offloaded frontend service running on the Provider may legitimately require access to a backend database residing on the Consumer cluster.

To satisfy this requirement, the Network Manager must enforce a granular policy where incoming connections are rejected by default, unless they originate explicitly from the Consumer's own offloaded pods. This necessitates the following technical constraints:

- **Source-Based Whitelisting**: The filtering logic at the Consumer's Gateway must actively inspect the source IP addresses of incoming packets. Traffic should be permitted exclusively if the source IP matches the known IP addresses of the offloaded pods actively managed by that Consumer.
- **Stateful Inspection**: The firewall rules must be stateful to permit return traffic for connections initiated by the Consumer (e.g., responses to requests dispatched to an offloaded pod) while simultaneously blocking unsolicited connection attempts from the Provider.

### IP Spoofing

An additional critical vulnerability emerges from the reliance on source IP addresses for traffic authentication. While source-based whitelisting effectively filters standard traffic, it is inherently susceptible to IP spoofing if the remote environment is fully compromised. A malicious actor who gains administrative control over the Provider's gateway could craft packets with forged source IP addresses. By masquerading as legitimate offloaded pods, the attacker could effectively bypass the Consumer's ingress filters, potentially gaining unrestricted access to the Consumer's internal network.

This scenario highlights a fundamental dichotomy in the computing continuum: the unavoidable baseline of trust versus the necessity of Zero Trust network principles. By delegating the execution of workloads, the Consumer inherently extends a

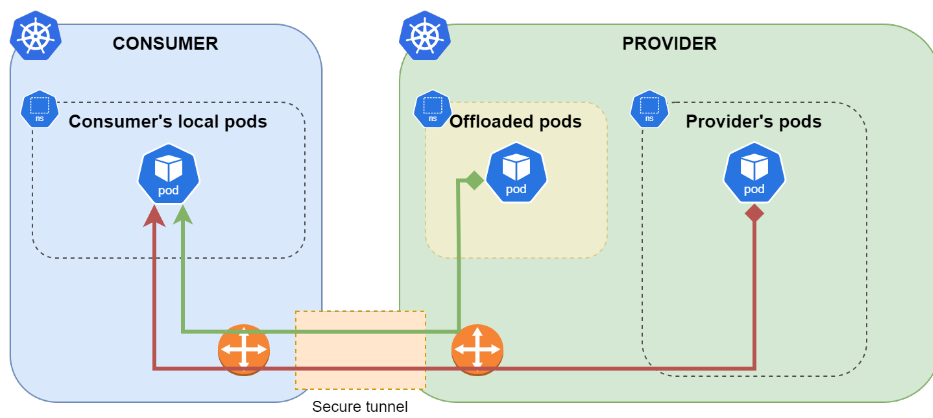


**Figure 3.2:** Consumer Protection traffic rules

degree of trust to the Provider's infrastructure. If a Provider is entirely malicious or deeply compromised, the integrity of the offloaded computation is already forfeit. Nevertheless, a security-conscious Consumer must operate under the assumption of potential compromise, aiming to contain the threat rather than relying solely on the Provider's perimeter defenses.

To mitigate the potential blast radius of such an attack vector, the Network Manager must support defense-in-depth strategies. Rather than universally trusting spoofable IP addresses across the entire local cluster, a risk-averse Consumer can implement layered firewalling rules. These policies would restrict incoming traffic originating from offloaded pods to strictly defined destinations, such as specific local namespaces or isolated resource groups. By ensuring that only a strictly necessary subset of the internal infrastructure is exposed to the remote environment, the Consumer drastically reduces the attack surface, preventing lateral movement even if the primary IP-based perimeter is breached.

### 3.1.3 Leaf-to-Leaf Protection

The complexity of multi-cluster topologies escalates significantly when a single Consumer cluster establishes peerings with multiple Providers simultaneously, thereby generating a "hub-and-spoke" architecture. In this configuration, the Consumer acts as the central routing hub, while the federated Providers represent the distinct "leaves". The Leaf-to-Leaf use case addresses the intricate security and routing challenges that emerge when a workload offloaded to one Provider requires communication with a workload offloaded to a different Provider via a Kubernetes Service.

From a networking standpoint, this scenario necessitates a transitive communication path: traffic originating from a pod in Provider A must traverse the Consumer's infrastructure before being correctly routed toward the destination pod in Provider B. The primary objective of Leaf-to-Leaf protection is to guarantee that legitimate inter-leaf traffic is not inadvertently dropped by overlapping security policies. Consequently, the Network Manager must be capable of recognizing this specific traffic pattern, enabling the Consumer to explicitly allow the transitive flow, while simultaneously configuring the receiving Provider to accept the ingress connections.

### 3.1.4 Multiconsumer Protection

In multi-tenant environments where a single Provider hosts resources owned by disparate, mutually untrusting Consumers, it is imperative to enforce strict network isolation to prevent cross-tenant pod-to-pod communication.

Conceptually, this scenario operates as an extension of the Provider Protection

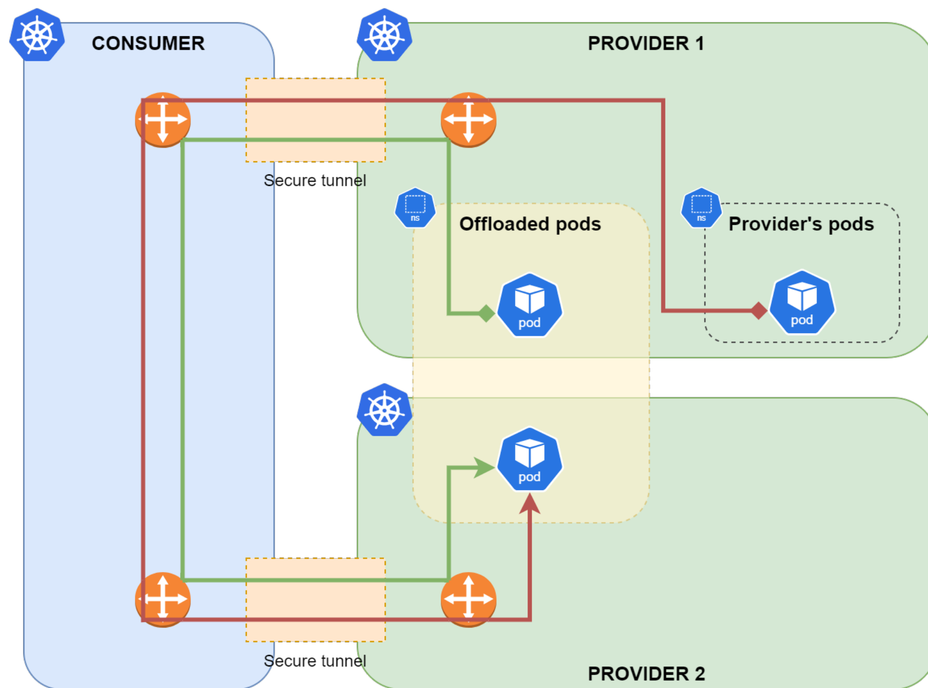


**Figure 3.3:** Leaf-to-Leaf traffic rules and transitive connectivity

model, which is fundamentally designed to block connections traversing outside the Consumer's authorized virtual perimeter. Nevertheless, explicitly addressing Consumer-to-Consumer isolation is a critical architectural requirement to validate the overall efficacy and robustness of the proposed solution. From a testing perspective, this distinction is vital because a Provider's native workloads might be subjected to different management and routing paradigms compared to the external workloads offloaded by a Consumer. The Network Manager must seamlessly navigate these varying contexts to ensure absolute tenant segregation.

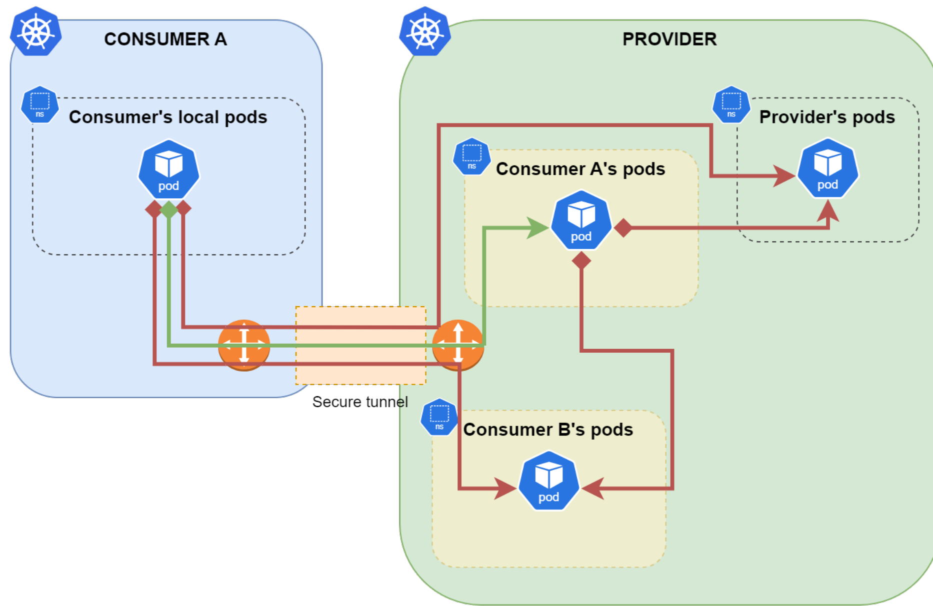


**Figure 3.4:** Consumer-to-Consumer firewalling

## 3.2 Firewalling solutions

The implementation of a network manager within the Liqo ecosystem requires careful evaluation of the traffic control tools available in the Kubernetes environment. Although NetworkPolicies are the standard tool for intra-cluster network segmentation, the challenges posed by multi-cluster offloading and resource federation require a more granular approach that is not constrained by the limitations of individual local orchestrators.

In this chapter, we will analyze the structural and functional differences between Kubernetes' native NetworkPolicies and the FirewallConfiguration resource introduced by Liqo.

### Liqo's tunnel compatibility

The first part of the study to determine the best tool to use to implement the firewalling system focused on the implications of the technologies used by Liqo.

Liqo uses GENEVE tunnels to transport traffic between the gateway and the cluster nodes, so the gateway Pod has several network interfaces: the default one for external connections, the VPN tunnel interface, and the GENEVE tunnel interfaces, one for each node in the cluster. Kubernetes NetworkPolicies only work with the default interfaces of the Pods, so it would not be possible to use them to regulate traffic leaving the Gateway. The policies, analyzing only the outermost L3 and L4 layers, regardless of the actual sender would always see the same source IP address: that of the gateway. This means that with NetworkPolicies alone, it would be impossible to create a common rule that limits gateway egress.

### Dependence on the Container Network Interface (CNI)

An important architectural distinction lies in the relationship with the network data plane.

NetworkPolicies are inherently CNI-dependent. Although the syntax is standardized at the API level, the actual enforcement of the rules falls entirely on the implementation of the installed network plugin (e.g., Calico, Cilium). This leads to significant behavioral discrepancies: for example, the Cilium implementation shows limitations in the use of "ipBlock" when referring to Pod IPs, while Calico, by default, may not filter ICMP traffic, allowing "ping" commands to be executed even when HTTP traffic is blocked[9].

In contrast, Liqo's FirewallConfiguration takes a CNI-agnostic approach. It interacts directly with the Linux kernel's netfilter framework, with the goal of having consistent behavior regardless of the network plugin used in the cluster. However, it should be noted that this approach requires careful interference management: advanced CNIs that use eBPF to bypass the traditional kernel network stack could theoretically circumvent the rules imposed via netfilter.

### Abstraction and Ease of Management

From the operator's point of view, the two resources operate at different levels of abstraction:

- NetworkPolicy (High-level): Offers simplified, declarative management. Through logical selectors such as "namespaceSelector" or "podSelector", it is possible to isolate entire groups of resources without knowing the underlying infrastructure details.

- FirewallConfiguration (Low-level): Requires a deeper understanding of packet flow. Being based on low-level rules, their configuration is more burdensome and less intuitive. In dynamic scenarios, such as adding new Pods to a namespace, FirewallConfiguration often requires a dedicated controller that translates Kubernetes events into new nftables rules in real time, ensuring that the enforcement always reflects the current state of the cluster.

### Flexibility and Expressiveness

NetworkPolicies are designed for a "basic" security model, limited mainly to filters on IP, ports, and selectors, with a default-deny logic. This rigidity prevents the creation of complex policies that require actions other than simple drop/allow or that need to intervene on specific kernel hooks.

FirewallConfiguration unlocks the full potential of nftables. It allows the operator to define complex rules on any network interface and on any routing hook (prerouting, input, forward, output, postrouting). This extensibility is fundamental for Liqo: should new filtering needs arise that have not yet been anticipated, the framework allows new features to be integrated into the Liqo core through direct contributions (Pull Requests), making the solution future-proof and adaptable to emerging solutions and requirements.

### Protocol and Encapsulated Traffic Management

One of the most critical limitations of NetworkPolicies in multi-cluster contexts is their inability to manage encapsulated traffic. Since NetworkPolicies operate at the Pod's virtual interface level, they have no visibility into traffic that is encapsulated in tunnels (such as Geneve, used by Liqo for peering). As a result, it is technically impossible to use a NetworkPolicy to regulate traffic exiting the tunnel or to inspect packets before they are decapsulated.

FirewallConfiguration, operating at the system level and on any interface (including peering tunnels), allows the administrator to precisely manage traffic that passes outside the direct control of the CNI. It can also apply rules within the gateway's network namespace. This capability is the fundamental requirement for implementing the protection mechanisms discussed above, where control must occur precisely at the threshold between the local network and the tunnel to the remote cluster.

### Implications for Computing Continuum and Multi-tenancy

Finally, from a cluster federation perspective, the issue of policy authority and ambiguity arises.

In a theoretical future scenario where Liqo automatically replicates NetworkPolicies between clusters (similar to what happens with Service and ConfigMap), a security risk would arise: a Consumer could alter its local NetworkPolicies and see them propagated to the Provider's cluster, effectively bypassing the security constraints imposed by the latter.

Furthermore, in the current scenario, ambiguities may still arise for the cluster administrator, who may find automatically created NetworkPolicies alongside the ones created by him.

The adoption of FirewallConfiguration inherently resolves this ambiguity. As a cluster-wide resource managed directly by Liqo's core components and not tied to individual namespaces, it clearly separates responsibilities. The Provider can define firewalling rules that are immutable from the Consumer's point of view, ensuring real and secure segregation in the multi-tenant model of the computing continuum.

**Summary Comparison Table**

| **Feature** | NetworkPolicy | FirewallConfiguration (Liqo) |
|---|---|---|
| **CNI Dependency** | High (behavior varies by CNI, e.g., Calico, Cilium) | Agnostic (based on Netfilter/Kernel) |
| **Abstraction Level** | High (Label and Selector based) | Low (Granular control for controllers) |
| **Rule Flexibility** | Limited (IP, Ports, Pods, Namespaces) | Maximum (Full nftables support, all hooks/interfaces) |
| **Encapsulated Traffic** | Not supported (cannot filter Geneve tunnels) | Supported (Direct management on peering interfaces) |
| **Multi-cluster Security** | Risk of manipulation by the Consumer cluster | Centralized/Cluster-wide; cannot be altered by the tenant |

**Table 3.1:** Comparison between standard NetworkPolicies and Liqo FirewallConfiguration.

## 3.3 Conclusion

The analysis of the use cases demonstrates that the default connectivity model of Kubernetes peering solutions is insufficient for secure multi-tenant operations. The

dichotomy of Provider and Consumer protection highlights the need for a flexible *Network Manager* that allows to create granular policies.

Initially, research focused exclusively on the use of FirewallConfigurations. This choice was dictated by the need to overcome the limited customization of standard NetworkPolicies, offering a framework dedicated exclusively to the peering system, which was less ambiguous and technically more versatile. However, in-depth technical analysis revealed structural criticalities related to the heterogeneity of modern CNIs (Container Network Interfaces).

The main obstacle is the increasingly widespread adoption of eBPF for data plane implementation. In the presence of CNIs that use TC (Traffic Control) or XDP (eXpress Data Path) hooks, traffic can completely bypass nftables chains. As a result, applying firewalling rules directly at the node level via FirewallConfigurations does not guarantee agnosticism with respect to the underlying CNI, risking compromising the integrity of the security posture.

To overcome these limitations, the strategy has been recalibrated towards a hybrid enforcement model, which leverages the strengths of both mechanisms:

- FirewallConfigurations on the Gateway: This component is used to regulate incoming traffic from remote clusters. Operating within an isolated network namespace separate from the direct scope of the network plugin, nftables rules maintain their effectiveness, ensuring robust perimeter control.
- NetworkPolicies for Intra-cluster Traffic: NetworkPolicies are the optimal solution for managing east-west flows and isolating internal workloads. Although there are implementation differences between the various CNIs, they offer sufficient standardized abstraction to meet the security requirements defined at this stage of the project, while ensuring compatibility with kernel offloading mechanisms.

# Chapter 4

# Implementation

This chapter details the logical architecture and design choices behind the proposed Network Manager. Its primary objective is to translate high-level, declarative security intents into concrete, low-level network primitive, specifically NetworkPolicy and FirewallConfiguration.

## 4.1 High level design

This chapter outlines the logical architecture of the proposed solution, analyzing the functional requirements and design choices underlying the definition of the networking model.

The primary objective of the system is to create an abstraction framework capable of interpreting high-level intents. These intents are designed to combine a simplified declarative syntax with granular control over security policies. The technological core of the project lies in the process of translating these intents into low-level resources: NetworkPolicy and/or FirewallConfiguration.

The architecture delegates the actual implementation of the rules to designated entities: in the case of FirewallConfiguration is Liqo's responsibility, while NetworkPolicy is interpreted and applied directly by the network plugin active in the cluster.

In the early stages of design, the exclusive use of FirewallConfiguration was considered. This choice was dictated by the goal of using a single component with predictable behavior that would not cause ambiguity, since FirewallConfigurations are only used for peering traffic management, while NetworkPolicies are intended for use by Kubernetes users. Furthermore, FirewallConfigurations are more flexible than NetworkPolicies, so thanks to their granular rules, they were able to cover all the cases covered by NetworkPolicies, making them the perfect candidate to be the only low-level tool to use. Figure 4.1 shows the high level flow for the first version of model.

However, experimental analysis highlighted structural limitations related to the heterogeneity of data planes in modern CNIs. Specifically, in configurations that adopt eBPF for packet forwarding, a phenomenon of bypassing standard Netfilter chains is observed. Since eBPF intercepts and routes traffic at a deeper level of the operating system (often directly at the network driver level or through specific kernel hooks), rules defined using traditional firewalling tools are ineffective.

This technological divergence has necessitated a paradigm shift towards a hybrid or adaptive model capable of discerning the underlying network backend to ensure the integrity of security policies. The chosen solution involves using NetworkPolicies to regulate intra-cluster traffic, while FirewallConfigurations on the gateway have been chosen to manage inter-cluster traffic. This is a strategic position, since the gateway's network namespace is not affected by the rules applied by the network plugins. Figure 4.2 shows the improved flow (V2) using both FirewallConfiguration and NetworkPolicy.

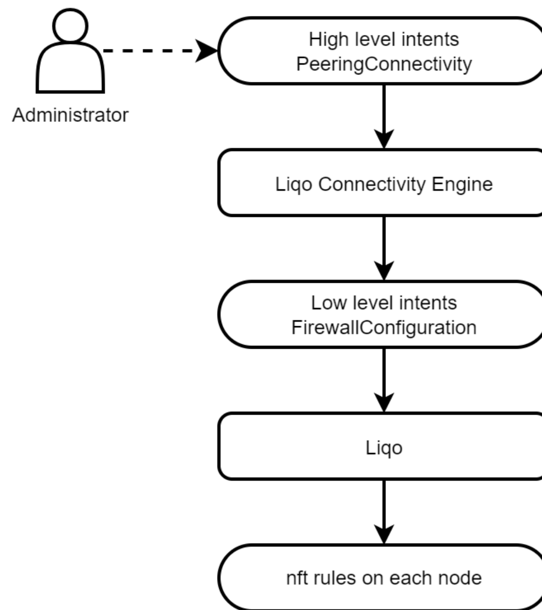


**Figure 4.1:** V1 model high level design

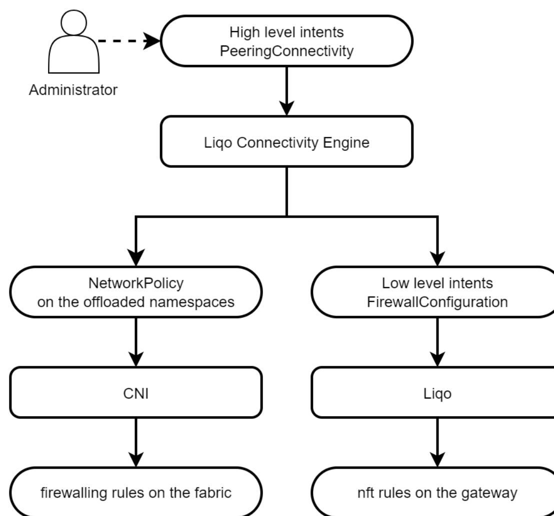


**Figure 4.2:** V2 model high level design

## 4.2 CRD

As illustrated in the architectural analysis in the previous chapters, the core of the Network Manager lies in its ability to translate high-level security intents into low-level network primitives and configurations (such as FirewallConfiguration and NetworkPolicy).

To make this conversion process operational, it is first necessary to rigorously define the structure and semantics of the high-level intents provided by the administrator. Within the declarative paradigm of Kubernetes, these intents are described and processed through the instance of a Custom Resource Definition (CRD). Therefore, this chapter is dedicated to designing a CRD suitable for this purpose, an interface modeled to abstract infrastructure complexities and precisely define the resource groups and traffic rules necessary to orchestrate security in the Computing Continuum.

### 4.2.1 Groups

To ensure the flexibility and scalability required to manage distributed infrastructures, the Network Manager cannot be limited to operating on individual static IP addresses. Instead, the administrator must be able to define dynamic pods grouping criteria that reflect the multi-tenant and hierarchical nature of the Computing Continuum. The CRD therefore introduces the concept of Groups, logical abstractions that allow resources to be categorized based on two fundamental variables: resource ownership and execution cluster (hosting).

The taxonomy of groups adopted is analyzed below, distinguishing between the Consumer and Provider perspectives.

#### The Consumer Perspective: Segmentation and “Cluster Slicing”

From the perspective of the Consumer cluster (the one outsourcing the workload), the primary need is to maintain a clear distinction between what resides within the local perimeter and what is projected outward.

- Grouping by Cluster Location: The system allows the administrator to define `local-cluster` and `remote-cluster` groups. This distinction is not merely nominal, but is based on Liqo’s dynamic CIDR resolution. By analyzing Liqo’s `Network` resource, the controller is able to extract both the original and remapped CIDRs (necessary to manage any IP address overlaps between federated clusters). This allows the generation of precise matching rules that automatically adapt to changes in the network topology.
- Offloaded Namespaces and Cluster Slices: One of the most innovative features is the management of offloaded namespaces. Since a single namespace can

logically extend across multiple clusters, the Consumer has complete visibility to perform micro-segmentation defined as "Cluster Slicing":

- `cluster-slice-local`: includes Pods physically residing on the local cluster
- `cluster-slice-remote`: identifies workloads running on various Providers

Technically, this distinction is made possible by integration with Liqo's CRD `NamespaceOffloading` and analysis of Pod metadata. Specifically, the controller identifies remote Pods through the `liqo.io/shadowPod` label, whose value `true` indicates that the resource is a shadow running on a remote infrastructure.

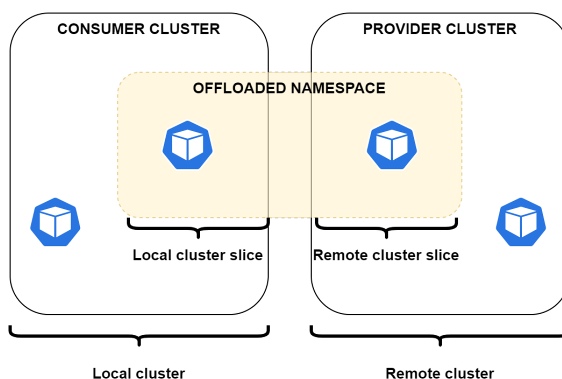


**Figure 4.3:** Resource groups from the consumer perspective

### The Provider's perspective: Tenant Isolation and Blind Filtering

The role of the Provider is intrinsically different: it must offer resources while ensuring the isolation of its core from potentially untrusted external entities.

Unlike the Consumer, the Provider operates in a condition of reduced visibility: it has no knowledge of the internal structure (original namespace or label) of the cluster from which the traffic originates. Therefore, filtering cannot be based on logical abstractions of the Consumer, but must be based on physical criteria and adherence to peering tunnels.

The only granular abstraction available to the Provider is the `offloaded` group, which aggregates all Pods belonging to a specific Consumer that are currently hosted on its resources. Inbound traffic filtering is then managed by cross-referencing the source CIDRs (obtained through the Network resource instances) and the identity of the originating tunnel.

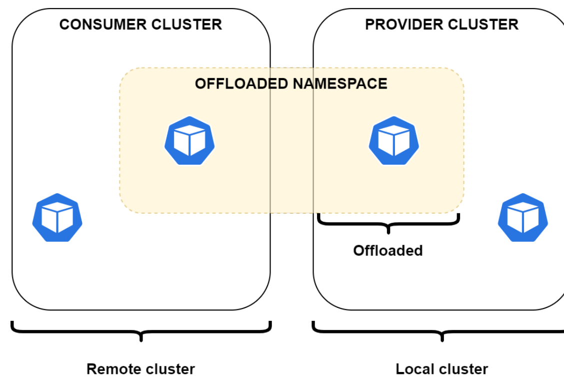


**Figure 4.4:** Resource groups from the provider perspective

### External CIDR

Within Liqo's network architecture, in addition to the pod CIDR, each participating cluster is assigned a specific block of IP addresses, defined as `external` CIDR. This dedicated subnet plays a crucial role in managing and redirecting service traffic in leaf-to-leaf communication scenarios. Specifically, in multi-provider environments, if a pod hosted on a given provider's infrastructure needs to transmit data to a pod located at a different provider, this CIDR allows traffic to be routed transparently through the consumer cluster, which acts as a central transit node. Furthermore, in the context of dynamic resource extension, it is of fundamental importance for a provider to ensure that pods undergoing offloading maintain smooth and uninterrupted connectivity to the external network of their home cluster.

In order to meet this architectural requirement and govern these communication flows in a granular manner, a logical construct called a `leaf` group has been introduced, designed specifically to define and implement this specific category of network rules.

### Internet traffic

Another critical aspect of network policy management pertains to the routing of egress traffic destined for the public internet. Specifically, it is essential to establish granular control over data packets directed towards non-private subnets, as well as DNS resolution requests targeting external nameservers. This level of governance is strictly required to explicitly allow or restrict the full outbound connectivity of the pods, depending on the underlying security, compliance, and operational requirements of the deployment.

To address these networking needs, the architecture introduces dedicated logical abstractions, namely the `internet` and `nameserver` groups. These constructs are explicitly designed to facilitate the formulation and automated enforcement of policies governing reliable access to external infrastructure.

## 4.2.2 Rules definition

The formalization of a connectivity rule within the system requires the specification of three fundamental parameters, defined as the "decision triplet": action, source, and destination.

### Action Logic and Security Paradigm

Considering the objectives of this first iteration of the controller—focused exclusively on traffic filtering (firewalling) functionality—the supported actions logically concern the authorization or inhibition of traffic. However, we have chosen to

adopt a default-deny paradigm natively. This approach is dictated both by Zero Trust security best practices and by the need to overcome the structural limitations of standard Kubernetes NetworkPolicies, which are often complex to manage in explicit denial configurations. Consequently, within the CRD, the only explicitly configurable action for this version of the controller is `allow`; the absence of a permissive rule inherently implies traffic blocking.

### Source and Destination Flexibility

To ensure maximum granularity in access control, the source and destination fields have been designed as optional. This architectural choice allows the user to define asymmetric rules (for example, filtered exclusively on the basis of the source, regardless of the destination endpoint, or vice versa).

The entities involved in a rule can be referenced through previously defined logical groups. However, in order to increase the flexibility of the solution, the CRD must be designed to be extended and support filters based on different types of metadata.

In addition to group-based abstraction, this version of the controller introduces support for Namespace filters. While it is not the priority, this feature is crucial in multi-tenant scenarios or complex infrastructures, where it is necessary to implement whitelisting strategies for shared resources located in isolated segments of the cluster.

### Example

Below is a representative example of the spec of a CRD instance on a Provider, illustrating the syntax and organization of the fields described above:

```yaml
spec:
  rules:
    # Rule 1: Allow traffic from remote cluster to offloaded pods
    - source:
        group: remote-cluster
      destination:
        group: offloaded
      action: "allow"

    # Rule 2: Allow traffic from offloaded pods back to remote cluster
    - source:
        group: offloaded
```

```
13        destination:
14          group: remote-cluster
15        action: "allow"
16
17    # Rule 3: Allow traffic from offloaded
  namespaces to other offloaded namespaces
18    - source:
19          group: offloaded
20        destination:
21          group: offloaded
22        action: "allow"
```

## 4.3 Extending FirewallConfiguration

The current implementation of `FirewallConfiguration` in Liqo provides a basic solution for managing firewalling rules in multi-cluster Kubernetes environments. In the context of the Network Manager project, a significant extension of these capabilities was necessary to meet advanced security and network traffic control requirements. In particular, the existing architecture had substantial limitations in two critical areas: support for connection tracking and management of named sets for filtering rule optimization.

This chapter analyzes two pull requests (PRs) submitted in the official Liqo repository that address these limitations by introducing advanced features for nftables-based firewalling.

The two Pull Requests represent a significant evolution of Liqo's firewalling capabilities. The introduction of connection tracking state matching and named sets transforms `FirewallConfiguration` from a basic solution to an advanced traffic control system, comparable to the features offered by enterprise solutions.

These extensions are particularly relevant in the context of multi-cluster environments, where the complexity of the network topology and security requirements demand granular and high-performance control tools. The adoption of `nftables` as a backend, combined with the high-level abstractions offered by Kubernetes Custom Resources, provides an optimal balance between flexibility and ease of use.

The integration of these features into the Network Manager is an important step towards the realization of secure, dynamic, and scalable multi-cluster topologies, the primary goal of the Liqo project.

### 4.3.1 Support for Connection Tracking State

Connection tracking is a fundamental component of modern firewalling systems, allowing the distinction between packets belonging to established connections and new connections. This update introduces support for matching on connection tracking states (`ctstate`) in firewall rules, filling a critical gap in the existing implementation.

The PR[10] introduces the `ctstate` field as a valid matching criterion within filtering rules. The supported states are the ones defined by nftables:

- `new`: The packet is initiating a connection (e.g., a SYN packet)
- `established`: Connections already established
- `related`: Connections related to existing connections
- `invalid`: Packets that don't follow the state machine (often dropped for security).

- `untracked`: Packets not tracked by the connection tracking system

The behaviour of the filter changes based on the chosen operation:

- `eq`: checks whether the packet state matches one of the specified values
- `neq`: checks whether the packet state does not match any of the specified values

The following YAML configuration illustrates the use of `ctstate` matching to implement an asymmetric firewalling rule:

```yaml
spec:
  table:
    family: IPV4
    name: test-table
    chains:
      - hook: postrouting
        name: test-chain
        policy: accept
        priority: 99
        type: filter
        rules:
          filterRules:
            - action: accept
              match:
                - ctstate:
                    value:
                      - established
                      - related
                  op: eq
```

The rule defined in FirewallConfiguration must be converted to an nftable rule. Rules are defined with low-level expressions as in the following example. `stateBits` are the status bits obtained from the conversion of the rules (e.g., established, related...) and `cmpOp` is the `eq` or `neq` operation.

```go
rule.Exprs = append(rule.Exprs,
    &expr.Ct{
        Register:       1,
        SourceRegister: false,
        Key:            expr.CtKeySTATE,
    },
```

```
7       &expr.Bitwise{
8           SourceRegister: 1,
9           DestRegister:   1,
10          Len:            4,
11          Mask:           binaryutil.NativeEndian.PutUint32(
    stateBits),
12          Xor:            []byte{0x0, 0x0, 0x0, 0x0},
13      },
14      &expr.Cmp{
15          Op:       cmpOp,
16          Register: 1,
17          Data:     []byte{0, 0, 0, 0},
18      },
19  )
```

The functionality was validated through a real-world scenario in which it is necessary to block incoming connections from a specific subnet, while maintaining the ability to make outgoing connections to it. Without connection tracking support, a blocking rule would also prevent responses from locally initiated connections from returning. The introduction of `ctstate` matching allows us to distinguish between new connections (to be blocked) and packets belonging to already established connections (to be accepted), thus achieving directional traffic control.

### 4.3.2 Named Sets management

The second PR[11] addresses a different but complementary issue: the efficient management of firewalling rules involving extensive lists of IP addresses or other matching criteria. The use of named sets in nftables significantly improves the readability, maintainability, and performance of firewall configurations, particularly when managing dynamic lists of items.

To support this capability, the `FirewallConfiguration` Custom Resource has been augmented with a novel `sets` field, enabling the declarative definition of named sets. Each defined set encapsulates the following properties:

- Name: A unique identifier for the set within the given scope
- Key type: The structural data type of the set's keys (presently supporting `integer`, `ipv4_addr`, and `ipv4_cidr`)
- Data type: An optional parameter defining the type of the associated data values, mirroring the permissible key types
- Elements: The explicit enumeration of values to be incorporated into the set.

Concurrently, the `FirewallConfiguration` controller's reconciliation loop has been expanded to autonomously govern the complete lifecycle of these named sets. This lifecycle management encompasses:

- Creation: The provisioning and instantiation of novel sets directly within the underlying nftables subsystem
- Update: The dynamic mutation of elements within pre-existing sets to reflect declarative state changes
- Removal: The systemic deletion of obsolete sets that are no longer referenced by active policies

From a usability perspective, users can reference these named sets within matching rules utilizing the `@<set_name>` syntax. This declarative notation strictly adheres to established nftables conventions, thereby minimizing syntactic ambiguity during rule generation within the Liqo ecosystem.

Furthermore, the validating admission webhook has been fortified to guarantee structural and semantic integrity. The webhook strictly enforces:

- The validity and uniqueness of the defined sets
- The referential integrity of the sets invoked within firewall rules (preventing dangling references)
- Strict adherence to data type constraints during element assignment

The following declarative manifest illustrates the definition and subsequent application of a named set within a `FirewallConfiguration`:

```yaml
spec:
  table:
    name: test_table
    family: IPV4
    # New sets field to define named sets
    sets:
      - name: test_set
        keyType: ipv4_addr
        elements:
          - key: "8.8.8.4"
          - key: "8.8.8.8"
    chains:
      - hook: postrouting
        name: test_chain
```

```
        policy: accept
        priority: 99
        type: filter
        rules:
          filterRules:
            - action: drop
              match:
                - ip:
                    # Referencing the named set using @<set_name>
                    value: "@test_set"
                    position: "dst"
                  op: "eq"
```

The architectural integration of named sets yields several definitive operational advantages:

- Scalability: Facilitates the highly efficient management of target lists comprising thousands of elements without the linear performance degradation associated with traditional rule evaluation
- Maintainability: Enables the centralized mutation of referenced elements (addition or removal) without necessitating the modification of the underlying rule definitions
- Performance: Leverages the internal search optimizations intrinsic to nftables (utilizing advanced data structures like hash tables) for accelerated packet matching
- Reusability: Allows a singularly defined set to be invoked concurrently across multiple distinct firewall rules, reducing configuration redundancy

## 4.4 Rules translation

A cornerstone of the proposed controller's architecture is its translation engine, a critical component tasked with converting high-level, declarative security intents expressed via the *PeeringConnectivity* Custom Resource Definition (CRD) into concrete, actionable instances of *FirewallConfiguration* and *NetworkPolicy*. This bridging of the semantic gap requires a sophisticated parsing mechanism capable of performing deep semantic analysis of traffic sources and destinations, thereby correctly mapping user-defined abstractions onto the underlying network primitives supported by the orchestrator.

The translation parser must dynamically process the source and destination fields, resolving the entities into one of two primary architectural constructs:

1. **Namespace**: Standard Kubernetes logical partitions
2. **Groups**: Custom logical abstractions conceptualized at the codebase level to aggregate heterogeneous cluster entities, requiring specialized programmatic resolution logic to generate valid network matching rules.

### 4.4.1 Generation Logic

#### FirewallConfiguration

When synthesizing low-level rules for FirewallConfiguration, the engine must translate abstract groupings into tangible IP-based constructs.

For Kubernetes Namespaces, this derivation is deterministic: the translation engine dynamically queries the Kubernetes API to extract the specific set of IP addresses belonging to the Pods actively running within that namespace, binding them to a match rule.

Conversely, to accommodate the dynamic flexibility demanded by custom logical Groups, the architecture should implement a modular pattern utilizing specialized functions that yield *FirewallConfiguration's MatchRules* data structures for each group defined in code.

#### NetworkPolicy

The translation of intents into Kubernetes NetworkPolicy resources operates at a higher tier of the infrastructure abstraction. NetworkPolicies natively enable the formulation of elevated rules that are inherently simpler to define and maintain. Because NetworkPolicy enforcement is executed by the cluster's network plugin, many of the complex pod-matching operations such as parsing label selectors that are explicitly required for FirewallConfiguration sets, are securely delegated directly to the CNI's data plane.

Generating a match rule for a Namespace is trivialized in this context, as the controller can leverage the native `NamespaceSelector` primitive provided by the Kubernetes API.

On the other hand, custom groups should remain managed by specialized functions that programmatically return the necessary NetworkPolicy match specifications.

### Group interface

In light of the aforementioned architectural considerations, each abstract group should be programmatically mapped to a specific functional interface defined by the *groupFuncts* struct, which encapsulates the entire lifecycle of its resolution logic:

```
type groupFuncts struct {
     MakeFirewallConfigurationRule func(ctx context.Context, cl client.Client, clusterID string, position networkingv1beta1firewall.MatchPosition) ([]networkingv1beta1firewall.Match, error)
    MakeFirewallConfigurationSets func(ctx context.Context, cl client.Client, clusterID string) ([]networkingv1beta1firewall.Set, error)
    MakeNetworkPolicyRule         func(ctx context.Context, cl client.Client, clusterID string) ([]networkingv1.NetworkPolicyPeer, []networkingv1.NetworkPolicyPort, error)
}
```

The generation of rules for the `FirewallConfiguration` resource necessitates the implementation of two distinct functions: `MakeFirewallConfigurationRule` and `MakeFirewallConfigurationSets`. The former is strictly responsible for computing and returning the core matching rule to be appended to the security policy. The latter serves as an optional auxiliary mechanism, explicitly designed to yield any specific data structures or sets required to properly integrate and validate the rule within the broader FirewallConfiguration instance.

Conversely, the instantiation and management of `NetworkPolicies` involve a significantly more streamlined architectural approach, primarily due to their inherent structural simplicity. Within this context, only the `MakeNetworkPolicyRule` function is strictly required. This component is exclusively tasked with generating the appropriate matching rules, which are subsequently attached to either the ingress or egress directional traffic specifications.

### 4.4.2 Groups conversion

Each group requires a different logic for conversion into low-level rules:

- local-cluster: Match on the CIDR of the pods in the current cluster, which can be obtained by reading the status of Liqo's Network resource called "pod-cidr"
- remote-cluster: Match on the CIDR of the remote cluster pods, obtainable by reading the status of Liqo's Network resource called "<cluster_id>-pod"
- leaf: Match on the remote cluster's external CIDR, obtainable by reading the status of Liqo's Network resource called "<cluster_id>-external"
- offloaded: Matches on Pods with the label "offloading.liqo.io/origin" equal to the peering cluster ID.
- slice-local: To convert this group, the controller has to obtain the list of offloaded namespaces by reading the NamespaceOffloadingList resource list and find all pods in these namespaces without the "liqo.io/shadowPod" label.
- slice-remote: The list of pods with "liqo.io/shadowPod".
- internet: Match on all IP addresses not included in the RFC1918 subnets (10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16).
- nameserver: Matches for direct connections to port 53.

For both FirewallConfiguration and NetworkPolicy, it is possible to create match rules on subnets, so the logic is similar and does not require complex data structures.

When a rule relies on the use of labels, for FirewallConfiguration it is necessary to obtain the list of Pods that meet the requirement and create a set with their IP addresses, then creating a rule based on the match on addresses contained in that set. NetworkPolicies, on the other hand, have native support for label comparisons, making it very easy to write the rule.

### 4.4.3 Conversion Flow for FirewallConfiguration

A naive, sequential approach to translating the FirewallConfiguration specifications would process each declarative rule in isolation. This sub-optimal algorithm would entail:

- Resolving IP addresses and generating nftables sets for the source entity.
- Synthesizing the corresponding source match rule.
- Resolving IP addresses and generating nftables sets for the destination entity.

- Synthesizing the corresponding destination match rule.

However, this linear evaluation is highly inefficient. In complex, multi-tenant cluster topologies, different security rules frequently reference the same logical groups, leading to severe computational redundancy and bloat in the resulting nftables configuration.

To optimize controller performance and minimize the payload size of the generated configurations, the synthesis algorithm employs a pre-computation strategy. The engine parses the entire set of rules to identify and isolate unique groups prior to generating the low-level configurations. By consolidating the list of all unique entities involved, the costly process of IP resolution and set creation is executed strictly once per group, yielding a highly streamlined, performant, and scalable architecture. Figure 4.5 shows this algorithm from a high-level point of view.

### 4.4.4 Conversion Flow for NetworkPolicy

A critical phase of this generation involves filtering the broader PeeringConnectivity rule set to isolate and process exclusively those rules that pertain to the offloaded group. Once isolated, the engine determines the traffic directionality by evaluating the topological position of the offloaded group within the rule parameters (i.e., acting as the source or the destination). Based on this contextual analysis, the translated rule is accurately injected into either the Ingress or Egress policy specification.

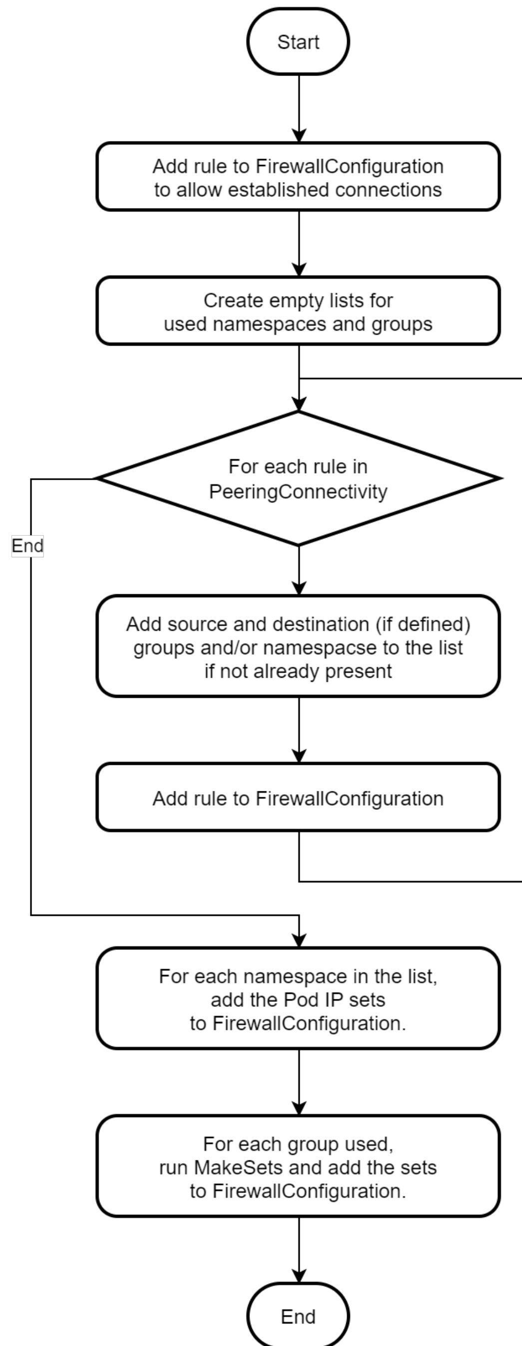


**Figure 4.5:** FirewallConfiguration reconciliation algorithm

## 4.5 Controller

This chapter aims to analyze in detail the architecture and operation of a reconciler designed specifically to orchestrate connectivity between geographically or logically separated Kubernetes clusters. The reconciler is the central component of a connectivity management system that implements the reconciliation pattern typical of Kubernetes operators, ensuring that the desired state of the network configuration is constantly aligned with the actual state of the underlying infrastructure.

The adopted approach is declarative and it allows the system to achieve high levels of resilience and self-healing: if a manual change or a network failure alters the infrastructure, the reconciler automatically detects the deviation during its next execution, re-applying the intended configuration without requiring external intervention. This mechanism is crucial when dealing with the cloud where imperative configuration management can simplify scalability and error management.

### 4.5.1 The reconciliation loop

At the foundation of Kubernetes' declarative orchestration is the reconciliation loop. The Liqo Network Manager[12] leverages this continuous control mechanism to ensure that the actual network security posture of federated clusters consistently matches the desired state defined by administrators via the PeeringConnectivity Custom Resource (CR). Unlike imperative models that execute sequential commands, this loop proactively monitors the system, continuously correcting any configuration drift between the intended architecture and the deployed infrastructure.

Triggered by the creation, modification, or deletion of a PeeringConnectivity instance, the reconciliation loop executes a strict, idempotent sequence of operations:

1. Resource Retrieval and Validation: The controller queries the Kubernetes API server for the latest PeeringConnectivity specification, validating the structural integrity of the requested rules (e.g., verifying action, source, and destination parameters) against predefined schemas.

2. Finalizer Management: If the resource is marked for deletion, the controller initiates cleanup routines. It utilizes Kubernetes finalizers to guarantee that all subordinate low-level constructs—such as FirewallConfiguration or NetworkPolicy manifests—are safely dismantled before the parent CR is permanently removed from etcd.

3. Intent Translation: For active resources, the controller routes the high-level security intents through a translation engine. This engine resolves logical abstractions (e.g., slice-local or offloaded groups) into concrete network parameters like IP addresses, CIDRs, and Pod label selectors.

4. State Alignment and Enforcement: The controller synthesizes the desired low-level configuration and performs a semantic comparison against the cluster's actual state. It then issues precise API calls (Create, Update, or Patch) to correct any detected discrepancies. This process is strictly idempotent, ensuring that repeated executions yield the same state without causing redundant network disruptions.

5. Status Update: To close the loop, the controller updates the Status subresource of the PeeringConnectivity CR. This grants the administrator immediate, observable feedback regarding the operational state and successful enforcement of the network policies.

Through this continuous, self-healing cycle, the Network Manager dynamically adapts to the ephemeral nature of workloads within the Computing Continuum, guaranteeing that multi-tenant isolation boundaries remain robust and consistent.

### 4.5.2 Triggers

Workloads in Kubernetes are inherently ephemeral: Pods are frequently rescheduled, IP addresses are dynamically allocated, and network topologies seamlessly shift as new clusters are peered and resources are offloaded. Consequently, the controller must be acutely aware of state changes occurring across various native and custom cluster entities to ensure that the generated FirewallConfiguration and NetworkPolicy rules remain strictly consistent with the high-level intents.

To achieve this real-time synchronization, the Liqo Network Manager leverages an event-driven architecture: the controller subscribes to infrastructure state changes (Create, Update, Delete events) without overwhelming the Kubernetes API server with inefficient, continuous polling.

The primary watch is established directly on the PeeringConnectivity CRD. Any administrative mutation to the declarative security intents immediately enqueues a reconciliation request. However, because the translation engine's logical abstractions depend on the real-time status of the cluster, the controller must also establish secondary watches on the following critical resources:

- Pods: The engine must track Pod lifecycles to dynamically extract IP addresses, which are necessary for populating the named sets within the FirewallConfiguration.

- offloading.liqo.io/NamespaceOffloading: Monitoring offloaded Namespaces to maintain a real-time registry of which specific namespaces are actively extended across federated clusters, useful when defining rules that apply to entire local or remote cluster slices.

- networking.liqo.io/Network: Tracking local, remote, and external CIDRs (such as pod-cidr and <cluster_id>-external)

In order to properly process Kubernetes watch events, it is necessary to implement handler functions capable of extracting the identifier of the target primary resource to be reconciled from the updated state of the monitored resources:

- podEnqueuer: Initiates the reconciliation process upon detecting modifications to relevant Pod resources. It maps the event to the Pod's owning cluster, deliberately discarding events for Pods that lack a multi-cluster owner reference
- networkPolicyEnqueuer: Monitors NetworkPolicy resources for state transitions, similarly enqueueing a reconciliation request for the corresponding owner cluster
- allPeeringConnectivityEnqueuer: Intercepts modifications to NamespaceOffloading resources and triggers a global reconciliation for all PeeringConnectivity objects, as offloading operations systematically impact all connected providers

### 4.5.3 Systemic Resilience

To maintain high performance and reliability within the dynamic environment of the Cloud, the controller's architecture must anticipate and gracefully mitigate systemic anomalies, transient failures, and concurrent state mutations. This resilience is achieved through a deliberate combination of strictly idempotent execution, asynchronous event queuing, rate limiting, and optimistic concurrency control.

#### Idempotent State Synchronization

A foundational requirement for the Liqo Network Manager is the absolute guarantee of idempotency. In distributed systems, this ensures that redundant or overlapping executions of the reconciliation loop consistently yield the exact same infrastructure state without inducing unintended side effects. Rather than issuing imperative commands to manipulate network rules, the controller computes the absolute desired state entirely in memory. It then executes a rigorous synchronization phase: fetching the current live state from the cluster's datastore (etcd) and performing a deep semantic comparison against the synthesized desired state. Based on this comparison, the controller issues highly targeted API operations—creating missing resources, or updating and patching those with identified discrepancies.

Crucially, if the live state already matches the desired configuration perfectly, the controller intentionally acts as a no-op. This behavior is vital for data plane

stability; by systematically avoiding redundant updates, the controller prevents the underlying Container Network Interface (CNI) and the Linux kernel's Netfilter framework from needlessly recompiling and reloading rulesets. Consequently, the data plane is shielded from unnecessary computational overhead and transient packet drops.

#### Asynchronous Processing via Workqueues

Because workloads in Kubernetes are inherently ephemeral and can generate sudden bursts of activity, synchronous event processing would risk stalling the controller's execution threads. To decouple event detection from actual state reconciliation, the engine implements an asynchronous workqueue mechanism. When a state change is detected—such as a Pod offloading or a policy mutation—the handler does not immediately reconcile the cluster. Instead, it enqueues a lightweight reconciliation request containing only the namespace and name of the target resource. This architecture allows the controller to absorb massive spikes in cluster activity seamlessly, processing requests at a sustainable pace.

#### Optimistic Concurrency Control (OCC)

Finally, the controller must safely navigate concurrent modifications to shared network resources. Kubernetes inherently prevents lost updates through Optimistic Concurrency Control (OCC), which tracks resource mutations via a ResourceVersion field. If the Network Manager attempts to update a low-level manifest (such as a NetworkPolicy) that has been modified by another cluster process since it was last fetched, the API server rejects the transaction with a Conflict error (HTTP 409). The controller handles these conflicts gracefully: the reconciliation loop deliberately aborts the stale operation and returns an error to the workqueue. Prompted by the rate limiter, the subsequent retry fetches the absolute latest resource state, recomputes the necessary adjustments, and safely reapplies the patch. This OCC mechanism guarantees robust data consistency across the federated infrastructure without the need for complex, manual locking protocols.

# Chapter 5

# Tests

Following the implementation phase, this chapter describes the testing campaign conducted to validate the correct functioning of the developed Kubernetes controller.

The evaluation phase has two objectives: first, to demonstrate the stability and reliability of the system by ensuring its interoperability when the Container Network Interface (CNI) adopted within the clusters changes. Second, to provide a comparative analysis between the final solution implemented and the initial theoretical model, based exclusively on the use of the FirewallConfiguration resource.

## 5.1 Experimental Methodology and Infrastructure Automation

Validating a cross-cluster Kubernetes controller, such as the Liqo Network Manager, requires a dynamic testing framework. Because Liqo's core utility is bridging administrative domains to form a computing continuum, testing must encompass varied network topologies and Container Network Interface (CNI) configurations to prove architectural resilience. Relying on manually provisioned clusters is inadequate for rigorous scientific inquiry, as it introduces human error and severely limits reproducibility.

To address these limitations, an automated Infrastructure-as-Code (IaC) framework[13] was developed to manage the lifecycle of ephemeral, multi-node environments using k3d (a lightweight k3s wrapper operating within Docker). By simulating heterogeneous conditions, the framework ensures the controller is rigorously evaluated against the complexities of real-world, cloud-native deployments. This methodological approach guarantees that Liqo's connectivity abstractions remain invariant regardless of the underlying cluster environment.

CNI plugin behavior is a critical variable in evaluating cross-cluster peering. Different implementations handle network policies and packet encapsulation in

fundamentally distinct ways, directly impacting both performance and security. Consequently, the automated provisioning tool dynamically injects and configures several industry-standard CNIs. The evaluation will be executed against the following configurations:

- Flannel (paired with Calico for Network Policy enforcement)
- Calico (operating in both nftables and eBPF modes)
- Cilium

The test environment instantiation follows a strict, automated six-step sequence:

1. Cluster provisioning: k3d clusters are created
2. Network plugin installation: the CNI chosen for the individual test is installed on each cluster
3. Liqo installation on each cluster
4. Cluster peering and namespace offloading
5. Resource generation (Pods, Services, and PeeringConnectivity)
6. Controller execution to generate FirewallConfigurations and NetworkPolicies based on the PeeringConnectivity resources

## 5.2 Validation Methodology: The Connectivity Verification Framework

Shifting the Liqo ecosystem from a permissive peering model to a zero-trust architecture requires empirical validation of the newly established security boundaries. The Liqo Network Manager acts as the enforcement layer, and to verify its efficacy, an automated Python-based testing framework was developed. This tool systematically audits the cross-cluster network topology to produce a comprehensive connectivity matrix, confirming that defined policies are both functional and strictly non-permissive where intended.

The framework[13] utilizes a declarative infrastructure discovery model. Users supply a YAML configuration detailing the target clusters, corresponding kubeconfig credentials, and specific namespaces designated for analysis. Upon initialization, the suite dynamically discovers all active Pod IP addresses and Service ClusterIPs within these scopes.

Crucially, the tool integrates natively with Liqo's offloading logic. In standard multi-cluster setups, a Pod offloaded from a local to a remote peer might be erroneously identified as two separate entities. The framework intelligently reconciles these identities by recognizing the resource's offloaded status, thereby preventing redundant entries in the connectivity matrix. Although currently optimized for the Liqo fabric, the architecture is highly modular, supporting future adaptations for generic multi-cluster environments independent of the Liqo control plane.

To accurately map the data plane's security posture, the verification tool implements two distinct probing mechanisms:

- ICMP Echo Requests (Ping): Verifies Layer 3 reachability and underlying network path availability.
- Application Layer Probes (Curl): Validates Layer 7 accessibility, ensuring security policies correctly manage stateful connections and protocol-specific traffic rather than merely dropping packets at the edge.

Additionally, the framework assesses perimeter security and core infrastructure accessibility across two critical vectors:

- Global Internet Egress: Evaluates connectivity to external, stable endpoints (e.g., 8.8.8.8) to confirm that the engine enforces egress policies, preventing unauthorized data exfiltration or access to public resources.
- Domain Name System (DNS) Resolution: Executes lookups (e.g., nslookup google.com) to ensure recursive DNS resolvers remain operational under the applied security constraints.

## 5.3 Test Suite

The test suite evaluates both functionality and boundary enforcement. To ensure comprehensive coverage, the following scenarios are systematically executed:

- Intra-Cluster Pod-to-Pod (ICMP/HTTP): Validates the consistency of local network policies.
- Cross-Cluster Pod-to-Pod (ICMP/HTTP): Verifies the boundary enforcement capabilities of the Liqo Engine.
- Pod-to-Service (HTTP): Audits Virtual IP (VIP) routing and associated security rules.
- Egress Connectivity (ICMP): Evaluates perimeter exit points.

- DNS Resolution (UDP/53): Confirms the operational status of service discovery.

The total number of tests performed, denoted as $T$, is the sum of the tests evaluating intra- and inter-cluster pod-to-pod connectivity ($T_{ptp}$), pod-to-service connectivity ($T_{srv}$), and egress and domain resolution for each pod ($T_{ext}$).

$$T = T_{ptp} + T_{srv} + T_{ext} \tag{5.1}$$

The individual components of this total depend on the overall cluster architecture. Let $c$ be the total number of clusters and $n_p$ be the total number of pods across all clusters. For any given cluster $i$, let $n_{pi}$ represent its number of pods and $n_{si}$ represent its number of services. The testing subsets are calculated as follows:

- Pod-to-Pod Connectivity:

$$T_{ptp} = 2n_p(n_p - 1) \tag{5.2}$$

- Pod-to-Service Connectivity:

$$T_{srv} = \sum_{i=1}^{c} n_{pi} n_{si} \tag{5.3}$$

- Egress and Domain Resolution:

$$T_{ext} = 2n_p \tag{5.4}$$

In this formulation, $2n_p(n_p - 1)$ represents the bidirectional Pod-to-Pod test matrix for both ICMP and HTTP probes. The term $\sum_{i=1}^{c} n_{pi} n_{si}$ accounts for all Pod-to-Service connection attempts within each cluster. Finally, $2n_p$ encompasses the egress and DNS resolution checks for every individual pod. While the framework utilizes a strict 1-second timeout per probe to efficiently handle dropped packets—an essential mechanism when testing "deny" rules—successful connections typically resolve in under 10ms, enabling high-throughput security validation.

### 5.3.1 Policy resources

The use case to be tested concerns the set of practices designed to protect consumers and providers, and is the combination of the scenarios described in the chapter on problem analysis.

For the consumer cluster, a single rule is enough: it permits all incoming traffic originating from pods offloaded to the provider:

```
spec:
  rules:
    - source:
        group: slice-remote
      action: "allow"
```

Conversely, the provider cluster requires a more comprehensive ruleset. These configurations must permit connectivity between the consumer and its offloaded pods, enable leaf-to-leaf communication, and grant internet access and DNS resolution for the offloaded workloads:

```
spec:
  rules:
    - source:
        group: remote-cluster
      destination:
        group: offloaded
      action: "allow"

    - source:
        group: offloaded
      destination:
        group: remote-cluster
      action: "allow"

    - source:
        group: offloaded
      destination:
        group: offloaded
      action: "allow"

    - source:
        group: leaf
      destination:
        group: offloaded
      action: "allow"
    - source:
        group: offloaded
      destination:
        group: leaf
```

```
30        action: "allow"
31
32      - source:
33          group: offloaded
34        destination:
35          group: internet
36        action: "allow"
37      - source:
38          group: offloaded
39        destination:
40          group: nameserver
41        action: "allow"
```

## 5.4 Results

This section details the functional outcomes of the conducted tests, categorized into three core operational scenarios: single peering, multiple providers, and multiple consumers.

The empirical data for Flannel and Calico (encompassing both nftables and eBPF data planes) are reported collectively, as both CNI plugins demonstrated functionally equivalent behavior. In contrast, the evaluation of Cilium is addressed in a dedicated subsection to examine the distinct outcomes and architectural limitations discovered during its NetworkPolicy enforcement.

To conclude, these findings are cross-examined with the initially proposed model, which utilized solely FirewallConfiguration. This comparison serves to highlight the fundamental behavioral disparities between environments driven by traditional nftables versus those leveraging eBPF.

### 5.4.1 Single peering

The first test concerns a single peering scenario: a Consumer and a Provider. In this scenario, the objective is to check not only the interactions between the two clusters, but also what happens between pods of the same namespace on the same cluster, in order to verify that the rules do not interfere with normal connections.

#### Infrastructure description

To test this scenario, an infrastructure was set up with only two clusters: a Consumer and a Provider. Both clusters have a local namespace that does not need

to be shared with the other cluster, and two pods were created for each of these namespaces. In addition, the Consumer offloaded a namespace called Offloaded that contains four pods: two hosted on the consumer and two on the provider. Figure 5.1 shows the high-level diagram of the infrastructure.

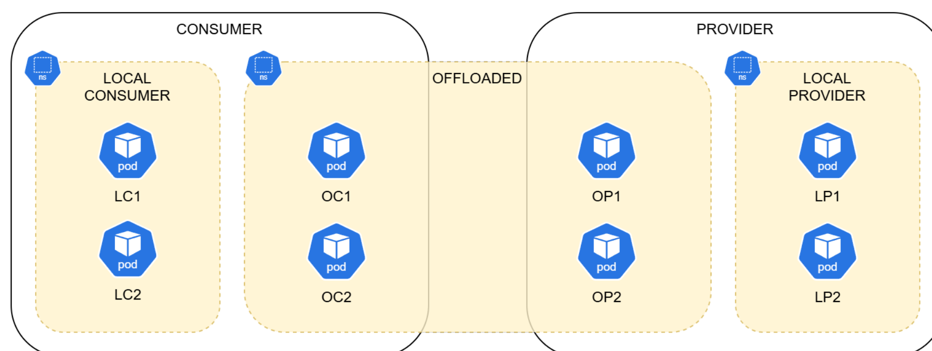


**Figure 5.1:** Single peering test environment

The pod's name format is composed of the following parts:

1. L if the namespace is local and not offloaded, O if the namespace has been offloaded
2. C if the Pod is hosted on the consumer, P if is hosted on the Provider
3. a number to identify the multiple pods in the same namespace and cluster

Example: LC1 is a pod in a private namespace on the Consumer's cluster. OP1 is a Pod created by the Consumer in an offloaded namespace that is hosted on the Provider.

### Results

In the test results, we can see that the connection between resources on the same cluster and belonging to the same owner is not blocked.

Furthermore, both as a direct connection and through services, the Consumer's pods are able to connect to their offloaded pods and vice versa, while they are prevented from accessing the Provider's services, while maintaining internet access and domain resolution capabilities.

Conversely, the Provider's pods have no way of contacting the Consumer's pods, neither those offloaded to their own cluster nor those in the Consumer's network.

Table 5.1 shows the results of Pod-to-Pod connection tests for both HTTP and ICMP requests as well as the connection tests towards the public internet,

demonstrating that connectivity and domain resolution remain allowed. Tables 5.2 and 5.3 show the results of HTTP calls to services available on the calling pod clusters, with 5.2 for the consumer and 5.3 for the provider.

| **source** | LC1 | LC2 | OC1 | OC2 | OP1 | OP2 | LP1 | LP2 | Internet | Nameserver |
|---|---|---|---|---|---|---|---|---|---|---|
| **LC1** | | Y | Y | Y | Y | Y | N | N | Y | Y |
| **LC2** | Y | | Y | Y | Y | Y | N | N | Y | Y |
| **OC1** | Y | Y | | Y | Y | Y | N | N | Y | Y |
| **OC2** | Y | Y | Y | | Y | Y | N | N | Y | Y |
| **OP1** | Y | Y | Y | Y | | Y | N | N | Y | Y |
| **OP2** | Y | Y | Y | Y | Y | | N | N | Y | Y |
| **LP1** | N | N | N | N | N | N | | Y | Y | Y |
| **LP2** | N | N | N | N | N | N | Y | | Y | Y |

**Table 5.1:** Single peering pod connectivity matrix (Y = Success, N = Failure)

| **source** | LC1 | LC2 | OC1 | OC2 | OP1 | OP2 |
|---|---|---|---|---|---|---|
| **LC1** | Y | Y | Y | Y | Y | Y |
| **LC2** | Y | Y | Y | Y | Y | Y |
| **OC1** | Y | Y | Y | Y | Y | Y |
| **OC2** | Y | Y | Y | Y | Y | Y |

**Table 5.2:** Single peering consumer's services connectivity matrix (Y = Success, N = Failure)

| **source** | OC1 | OC2 | OP1 | OP2 | LP1 | LP2 |
|---|---|---|---|---|---|---|
| **OP1** | Y | Y | Y | Y | N | N |
| **OP2** | Y | Y | Y | Y | N | N |
| **LP1** | N | N | N | N | Y | Y |
| **LP2** | N | N | N | N | Y | Y |

**Table 5.3:** Single peering provider's services connectivity matrix (Y = Success, N = Failure)

### 5.4.2 Multiconsumer

In the multi-consumer scenario, the goal is to ensure that, in addition to protection on individual peering (consumers who cannot contact provider resources and vice

versa), isolation between multiple consumers offloading onto the same provider works.

### Infrastructure description

The infrastructure created for this test involves the use of two consumers, Rome and Venice, and one provider, Milan. On each cluster, both consumer and provider, a local namespace was created, not to be offloaded or shared with other clusters, containing a single pod. In addition to the local namespaces, a namespace was also created on each consumer to be offloaded to the Milan provider. In the offloaded namespaces, one pod is hosted on the consumer and one on the provider. Figure 5.2 shows the high-level diagram of the infrastructure for this scenario.

The pod name is composed as follows:

1. It starts with L if it is hosted in a private namespace, followed by the initial of the cluster name: R = Rome, M = Milan, V = Venice

2. It begins with O if it is hosted on an offloaded namespace, followed by the initial of the consumer cluster (R = Rome, V = Venice) and C if it is hosted on the consumer or P if it is hosted on the Provider.

Example: LR is a pod in a private namespace on the cluster Rome. ORC is a Pod created by the Consumer Rome in an offloaded namespace that is hosted on the Consumer side.

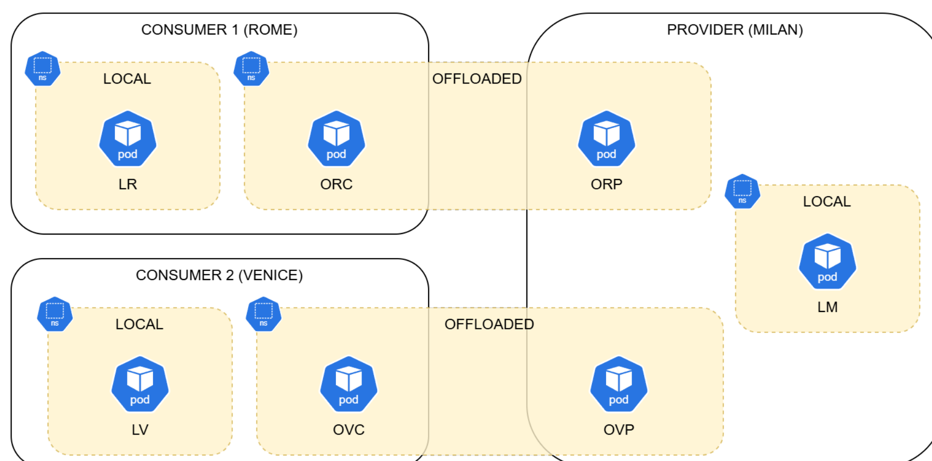


**Figure 5.2:** Multiconsumer test environment

### Results

The findings from this evaluation indicate that the system's behavior aligns with the observations from Scenario 1-1: intra-tenant communication is fully supported, while the consumer's resources remain strictly isolated from the provider's infrastructure.

Furthermore, the second anticipated outcome was successfully validated, as inter-tenant connectivity is completely restricted. This tenant-level isolation is enforced by predefined egress policies that establish a strict network perimeter around each consumer's environment, thereby preventing unauthorized outbound connections to external resources.

Table 5.4 shows the outcomes of Pod-to-Pod connections for both HTTP and ICMP requests, plus the HTTP and DNS requests to the public internet. Tables **??** and 5.6 show connections to services in the consumers and the provider, respectively.

| **source** | LR | LV | LM | ORC | ORP | OVC | OVP | Internet | Nameserver |
|---|---|---|---|---|---|---|---|---|---|
| **LR** | | | N | Y | Y | | N | Y | Y |
| **LV** | | | N | | N | Y | Y | Y | Y |
| **LM** | N | N | | N | N | N | N | Y | Y |
| **ORC** | Y | | N | | Y | | N | Y | Y |
| **ORP** | Y | N | N | Y | | N | N | Y | Y |
| **OVC** | | Y | N | | N | | Y | Y | Y |
| **OVP** | N | Y | N | N | N | Y | | Y | Y |

**Table 5.4:** Multiconsumer pod connectivity matrix (Y = Success, N = Failure)

| **source** | LR | ORC | ORP |
|---|---|---|---|
| **LR** | Y | Y | Y |
| **ORC** | Y | Y | Y |

| **source** | LV | OVC | OVP |
|---|---|---|---|
| **LV** | Y | Y | Y |
| **OVC** | Y | Y | Y |

**Table 5.5:** Multiconsumer consumers' services connectivity matrix (Y = Success, N = Failure)

| **source** | LM | ORC | ORP | OVC | OVP |
|---|---|---|---|---|---|
| **LM** | Y | N | N | N | N |
| **ORP** | N | Y | Y | N | N |
| **OVP** | N | N | N | Y | Y |

**Table 5.6:** Multiconsumer provider's services connectivity matrix (Y = Success, N = Failure)

### 5.4.3 Multiprovider

The goal of the multi-provider scenario is to ensure that leaf-to-leaf connectivity is allowed despite the security policies applied. Leaf-to-leaf connectivity requires that a provider be able to reach a pod offloaded to another provider via a service, passing through the consumer.

#### Infrastructure description

To check leaf-to-leaf connectivity, a consumer must be connected to multiple providers. For this scenario, a consumer called Rome and two providers called Milan and Venice were created. Each cluster has a local namespace that is not offloaded and not shared with other clusters, containing a single pod. In addition to this namespace, an offloaded namespace has been created on the Rome consumer on both Milan and Venice, containing three pods: one on Rome, one on Milan, and one on Venice. Figure 5.3 illustrates the high-level diagram of the infrastructure for this scenario.

The pod name is composed as follows:

1. It starts with L if it is hosted in a private namespace, followed by the initial of the cluster name: R = Rome, M = Milan, V = Venice

2. It begins with O if it is hosted on an offloaded namespace, followed by the initial of the hosting cluster: R = Rome, M = Milan, V = Venice

Example: LR is a pod in a private namespace on the cluster Rome. OM is a Pod created by the Consumer Rome in an offloaded namespace that is hosted on Milan.

#### Results

This evaluation yielded the anticipated outcomes, successfully verifying that leaf-to-leaf traffic is permitted while strictly preserving the consumer's established security perimeter.

The empirical results demonstrate that strictly local pods—those entirely isolated from peer-sharing mechanisms—remain securely confined, capable of communicating solely with resources within the same administrative domain. Conversely, pods within offloaded namespaces successfully maintain connectivity with consumer pods according to their defined visibility scopes (Table 5.7).

Furthermore, as detailed in Tables 5.8 and 5.9, offloaded pods can successfully establish cross-provider connections via exposed services, whereas such external communication is explicitly restricted for local pods.

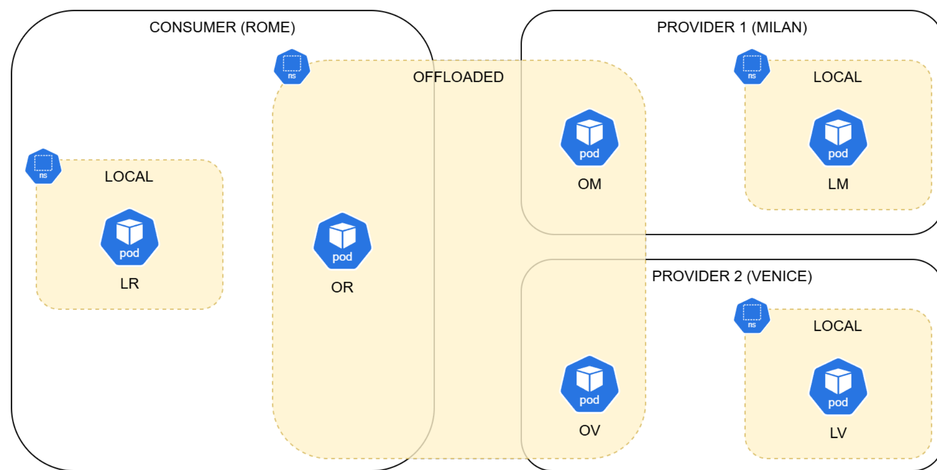


**Figure 5.3:** Multiconsumer test environment

| source | LR | LM | LV | OR | OM | OV | Internet | Nameserver |
|---|---|---|---|---|---|---|---|---|
| **LR** | | N | N | Y | Y | Y | Y | Y |
| **LM** | N | | | N | N | | Y | Y |
| **LV** | N | | | N | | N | Y | Y |
| **OR** | Y | N | N | | Y | Y | Y | Y |
| **OM** | Y | N | | Y | | | Y | Y |
| **OV** | Y | | N | Y | | | Y | Y |

**Table 5.7:** Multiprovider pod connectivity matrix (Y = Success, N = Failure)

| source | LR | OR | OM | OV |
|---|---|---|---|---|
| **LR** | Y | Y | Y | Y |
| **OR** | Y | Y | Y | Y |

**Table 5.8:** Multiprovider consumer's services connectivity matrix (Y = Success, N = Failure)

| source | LM | OR | OM | OV |
|---|---|---|---|---|
| **LM** | Y | N | N | N |
| **OM** | N | Y | Y | Y |

| source | LV | OR | OM | OV |
|---|---|---|---|---|
| **LV** | Y | N | N | N |
| **OV** | N | Y | Y | Y |

**Table 5.9:** Multiprovider providers' services connectivity matrix (Y = Success, N = Failure)

### 5.4.4 Differences with Cilium

The results obtained using Cilium demonstrate a notable divergence from those recorded with Flannel and Calico. This discrepancy can be attributed to an operational limitation associated with Kubernetes NetworkPolicies.

As highlighted in the official Cilium documentation, `ipBlock` rules fail to evaluate correctly when the specified IP address block falls within the cluster's Pod CIDR. Consequently, because certain network manager filters are provisioned using this rule type, they fail to function as intended, leading to erroneous blocking of legitimate network traffic.

The subsequent tables delineate the deviations observed in pod-to-pod connectivity relative to the previous baseline tests; connections that were inadvertently dropped are denoted by an 'X'. Table 5.10 shows the results in the single peering scenario, 5.11 shows the multi-consumer and 5.12 shows the multi-provider.

| **source** | LC1 | LC2 | OC1 | OC2 | OP1 | OP2 | LP1 | LP2 |
|---|---|---|---|---|---|---|---|---|
| **LC1** | | Y | Y | Y | **X** | **X** | N | N |
| **LC2** | Y | | Y | Y | **X** | **X** | N | N |
| **OC1** | Y | Y | | Y | **X** | **X** | N | N |
| **OC2** | Y | Y | Y | | **X** | **X** | N | N |
| **OP1** | Y | Y | Y | Y | | Y | N | N |
| **OP2** | Y | Y | Y | Y | Y | | N | N |
| **LP1** | N | N | N | N | N | N | | Y |
| **LP2** | N | N | N | N | N | N | Y | |

**Table 5.10:** Cilium single peering pod connectivity matrix (Y = Success, N = Failure, X = Incorrectly blocked)

| **source** | LR | LV | LM | ORC | ORP | OVC | OVP |
|---|---|---|---|---|---|---|---|
| **LR** | | | N | Y | **X** | | N |
| **LV** | | | N | | N | Y | **X** |
| **LM** | N | N | | N | N | N | N |
| **ORC** | Y | | N | | **X** | | N |
| **ORP** | Y | N | N | Y | | N | N |
| **OVC** | | Y | N | | N | | **X** |
| **OVP** | N | Y | N | N | N | Y | |

**Table 5.11:** Cilium multiconsumer pod connectivity matrix (Y = Success, N = Failure, X = Incorrectly blocked)

| **source** | LR | LM | LV | OR | OM | OV |
|---|---|---|---|---|---|---|
| **LR** | | N | N | Y | **X** | **X** |
| **LM** | N | | | N | N | |
| **LV** | N | | | N | | N |
| **OR** | Y | N | N | | **X** | **X** |
| **OM** | Y | N | | Y | | |
| **OV** | Y | | N | Y | | |

**Table 5.12:** Cilium multiprovider pod connectivity matrix (Y = Success, N = Failure, X = Incorrectly blocked)

### 5.4.5 Comparison with the FirewallConfiguration-only model

During the initial analysis phase, it was decided to proceed with a model relying exclusively on FirewallConfiguration. However, this approach was subsequently

abandoned after observing that network plugins utilizing an eBPF-based dataplane allowed packets to bypass the firewall rules implemented in nftables.

Following the experimentation with the new hybrid model, Pod-to-Pod connectivity tests were executed in a direct connectivity scenario comprising a single consumer and a single provider to demonstrate the distinct behaviors resulting from the specific CNI plugin deployed.

The results obtained with Flannel and Calico operating in nftables mode are consistent with prior findings from the new hybrid model. Conversely, Cilium and Calico operating in eBPF mode exhibit significant variations, as illustrated in Table 5.13.

As can be observed, certain connections are still successfully blocked; however, this applies exclusively to intra-cluster traffic. Rules governing inter-cluster traffic are bypassed, as the dataplane leverages TC or XDP hooks for this specific type of traffic.

This demonstrates the theoretical feasibility of deploying both solutions on nftables dataplanes, thereby granting administrators the flexibility to select their preferred model.

| **source** | LC1 | LC2 | OC1 | OC2 | OP1 | OP2 | LP1 | LP2 |
|---|---|---|---|---|---|---|---|---|
| **LC1** | | Y | Y | Y | Y | Y | N | N |
| **LC2** | Y | | Y | Y | Y | Y | N | N |
| **OC1** | Y | Y | | Y | Y | Y | N | N |
| **OC2** | Y | Y | Y | | Y | Y | N | N |
| **OP1** | Y | Y | Y | Y | | Y | **X** | **X** |
| **OP2** | Y | Y | Y | Y | Y | | **X** | **X** |
| **LP1** | N | N | N | N | **X** | **X** | | Y |
| **LP2** | N | N | N | N | **X** | **X** | Y | |

**Table 5.13:** Single peering pod connectivity matrix with the old model in Calico eBPF and Cilium (Y = Success, N = Failure, X = Incorrectly permitted)

### 5.4.6 Performance impact

In addition to functional validation, evaluating the performance overhead introduced by the implementation of network policies is a critical aspect of assessing the viability of the Liqo Network Manager. In highly distributed cloud-native environments, security enforcement mechanisms must not become bottlenecks that degrade application throughput or heavily tax node resources. This section analyzes the quantitative impact of the deployed firewalling rules and network policies on CPU utilization and network latency.

### Experimental Methodology

To rigorously measure the performance impact, the experimental setup consisted of a single peering scenario involving a consumer and a provider cluster. The consumer cluster was configured to host a deployment of 50 NGINX pods, which acted as the target web workload. Conversely, the provider cluster hosted a dedicated pod running k6, an open-source load testing tool, tasked with generating a high volume of HTTP requests towards the offloaded NGINX services.

The evaluation was conducted in two distinct phases under a constant load of 400 Virtual Users (VUs):

- Baseline: Traffic generation without any custom security policies applied.
- Enforced: Traffic generation with the complete set of generated NetworkPolicies and FirewallConfigurations active on the dataplane.

### CPU Utilization Analysis

The CPU utilization metrics were collected during the execution of the load tests to observe the strain placed on the kernel's network stack (particularly sys and softirq times) by packet inspection and rule evaluation.

| **CPU metric** | Baseline (Without Policies) | Enforced (With Policies) | Delta |
|---|---|---|---|
| **%usr** | 11.51% | 11.11% | -0.40% |
| **%sys** | 22.90% | 23.14% | +0.24% |
| **%soft** | 19.35% | 20.27% | +0.92% |
| **%idle** | 46.17% | 45.42% | -0.75% |

**Table 5.14:** Performance impact results

The empirical data (table 5.14) demonstrates that the enforcement of active network policies does not introduce any noticeable processing overhead. The observed variations—such as a mere 0.75% decrease in idle time and marginal shifts of +0.92% in %soft (softirq) and +0.24% in %sys CPU time—do not indicate a structural increase in computational cost. Instead, these minor deltas fall entirely within normal statistical variance and are likely attributable to external background processes or standard systemic noise rather than the evaluation of packets against the nftables sets and maps. This confirms that the pre-computation and grouping logic implemented in the controller successfully optimizes the underlying firewall structures without placing any actual additional strain on the CPU.

### Network Latency and Throughput

The load testing results acquired via k6 further corroborate the CPU metrics, demonstrating that the introduction of security boundaries does not severely hinder network performance. Both tests successfully completed 100% of their checks (HTTP 200 OK) with zero failed requests, proving the stability of the connections under load.

| **k6 Metric** | Baseline (Without Policies) | Enforced (With Policies) | Delta |
|---|---|---|---|
| **Throughput (reqs/s)** | 3791.26 | 3771.88 | -0.51% |
| **Average Latency** | 4.43ms | 4.83ms | +9.02% |
| **Median Latency** | 2.92ms | 3.13ms | +7.19% |
| **p(90) Latency** | 9.20ms | 10.24ms | +11.30% |
| **p(95) Latency** | 12.76ms | 14.46ms | +13.32% |

**Table 5.15:** Latency impact results

The comparative analysis (table 5.15) reveals that overall throughput remains virtually unchanged, experiencing a negligible drop of 0.5% (from 3791 req/s to 3771 req/s). Similarly, the latency profile shows minimal degradation. The average HTTP request duration increased by only 0.40ms. When observing the tail latency percentiles, the 90th and 95th percentiles increased by roughly 1.04ms and 1.70ms, respectively.

### Conclusion

Ultimately, the quantitative tests demonstrate that the proposed Network Manager architecture is highly performant. The translation of high-level intents into low-level primitives does not introduce noticeable bottlenecks, ensuring that the Computing Continuum can be secured without sacrificing the speed and efficiency required by modern distributed applications.

# Chapter 6

# Conclusion and future work

The transition from centralized cloud computing models to the Computing Continuum paradigm has radically redefined IT infrastructure management requirements, introducing unprecedented complexity in the areas of orchestration and network security. In distributed scenarios, inherently characterized by a plurality of heterogeneous providers operating in a context of mutual distrust, traditional security models based on physical nodes are obsolete. Although the Liqo platform excels at extending the Kubernetes control plane beyond administrative boundaries through dynamic peering, enabling transparent computation offloading, its basic network model allowed indiscriminate connectivity between clusters, presenting critical vulnerabilities in multi-tenant scenarios. This thesis work successfully addressed and resolved this issue by designing and implementing an advanced Network Manager natively integrated into the Liqo ecosystem.

The research and development process highlighted a crucial architectural challenge in the cloud-native networking landscape: the high heterogeneity of Container Network Interface (CNI) plugins. Initially, the project envisaged the exclusive use of Liqo's FirewallConfiguration resource to translate security intentions into low-level rules. However, experimental analysis revealed that modern data planes, implemented using eBPF technologies (as in the case of Cilium), operating at deep levels of the kernel, can bypass the standard chains of the Netfilter framework on which traditional rules are based. This discovery necessitated a strategic paradigm shift toward a hybrid enforcement model. By delegating intra-cluster traffic control logic to standard NetworkPolicy and shielding inter-cluster traffic via FirewallConfiguration applied within the gateway's isolated network namespace, the final architecture achieved a robust "defense-in-depth" posture that is largely agnostic to the underlying network plugin.

From an engineering perspective, the developed solution abstracts infrastructure complexity by allowing administrators to define high-level declarative security intents via the Custom Resource Definition (CRD) PeeringConnectivity. The

controller's translation engine, aided by a robust reconciliation loop, efficiently resolves complex logical abstractions—such as offloaded namespaces and "cluster slices"—translating them into concrete network primitives. A further fundamental contribution of this work lies in the extension of Liqo's core firewalling capabilities. The introduction of connection tracking (ctstate) support to manage connection directionality and named sets management to optimize IP rule scalability have transformed a basic solution into an enterprise-grade traffic control system. The rigorous validation methodology adopted has empirically confirmed the reliability of the proposed model. The use of an automated Infrastructure-as-Code framework has allowed the controller to be tested on complex topologies.

The rigorous validation methodology adopted has empirically confirmed the reliability of the proposed model. The use of an automated Infrastructure-as-Code framework has allowed the controller to be tested on complex topologies and heterogeneous CNI configurations, including Flannel, Calico, and Cilium. The results unequivocally demonstrated that the Network Manager successfully enforces the required security boundaries, effectively handling complex use cases such as Provider and Consumer protection, transitive Leaf-to-Leaf connectivity, and strict isolation in multi-tenant environments. While a specific behavioral deviation related to the evaluation of ipBlock rules within CIDR Pods when using Cilium was identified, the overall architecture proved resilient and perfectly capable of ensuring tenant segregation in real-world scenarios.

In summary, this thesis bridges the semantic gap between high-level administrative directives and low-level enforcement, providing a solid foundation for the implementation of granular policies within the Computing Continuum. While the developed solution currently focuses on granular firewalling to allow organizations to securely outsource workloads to untrusted administrative domains, it ultimately serves as the baseline for a comprehensive network manager. Future work should focus on expanding this foundation in two primary directions. First, advanced routing capabilities could be introduced, such as routing internet-bound traffic from an offloaded pod back through its consumer network. Second, the system's architecture could be generalized to increase interoperability; while the current model relies on the creation of standard FirewallConfigurations and NetworkPolicies, extending support to encompass CNI-specific network policies would greatly enhance its adaptability. Together, these developments will further consolidate the ecosystem as the orchestrator of choice for next-generation hybrid and distributed infrastructures.

# Bibliography


[1] Giulio Brazzo. «Securing the computing continuum with fine-grained automatic network policies». Master's degree thesis. Politecnico di Torino, 2025. URL: `https://webthesis.biblio.polito.it/36360/` (cit. on p. 2).

[2] Engineering Ingegneria Informatica S.p.A. *Engineering Ingegneria Informatica - Digital Transformation Company*. Engineering Ingegneria Informatica S.p.A. URL: `https://www.eng.it/it` (visited on 03/12/2026) (cit. on p. 2).

[3] Engineering Ingegneria Informatica S.p.A. *IPCEI-CIS AVANT (dAta and infrastructural serVices for the digitAl coNTinuum)*. Engineering Ingegneria Informatica S.p.A. URL: `https://www.eng.it/it/insights/stories/research-projects/ipcei-cis-avant` (visited on 03/12/2026) (cit. on p. 2).

[4] Kubernetes Authors. *Network Policies - Kubernetes Documentation*. URL: `https://kubernetes.io/docs/concepts/services-networking/network-policies/` (visited on 03/09/2026) (cit. on p. 10).

[5] Cilium Authors. *CNI Benchmark: Understanding Cilium Network Performance*. May 2021. URL: `https://cilium.io/blog/2021/05/11/cni-benchmark/` (visited on 03/09/2026) (cit. on p. 11).

[6] Liqo Authors. *Peering - Liqo 1.1.2 Documentation*. URL: `https://docs.liqo.io/en/v1.1.2/features/peering.html` (visited on 03/09/2026) (cit. on p. 18).

[7] Liqo Authors. *Offloading - Liqo 1.1.2 Documentation*. URL: `https://docs.liqo.io/en/v1.1.2/features/offloading.html` (visited on 03/09/2026) (cit. on p. 18).

[8] Liqo Authors. *Network Fabric - Liqo 1.1.2 Documentation*. URL: `https://docs.liqo.io/en/v1.1.2/features/network-fabric.html` (visited on 03/09/2026) (cit. on p. 18).

[9] Cilium Authors. *Network Policy - Cilium 1.19 Documentation*. 2024. URL: `https://docs.cilium.io/en/v1.19/network/kubernetes/policy` (visited on 03/09/2026) (cit. on p. 31).

[10] Riccardo Tornesello. *add connection tracking state matching support*. `https://github.com/liqotech/liqo/pull/3144`. GitHub Pull Request, repository liqotech/liqo. 2026 (cit. on p. 44).

[11] Riccardo Tornesello. *Add named sets to FirewallConfiguration*. `https://github.com/liqotech/liqo/pull/3155`. GitHub Pull Request, repository liqotech/liqo. 2026 (cit. on p. 46).

[12] Riccardo Tornesello. *liqo-network-manager*. `https://github.com/riccardotornesello/liqo-network-manager`. GitHub repository. 2026 (cit. on p. 54).

[13] Riccardo Tornesello. *kubernetes-testbench*. `https://github.com/riccardotornesello/kubernetes-testbench`. GitHub repository. 2026 (cit. on pp. 58, 59).