Contents

1. Introduction 3

2. Background and Related Work 5

2.1. Basics 5

2.1.1. Containerization 5

2.1.2. Container orchestration 6

2.2. Kubernetes architecture 7

2.2.1. Control plane 8

2.2.2. Nodes 9

2.2.3. Objects 9

2.2.4. Cluster networking 11

2.3. The concept of traffic engineering in Kubernetes 12

2.3.1. Ingress traffic management 12

2.3.2. Egress traffic management 14

2.4. Container network interface (CNI) 14

2.4.1. Overview of selected CNI plugins 15

2.5. Related work 16

3. Introduction to Egress and Ingress Scenarios in Selected CNI Plugins 18

3.1. Egress scenario: routing outgoing traffic via Egress Gateway 18

3.1.1. Egress gateway in selected CNI plugins 19

3.2. Ingress scenario: splitting incoming traffic via Gateway API 23

3.2.1. Traffic splitting in selected CNI plugins 25

4. Implementing Egress and Ingress Scenarios Using Selected CNI Plugins 28

4.1. Tools and automation 28

4.2. Egress scenario implementation ... 31
4.3. Ingress scenario implementation ... 36
4.3.1. Cluster provisioning ... 36
4.3.2. Infrastructure ... 36
4.3.3. The differences between cloud and local runs ... 40
5. Performance comparison ... 41
5.1. Egress Scenario ... 41
5.2. Ingress Scenario ... 49
6. Conclusions ... 54

1. Introduction

The purpose of this thesis is to evaluate and compare the traffic engineering capabilities of Container Networking Interface (CNI) plugins in Kubernetes clusters. The study focuses on understanding how CNI plugins can manage both incoming and outgoing traffic within the cluster. Traffic engineering plays a crucial role in efficient Kubernetes networking performance and security. By implementing traffic policies and load balancing, data flow can be optimized, ensuring faster transmission and more secure communication with external services. By analyzing their usage in real-world scenarios, this thesis aims to identify the strengths and limitations of these plugins. The evaluation includes measurements of critical performance metrics, such as CPU usage, memory consumption, and networking benchmarks, to provide a clear view of their efficiency. The findings of this research are intended to guide Kubernetes users in selecting the most suitable CNI plugin for their specific traffic engineering needs for more efficient, reliable, and scalable Kubernetes networking solutions.

The study introduces containers, orchestration, and Kubernetes, the key technologies that transformed the deployment and management of applications in cloud environments. The work presents and analyzes the traffic engineering capabilities of Cilium and Antrea, two Kubernetes CNI plugins. The study is divided into two scenarios: egress and ingress. The first one demonstrates both CNI implementations and performance an egress gateway, which is used to control outgoing traffic from inside the cluster. Potential use cases are outlined, and networking performance, such as throughput and round-trip time, is compared across the CNIs, both with and without the egress gateway. In the ingress scenario, the Gateway API provided by Antrea and Cilium will be examined, with example use cases demonstrated. Special attention will be given to traffic weighting and its effectiveness in splitting incoming traffic.

Chapter 1 provides an overview of the thesis, establishing the context and problem statement. The chapter also outlines the structure of the subsequent chapters. Chapter 2 introduces Kubernetes concepts and points out current features of CNI plugins. Chapter 3 explains both egress and ingress scenarios, illustrating how different Container Network Interface plugins can affect each of the traffic management directions. Next, chapter 4 presents the implementation of both scenarios, along with a description of each tool used during the evaluation of CNI features. Chapter 5 compares the results gathered during the evaluation of CNI plugins and their implementations used in egress and ingress scenarios. Finally, Chapter 6 concludes the thesis by summarizing the key findings of the research. It also discusses how future work can evaluate the performance of CNI implementations in traffic engineering.

2. Background and Related Work

This chapter will introduce a concept of containerization, orchestration along with exploring fundamental concepts of Kubernetes tool, addressing the management of incoming (ingress) and outgoing (egress) traffic within a Kubernetes cluster. Finally, comparison of selected Container Network Interface plugins, pointing out their key features. The end will conclude with a literature overview.

2.1. Basics

In this section two key Kubernetes concepts will be outlined: containerization and orchestrations, showing their roles and benefits in application deployments. These concepts are fundamental for managing modern large-scale environments containing distributed systems.

2.1.1. Containerization

Containerization is packaging an application along with all necessary runtime stuff like libraries, executables or assets into an object called "container". The main benefits of containers are [1]:

– Portable and Flexible – a container can be run on bare metal or a virtual machine in the cloud, regardless of the operating system. Only container runtime software like Docker Engine or *containerd* is required, which allows interacting with the host system.

– Lightweight – a container shares the operating system kernel with the host machine, so there is no need to install a separate operating system inside.

– Isolated – it does not depend on the host's environment or infrastructure.

– Standardized – the Open Container Initiative standardizes runtime, image, and distribution specifications.

A container image is a set of files and configuration needed to run a container. It is immutable, only new images can be created with the latest changes. The layer contains one modification made to an image. All layers are cacheable and can be reused when building an image. The mechanism is useful when compiling large application components inside one container [2].

2.1.2. Container orchestration

Container orchestration is coordinated deploying, managing, networking, scaling, and monitoring containers process. It automates and manages whole container's lifecycle, there is no need to worrying about of deployed app, orchestration software like Kubernetes will take care of its availability [1].

K8s (Kubernetes) is an open-source orchestration platform capable of managing containers. Key functionalities are [3]:

– Automated rollouts and rollbacks – updates or downgrades the version of deployed containers at a controlled rate, replacing containers incrementally.

– Automatic bin packing – allows specifying the exact resources needed by a container (CPU, memory) to fit on the appropriate node.

– Batch execution – makes it possible to create sets of tasks that can be run without manual intervention.

– Designed for extensibility – permits adding features using custom resource definitions without changing the source code.

– Horizontal scaling – scales (replicates) the application based on its need for resources.

– IPv4/IPv6 dual-stack – allocates either IPv4 or IPv6 to pods and services.

– Secret and configuration management – allows storing, managing, and updating secrets. Containers do not have to be rebuilt to access updated credentials.

– Self-healing – restarts crashed containers or those that fail as specified by the user.

– Service discovery and load balancing – advertises a container using its DNS name or IP and load balances traffic across all pods in the deployment.

– Storage orchestration – mounts the desired storage, like local or cloud-provided storage, and makes it available for containers.

Understanding Kubernetes workflow becomes significantly easier by familiarizing yourself with its architecture, which will be discussed in the following section.

2.2. Kubernetes architecture

A Kubernetes cluster is a group of machines that run containers and provide all the necessary services to enable communication between containers within the cluster, as well as access to the cluster from the outside. There are two types of components, a control plane and worker node. A minimum of one of each is needed to run a container, but to provide a more robust and reliable production cluster, it is better to use two to three control plane nodes [3].

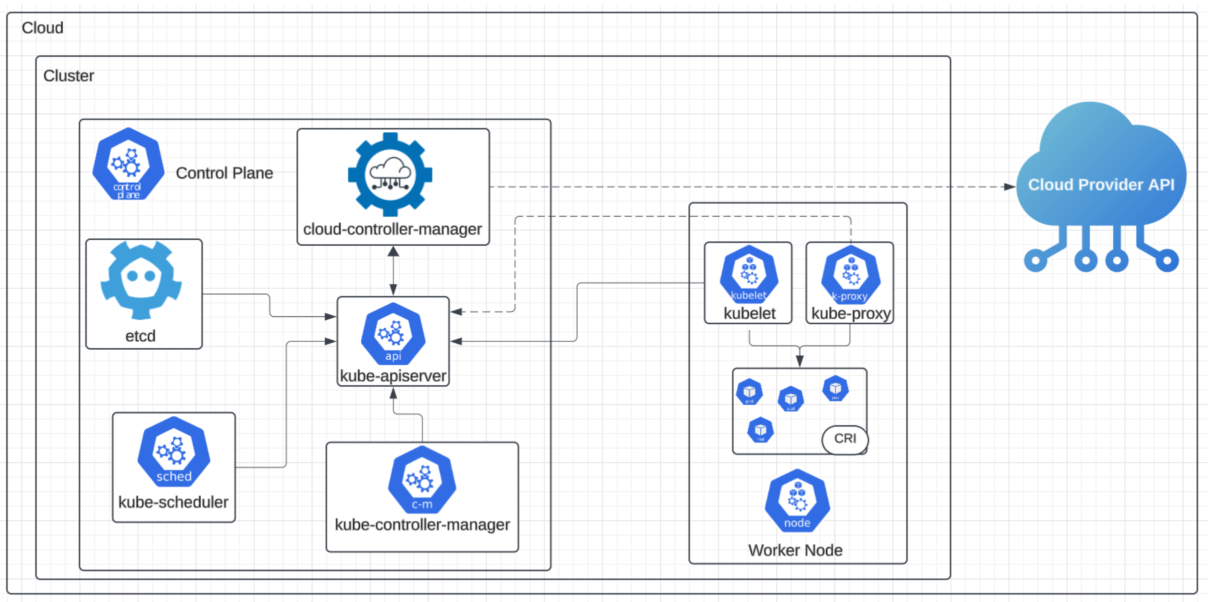


Figure 2.1: Kubernetes Cluster Architecture [3].

In Figure 2.1 there is graphical representation of Kubernetes cluster. Not all components shown in the figure are mandatory for Kubernetes to work correctly. At the control plane part, *cloud-controller-manager* might not be mandatory, in on-premises configurations where interacting with cloud provider is not needed. On the right side of the figure in node representation is *kube-proxy* component, which is not mandatory as some networking plugins can provide own implementation of proxy. This is an example of "Designed for extensibility", where Kubernetes can acquire 3rd-party features without changing its source code [3].

2.2.1. Control plane

Control plane is like a brain in Kubernetes cluster. Interaction with cluster using *kubectl* tool to perform requests is handled by *kube-apiserver*. It is responsible for communication with worker nodes running pods, the smallest unit managed by K8s that has containers inside [3].

cloud-controller-manager

This component allows Kubernetes clusters to interact with cloud provider's API. It is combined with *kube-controller-manager* as single binary and can be replicated. This is the only component that talks to the cloud provider, separating other components from direct communication with the cloud. When running without cloud environment this component is absent [3].

etcd

Etcd is an open-source distributed key-value store service often used in distributed systems. It is responsible for maintaining both the current state and its previous version in its persistent memory [3][4].

kube-apiserver

Exposes Kubernetes API to interact with a cluster. Takes responsibility for handling all requests from components and users. This is the component which answers cluster administrator requests sent by *kubectl* [3].

kube-controller-manager

Component which runs controller processes. Its compiled binary consists of multiple controllers. Example controllers are [3]:

- Node controller – observes worker nodes if are up and running,
- Job controller – responsible for batch execution jobs,
- EndpointSlice controller – connects services with pods.

kube-scheduler

Takes care of pods which are not assigned to a worker node yet. *kube-scheduler* is looking for node that meets pod's scheduling requirements and fit a pod on that node. Such a node is called feasible node [3].

2.2.2. Nodes

All the below-mentioned components run on every node in a cluster.

Container runtime

Node's key component, has ability to run, execute commands, manage, and delete containers in efficient way [3].

kube-proxy

Create networking rules which allow communicating with pods from outside cluster. If available *kube-proxy* uses operating system packet filtering to create set of rules. It is also able to forward traffic by itself. This component is optional, can be replaced with a different one if the desired one implements key features [3].

kubelet

It is responsible for managing containers inside pod on its node. Uses Container Runtime Interface to communicate with containers [3].

2.2.3. Objects

Namespace

The purpose of namespace object is to isolate groups of resources like pods, deployments, services etc. in a cluster. It helps to organize clusters into virtual sub areas of working space. If *Service* is created in some custom namespace <service-name>.<namespace-name>.svc.cluster.local DNS entry within cluster is created [3].

Pods

Pods are the smallest deployable objects in Kubernetes. They contain one or more containers, which can communicate with each other using localhost interface. Because they share IP addresses, they cannot use the same ports. It is useful when our service consists of two applications coupled together. For example, there is a pod which has two containers, one responsible for compiling code, the second one is creating a cache entry from compiled object and uploading it to some data storage. It is more logical to share data among containers in a pod than on node between pods, as it is easier. Scaling is simpler, as it involves replicating a single pod instead of managing two separate pods. Moreover, communication between applications happens using the localhost interface. In scenario where there are two pods, each with one container, ClusterIP *Service* is usually implemented. However, the most common approach is to run one container per pod, where a pod is just managing wrapper for containerized application. Also, rather than creating pod directly it is more common to use workload resource like *Deployment* [3].

ReplicaSet

Basically *ReplicaSet* consists of pod template and runs desired number of pods [3].

Deployment

Deployment is a higher-level abstraction over *ReplicaSet*, that manages its lifecycle. It provides more features like rolling back an app, as it keeps history of configurations [3].

DaemonSet

Running pods using DaemonSet guarantees that every node will have a copy of the desired pod (if resource requirements are met etc.). It can automatically add or remove pods if the number of nodes changes. The typical usage is creating monitoring pod on every node [3].

StatefulSet

StatefulSet, unlike a *Deployment*, is stateful. It saves an identity of each pod and if e.g., some persistent storage is assigned to specific e.g., database pod, when it dies, Kubernetes will recreate the pod on the same node as before [3].

Job

Runs pod that does one task and exists. Kubernetes will retry execution if pod fails specific number of tries set in its configuration [3].

CronJob

Behaves like *Job* but can regularly run at specified intervals for tasks like database backups or log rotation [3].

Service

Service exposes an application running inside a cluster by using an endpoint. As a pod is ephemeral resource and its address changes sometimes (e.g., when pod is recreated), it is better to create DNS name that resolves IP address. Moreover, the service will not advertise unhealthy pods. Usually, a service exposes one port per service, but for example web app might expose HTTP and HTTPS ports. There are four types of services [3].

1. ClusterIP – makes one pod available to other inside cluster by exposing application using inter-cluster IP address. Although it is oriented to be accessible within the cluster, objects like *Ingress* or *Gateway API* can expose service to the outside.

2. NodePort – by default allocates port (from range 30000-32767) to publish service on every node's IP address. In this scenario every node on the specified port acts like a proxy to the deployed app.

3. LoadBalancer – Kubernetes does not provide load balancer by default and when creating such a service it interacts with cloud provider to create external service for traffic balancing. A load balancer can be installed inside cluster.

4. ExternalName – allows pods inside Kubernetes to access external service using defined name rather than using IP address

2.2.4. Cluster networking

Networking is the most important thing in Kubernetes, the whole point is to obtain reliable and robust communication among containers, pods, services, nodes, and external systems in a cluster [3]. There are four types of network communication [3]:

1. container-to-container – communicates by sharing network resources inside a pod

2. pod-to-pod – every pod can communicate with any other pod without the need to use NAT as every of them has its own IP address [5].

3. pod-to-service – covered by service type ClusterIP, which provides inter-cluster IP address

4. external-to-service – held by services type NodePort and Loadbalancer, which expose pod to the outside

Kubernetes allocates IP addresses to nodes, services, and pods [3]:

– *kubelet* or *cloud-controller-manager*, depending on local or cloud infrastructure allocates IP address for nodes

– *kube-apiserver* allocates IP address for services

– the allocation of an IP address to a pod is the responsibility of the networking plugin, which is an implementation of *Container network interface (CNI)*.

2.3. The concept of traffic engineering in Kubernetes

Traffic Engineering is a key concept in Kubernetes to provide production-ready, reliable, and efficient network. In this section ingress and egress traffic will be explained.

2.3.1. Ingress traffic management

Ingress

Ingress is an object that manages outside cluster access to services inside a cluster. It is a single point of entry to route traffic to specified pod based on configuration. This is only a higher abstract object that specifies routing rules in cluster. Real functionalities are provided by an *Ingress Controller*. Nowadays the development of Ingress is frozen, Kubernetes authors pay attention to its successor a *Gateway API* [3].

Ingress Controller

Ingress Controller fulfills an *Ingress* and starts serving an application which performs configured rules. Any implementation has its own features, but common functionalities are L4/L7 load balancing, host and path-based routing, SSL termination. This is the real application that runs in a pod. Ingress Controller must be installed manually and is not part of Kubernetes, however the container orchestration tool developers maintain AWS, GCE, and nginx ingress controllers [3].

Gateway API

The functionalities of Gateway API are so wide, that the Kubernetes authors use term "project". The project focuses on L4 and L7 routing in a cluster. It succeeds *Ingress*, Load Balancing and service mesh APIs. The Gateway API resource model is role-oriented [6].

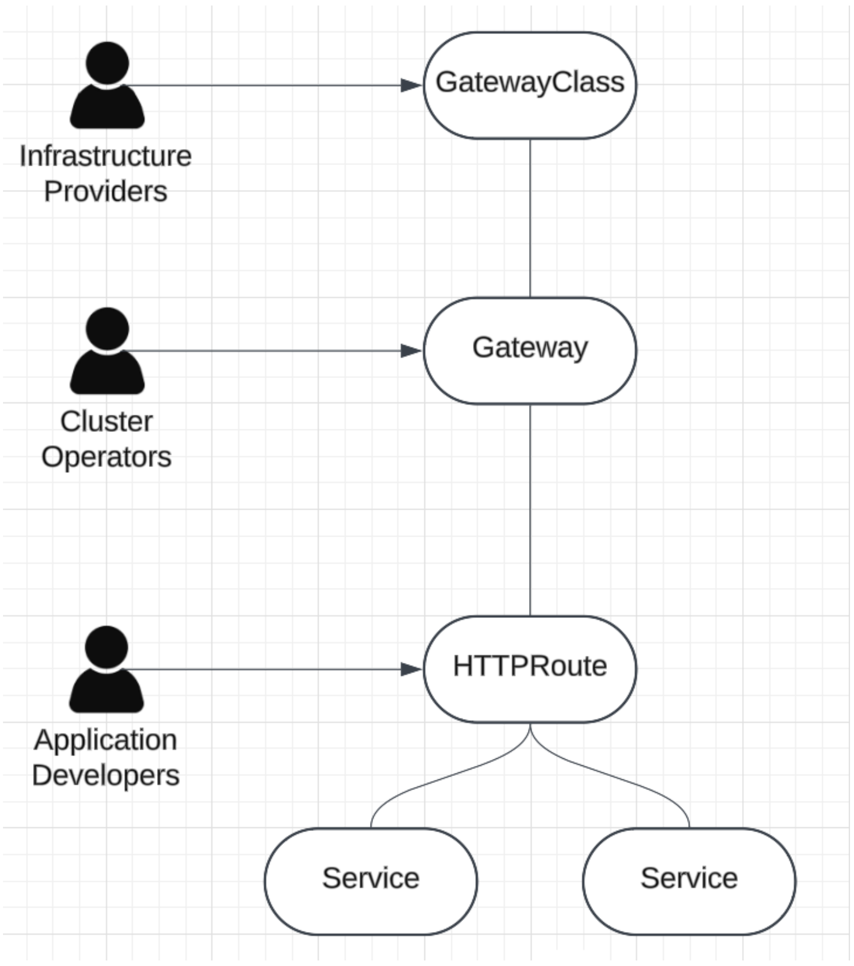


Figure 2.2: Gateway API roles-oriented resource model [6].

The model focuses on three separate groups of people who interact with a cluster on various levels.

On a top of Figure 2.2 there are infrastructure providers, who provide GatewayClass resource. They are responsible for the overall multiple clusters, rather than ensuring developers can access pods correctly [6]. The GatewayClass resource is a class of Gateway which can be created. It defines specific types of load balancing implementations and provides clear explanation of capabilities available in Kubernetes resource model. The functionality is like *Ingress*. There can be more than one GatewayClasss created [6].

Cluster operators are in the middle of Figure 2.2, they make sure that cluster meets requirements for several users. As maintainers define Gateway resource, some load balancing system is provisioned by GatewayClasss. Gateway resource defines specific instance which will handle incoming traffic. Allows defining specific protocol, port or allowed resources to route inbound traffic [6].

End users specified on Gateway API resource model in Figure 2.2 are application developers. They focus on serving applications to the clients by creating a resource named HTTPRoute. The resource defines HTTP routing from defined gateway to end API objects like service. It can to split traffic using "weight" as a key, which represents the percentage of the total traffic to be routed. GRPCRoute is similar, but operates on different protocols [3][6].

Gateway API is not an API Gateway. An API Gateway in general is responsible for routing, load balancing, information exchange manipulation and much more depending on specific implementation. Gateway API is set of three resources mentioned earlier, which creates a role-oriented Kubernetes service networking model. Creators of Gateway API provide a clear explanation: "Most Gateway API implementations are API Gateways to some extent, but not all API Gateways are Gateway API implementations" [6].

2.3.2. Egress traffic management

Egress traffic refers to connections which leave cluster and are initiated inside by pods. In contrast to the Ingres object, in Kubernetes there is no Egress resource, outgoing traffic route logic is implemented by Container Network Interface plugin. The most common approach in managing egress traffic is to use Kubernetes Network Policies to deny all outgoing traffic and then allow only key connections. The limitation is that all external services need to be specified with IP address in policies. Any change in external resource's IP requires a change in policy configuration. If any pod is trying to access external service, source network access translation (SNAT) needs to be performed to map inter-cluster pod IP to externally routed nodes IP. When the response is accessing cluster, SNAT is performing translation in opposite way. Another key egress concept in Kubernetes is an egress gateway. This is a node which proxies outgoing traffic from a cluster, specified by provided configuration (e.g., by labeling pods, depends on CNI implementation). The important thing is that the internal pod's IP address is masqueraded into IP address of an egress gateway, outside peer does not see ephemeral IP of a pod. Egress gateway is also a CNI specific implemented resource [7] [8].

2.4. Container network interface (CNI)

CNI is standardized by Cloud Native Computing Foundation set of API rules which defines container networking. CNI is responsible for pod-to-pod communication, which includes assigning IP addresses, configuring network interface inside container and routing [5].

2.4.1. Overview of selected CNI plugins

Table 2.1: Comparison of Antrea and Cilium [9][8].

Feature/Plugin	**Antrea**	**Cilium**
Dataplane	Open vSwitch	eBPF
Encapsulation	VXLAN or Geneve	VXLAN or Geneve
Encryption	IPsec or WireGuard tunnels	IPsec or WireGuard tunnels
Security	Extends Kubernetes Network Policies	Advanced security policies
Observability	Theia and Grafana for visualization	Hubble
Purpose	Simplified Kubernetes networking management	For large-scale cluters
Additional features	Network policies for non-Kubernetes nodes	BGP to advertise network outside cluster
Gateway API	No support	Fully supports Gateway API
Egress Gateway	Basic egress gateway capabilities	Advanced egress gateway support

Antrea is an open-source CNI plugin which is built on Open vSwitch which requires kernel version greater than 4.6 [9]. OvS is a virtual switch with capability of handling traffic flow between virtual machines and containers [10]. Antrea's focus is L3/L4 networking and security services, such as network policies. The resource is responsible for managing traffic flow between pods. By default, every pod can communicate with any other pod, but network policies can specify if pod A is able to talk to pod B [3]. Consider scenario with three pods, client, frontend, and backend. There is no need to allow client communication directly with backend, so network policies allow traffic flow from client to frontend and direct communication with backend is not allowed [7].

Cilium, open-source CNI which uses eBPF (extended Barkeley Packer Filter) for packet processing, security and deep observability using Hubble [8]. Some environments may not be suitable because Cilium requires a kernel version of 5.4 or higher [8]. eBPF is a technology that allows running defined programs, with custom logic inside operating system kernel in a privileged context without requiring of any kernel source code changes or loading modules. The lack of switching between kernel space and user space reduces latency [11].

Figure 2.3, shows standard container networking on the left side and Cilium eBPF container networking on the right. The whole point of eBPF networking is skipping overhead that comes from *iptables*. Moreover, eBPF implements hash tables for storing routing policies, which time complexity is O(log n), compared to *iptables* array O(n). It makes clear that large-scale clusters will benefit from using eBPF [12].

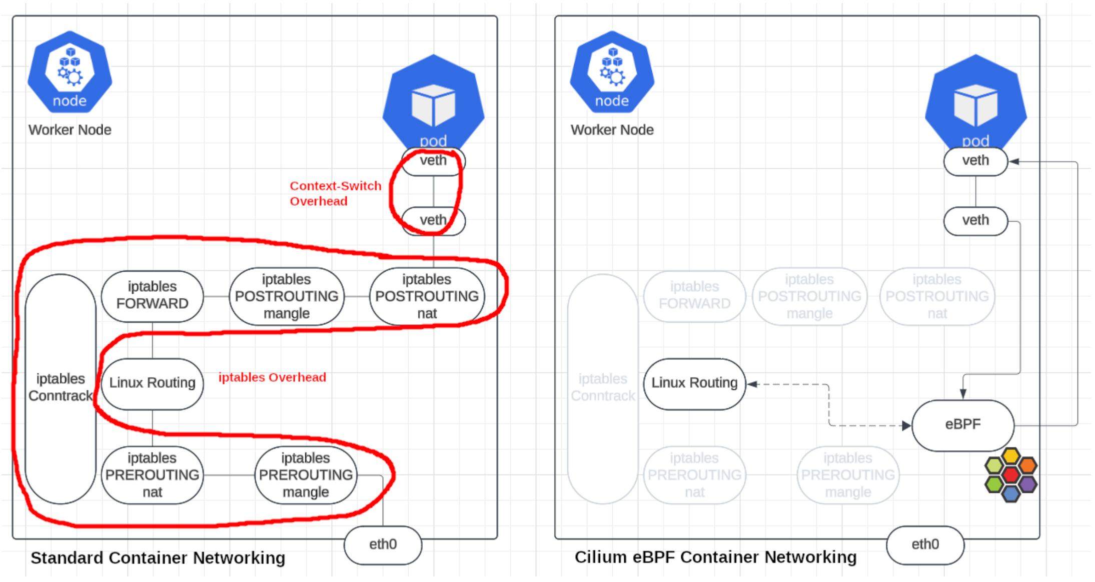


Figure 2.3: Cilium eBPF host-routing [13].

2.5. Related work

As discussed in [14] the performance of different CNIs can vary widely, some CNIs performing two to three times better than others, making it essential to choose the right plugin for a particular workload. Authors say that developing automated methodology of CNI plugin evaluation is a key aspect, specifically in large High-Performance Computing (HPC) environments. This allows for reproducible and consistent tests across different configurations, reducing the overhead of manual testing. To achieve that, tools like Ansible can be helpful. They state that Linux Kernel or NIC can be a bottleneck in networking performance, so they extend maximum buffer size, scale TCP window, disable TCP Selective Acknowledgement, increase SYN Queue Size, or enable Generic Receive offload. The paper shows results of comparison four CNI plugins, such as Antrea, Cilium, Calico, Flannel using TCP/UDP in base and with optimized system settings [14].

In [15] the authors measure CNI plugins for inter-host and intra-host communication using UDP and TCP protocols. They introduce the concept of CPU cycles per packet (CPP) to evaluate CNI efficiency. They measure CPP spent in each network component using the Linux *perf* tool. By measuring throughput, RTT, and latency, they compare how different CNI plugins (Flannel, Weave, Cilium, kube-router, Calico) compared to its network models [15].

In a paper, the same authors as before focus on functionality, performance, and scalability. They scale testbed up to 99 *Iperf* client and 99 *Iperf* server pod, mentioning that 100 pods is a limit for Kubernetes node [16].

The authors of [17] state that in the coming years, fifth generation mobile networks (5G) will deploy a significant part of their infrastructure in the cloud-native platforms, resulting in the creation of large-scale clusters. Such production environments containing thousands of pods require creating stable, reliable, and efficient networks. They do not focus their attention on which CNI uses in this scenario, rather highlight such concepts as highly performant networking, security, and observability. Authors state that the key to meet this expectation is eBPF (extended Barkeley Packet Filter) [17].

3. Introduction to Egress and Ingress Scenarios in Selected CNI Plugins

In modern Kubernetes networking, managing traffic flow into and out of a cluster is crucial for performance and security. Different CNI plugins provide distinct mechanisms for controlling network traffic, each offering a unique approach to networking implementation. Understanding how traffic is managed in both ingress and egress scenarios is essential for improving security, optimizing performance, and enhancing overall network efficiency.

3.1. Egress scenario: routing outgoing traffic via Egress Gateway

The Egress Gateway can play a key role in a cluster's security. It can enforce routing all outgoing connections initiated within labeled pods through the gateway node. The node can route all outgoing traffic through a security system to scan each packet for potential threats, ensuring that outgoing traffic only accesses secure services outside the cluster.

To illustrate this concept, consider a scenario where the IT department of a financial company manages Kubernetes clusters in their local laboratory. The infrastructure is used to create a production-ready, efficient, and secure environment for financial services, where handling sensitive data and adhering to strict regulatory standards is critical. Leaving unmonitored critical traffic leaving the cluster can create vulnerabilities, potentially exposing the system to data exfiltration from financial applications. They decided to analyze all outgoing traffic from financial services pods using intrusion detection system (IDS) software. However, they also provide services that do not require such robust security. Redirecting every request to the traffic analyzer would add unnecessary overhead to exposed applications and cause higher latency. Cluster operators decided to leverage an egress gateway to route all outgoing traffic from financial services into a security tool to monitor and analyze packets. However, end users started complaining that their applications were showing errors like "503 Service Unavailable." The IT administrators began troubleshooting and concluded that the egress gateway was acting as a bottleneck in the cluster. They started searching online for solutions and decided to create separate gateways for each deployment of their services [18]. End users stopped complaining about poor service availability.

3.1.1. Egress gateway in selected CNI plugins

Container Network Interface (CNI) plugins implement their own egress gateways, offering unique features. This section explores the capabilities of the Antrea and Cilium CNI plugins, focusing on how they handle outbound traffic and integrate with other networking components. Understanding these implementations is essential for Kubernetes operators to select the right CNI plugin for their specific requirements.

Antrea

The Antrea Egress CRD (*Custom Resource Definition*) API is a resource that controls how pods in a cluster access external services. This resource specifies which egress IP should be used by selected pods. When a pod communicates with an external network, the traffic is routed through the node that has the specified egress IP (egress gateway). The source IP address of the traffic is then translated to the configured IP address [9].

Figure 3.1 shows the architecture of the communication flow when an egress gateway is configured in the Antrea CNI. When a pod running on a Kubernetes node tries to access an external service (assuming it is labeled to route its outbound traffic through the egress node), the traffic is tunneled (through OvS) to the gateway node, and the Antrea Agent performs SNAT. After translation, the next network peer that communicates with the egress gateway will see its IP as the source IP, instead of the IP address of the pod [9].

Let’s explain the egress configuration YAML from Listing 1 [9]:

– Antrea allows matching the pods that route through the egress gateway based on two criteria:

  1. *namespaceSelector* – specifies which pods within the specified namespace should redirect outbound traffic;

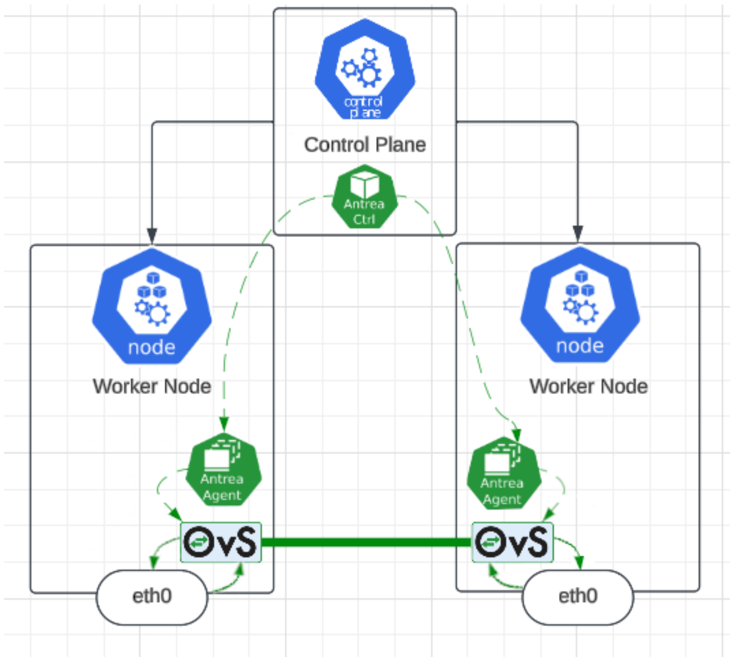


Figure 3.1: Antrea Egress Architecture [9].

2. *podSelector* – selects pods with the specified labels. For example, it can match pods labeled with role=web to redirect traffic;

– *egressIP* – specifies SNAT IP address of an egress gateway, to which traffic is tunneled;

– *externalIPPool* – name of *externalIPPool* resource which contains pool of IP addresses to allocate if egressIP is not set.

It is possible to configure a failover egress gateway node using Antrea. To do this, *egressIP* and *externalIPPool* must be set (with *egressIP* being part of *externalIPPool*). When the current egress gateway stops working, another node within the *externalIPPool* will be selected. This infrastructure, with a failover service, is part of a high-availability setup for production environments (useful, for example, for the IT department previously mentioned) [9].

The *ExternalIPPool* resource from Listing 2 can be configured with the following fields [9]:

– *ipRanges* – IP pools range can be configured using a pair of IP (start and end), or by setting CIDR (Classless Inter-Domain Routing) range

– *nodeSelector* – will apply only on nodes specified by this field, e.g., nodes labeled with network-role: egress-gateway

```yaml
apiVersion: crd.antrea.io/v1alpha2
kind: Egress
metadata:
    name: egress-prod-web
spec:
    appliedTo:
    namespaceSelector:
        matchLabels:
        env: prod
    podSelector:
        matchLabels:
        role: web
    egressIP: 10.10.0.8
    externalIPPool: prod-external-ip-pool
status:
    egressNode: node01
```

Listing 1: Egress resource example [9].

```yaml
apiVersion: crd.antrea.io/v1alpha2
kind: ExternalIPPool
metadata:
    name: prod-external-ip-pool
spec:
    ipRanges:
        - start: 10.10.0.2
          end: 10.10.0.10
        - cidr: 10.10.1.0/28
    nodeSelector:
    matchLabels:
        network-role: egress-gateway
```

Listing 2: *ExternalIPPool* resource example [9].

Cilium

To take advantage of Cilium's egress gateway features, eBPF masquerading must be enabled, and the node's *kube-proxy* component must be replaced with Cilium's implementation [8]. As shown in Figure 3.2, the Cilium agent injects routing information into eBPF maps within the kernel (relying on kernel support for eBPF features). Traffic from worker nodes is redirected to the egress gateway node, where it is SNATed and leaves the cluster. These routes, defined by Cilium policies configured in the control plane, ensure that every node is aware of which pods should redirect traffic to the designated egress node [8]. For node-to-node communication, Cilium encapsulates all traffic using UDP-based VXLAN or Geneve protocols [8].

Similar to Antrea egress resources, Cilium has its own *CiliumEgressGatewayPolicy* present in Listing 3 [8]:

Cilium allows matching the traffic that route through the egress gateway by [8]:

– *podSelector* – matching pods based of used selector, like previous matching labels in

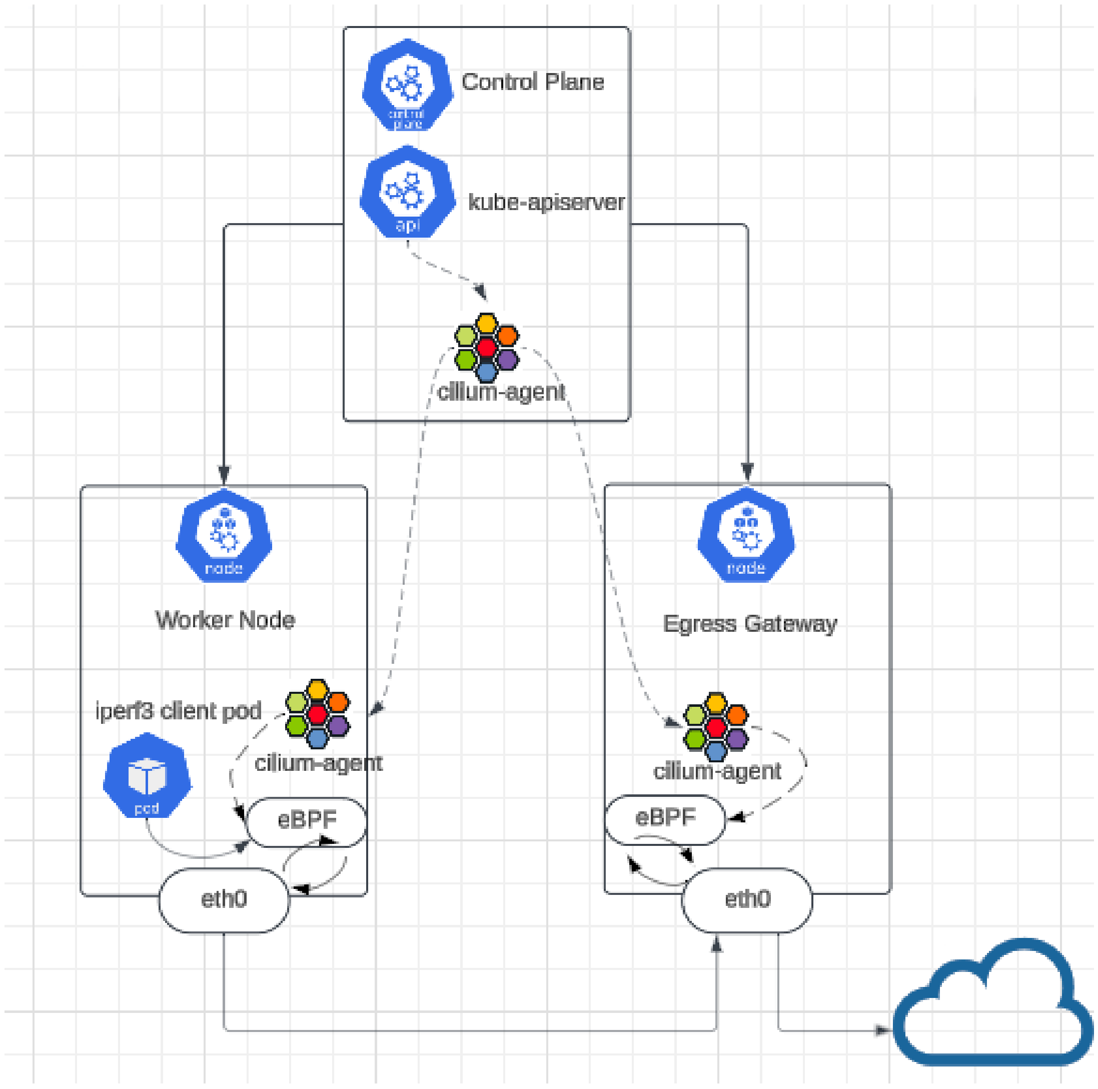


Figure 3.2: Cilium Egress Architecture [19].

Antrea, or by matching expressions (key operator, values). More than one *podSelector* can be used

– *destinationCIDRs* – an app in pod is requesting some external service, if this resource is match by defined CIDR, the request is routed to egress gateway. For 0.0.0.0/0 all traffic is outgoing by egress gateway. Setting *excludedCIDRs* is possible to exclude some IPs.

Selecting an egress gateway can be done in three ways: by matching node labels, using IP address in *egressIP* field (as in Antrea) or by interface name [8].

Both Antrea and Cilium allow us to configure an egress gateway in multiple ways. Cilium offers more flexibility in defining which traffic should be routed through the egress gateway. Unlike Antrea, Cilium can specify traffic by destination CIDR. While Cilium provides more capabilities for matching egress traffic, Antrea's implementation allows for the creation of a

```yaml
apiVersion: cilium.io/v2
kind: CiliumEgressGatewayPolicy
metadata:
name: egress-sample
spec:
selectors:
- podSelector:
    matchLabels:
        org: empire
        class: mediabot
        io.kubernetes.pod.namespace: default
    matchExpressions:
        - {key: testKey, operator: In, values: [testVal]}
        - {key: testKey2, operator: NotIn, values: [testVal2]}
destinationCIDRs:
- "0.0.0.0/0"
excludedCIDRs:
- "192.168.1.0/24"
egressGateway:
    nodeSelector:
    matchLabels:
        node.kubernetes.io/name: a-specific-node
    egressIP: 10.168.60.100
```

Listing 3: Egress resource example [9].

failover node, which will route traffic if the primary one fails. Creating a high-availability egress gateway in Cilium is possible only with the enterprise, paid version of the plugin. Additionally, Cilium leverages eBPF, which is designed for large-scale clusters [8]. It is not clear which egress gateway CNI implementation is the best to use, as each has its advantages and drawbacks. In the next section, both gateways will be evaluated using networking tools in a local environment.

3.2. Ingress scenario: splitting incoming traffic via Gateway API

The Gateway API, as a successor to the Ingress object, provides enhanced features for traffic management and introduces a role-oriented approach to separate Kubernetes user and operator concerns. It supports traffic splitting, header modification, and URL rewriting. Additionally, the Gateway API supports key protocols such as HTTP, HTTPS, TCP, UDP, and gRPC. With its wide range of features, it can be used in various ways [19].

Canary Deployment

Canary Deployment is one of the most common methods used to roll out a new application version to end users, ensuring that everything is working as expected before the full release. The core idea is to release new versions of software only to a small group of users, leaving most users unaware of the new release [20].

This is where the traffic splitting feature of the Gateway API can be used [8]. There are typically five stages in the deployment process [20]:

- Initial state – The stable version of the application is served.
- Canary stage – The updated version of the application is visible only to 5% of users. As some users interact with the newly provisioned software, the most common errors should be visible (if any).
- Early stage – The second stage of canary deployment, where the new app is available to 25% of total connections. At this point, less frequent bugs may be observed.
- Mid stage – The app is available to 50% of end users. Half of the traffic is routed to the latest version of the app. At this point, the performance of the rolled-out software is monitored.
- Late stage – Most of the traffic (75%) is handled by the recent version of the application. This stage precedes the full release of the new software.
- Full stage – 100% of users are served the new application version.

If any anomalies are detected during any stage of the canary deployment, the new application version should be immediately rolled back.

The Gateway API is not designed for software deployment and does not have capabilities for rolling back applications automatically. In this usage, the gateway is used only for weighted traffic splitting.

Traffic Mirroring with Gateway API

A company is offering a weather API, but not all endpoints are publicly available. Some features are secured and paid. Securing these paid interfaces is not as critical as securing more sensitive and confidential data. Recently, the company decided to start analyzing incoming traffic on these secured endpoints to ensure that only authorized requests are being handled. However, the company does not have the infrastructure capabilities to analyze all incoming traffic for these endpoints. As their services are HTTP-based inside a Kubernetes cluster, they can take advantage of the Gateway API's traffic splitting feature. The security team decided to route 40% of the incoming traffic to a traffic analyzer to evaluate whether and how requests might bypass the paywall. The cluster management team (acting as both infrastructure providers and cluster operators, according to the roles in the Gateway API model) decided to split the traffic using the Gateway API, as they need general usage of the API Gateway (since the company offers RESTful APIs). Pure traffic splitting is not sufficient in this case because all incoming traffic must be handled in response. The solution is to split 40% of the traffic to a different Kubernetes service, mirror the traffic to the analyzer, and then route it back to the pod containing the application. While the cluster managers implement the second deployment to mirror requests to the traffic analyzer, the app developers created an HTTPRoute object with appropriate weights for each of the services. Figure 3.3 shows how the cluster infrastructure might look in this case.

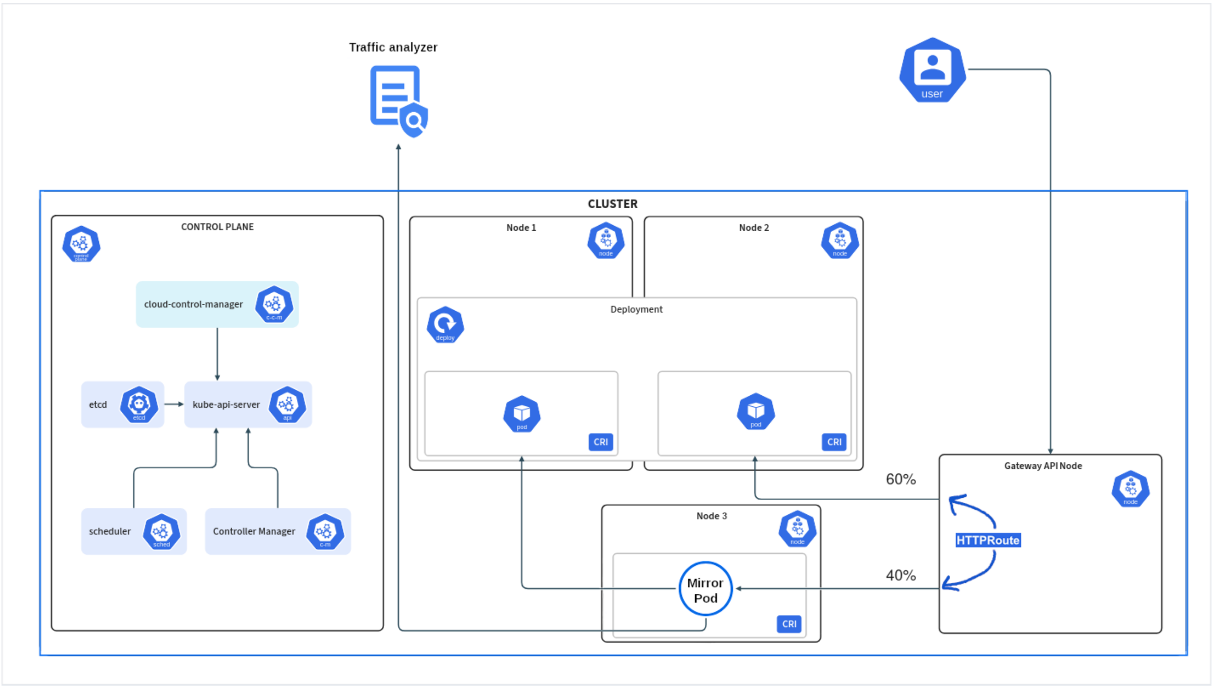


Figure 3.3: Example Kubernetes cluster with traffic mirroring [3].

3.2.1. Traffic splitting in selected CNI plugins

Unfortunately, the Antrea plugin does not provide an implementation of the Gateway API. In fact, Cilium is the only CNI plugin that offers this functionality. To evaluate the cluster networking implementation with Antrea CNI, the NGINX Gateway Fabric can be used instead.

Antrea + NGINX

Figure 3.4 shows an example cluster where the Gateway API is configured to work in the canary stage. Antrea CNI is installed, and the antrea-ctrl and antrea-agent pods are deployed on the nodes. OvS tunneling among nodes is set up [9]. On the control plane node, the NGINX

Gateway API is deployed as a pod [21]. This differs from Cilium, where the Gateway API is not run as a single pod [8]. In this setup, traffic is processed by the NGINX Gateway API pod, while Antrea handles the networking stack, integrating with NGINX to manage traffic routing.

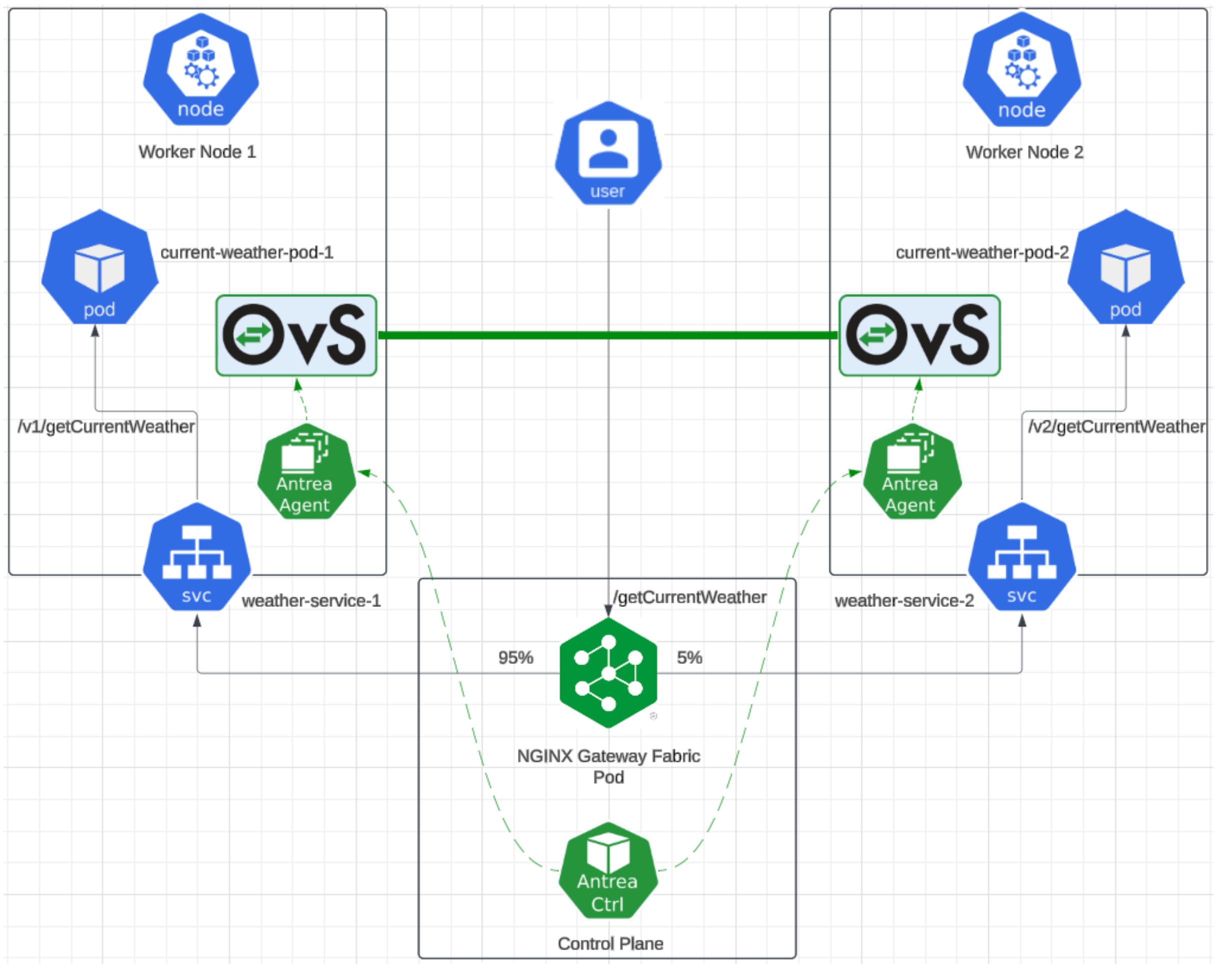


Figure 3.4: Example Kubernetes cluster with Antrea CNI and NGINX Gateway Fabric in canary stage of canary deployment [3][9].

Cilium

In Figure 3.5, an example canary cluster stack using the Cilium plugin is presented. The arrows represent the real data traffic flow, while the dashed arrows indicate the configuration flow. The HTTPRoute resources are pulled by the cilium-agent, which prepares the configuration and injects it into the Envoy proxy and eBPF. When Envoy receives an incoming request, it understands the traffic-splitting ratio and decides whether to route the traffic to a local pod or to a different pod, as specified in the HTTPRoute configuration [8]. The HTTPRoute resource for the Cilium Gateway API will look almost exactly the same as in Listing 4, with the only difference being the parentRefs name, which defines which Gateway is being used.

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
    name: current-weather-route
spec:
    parentRefs:
    - name: nginx-gw
    rules:
    - matches:
        - path:
            type: PathPrefix
            value: /getCurrentWeather
        backendRefs:
        - kind: Service
            name: current-weather-pod-1
            port: 8080
            weight: 95
        - kind: Service
            name: current-weather-pod-2
            port: 8090
            weight: 5
```

Listing 4: Egress resource example [9].

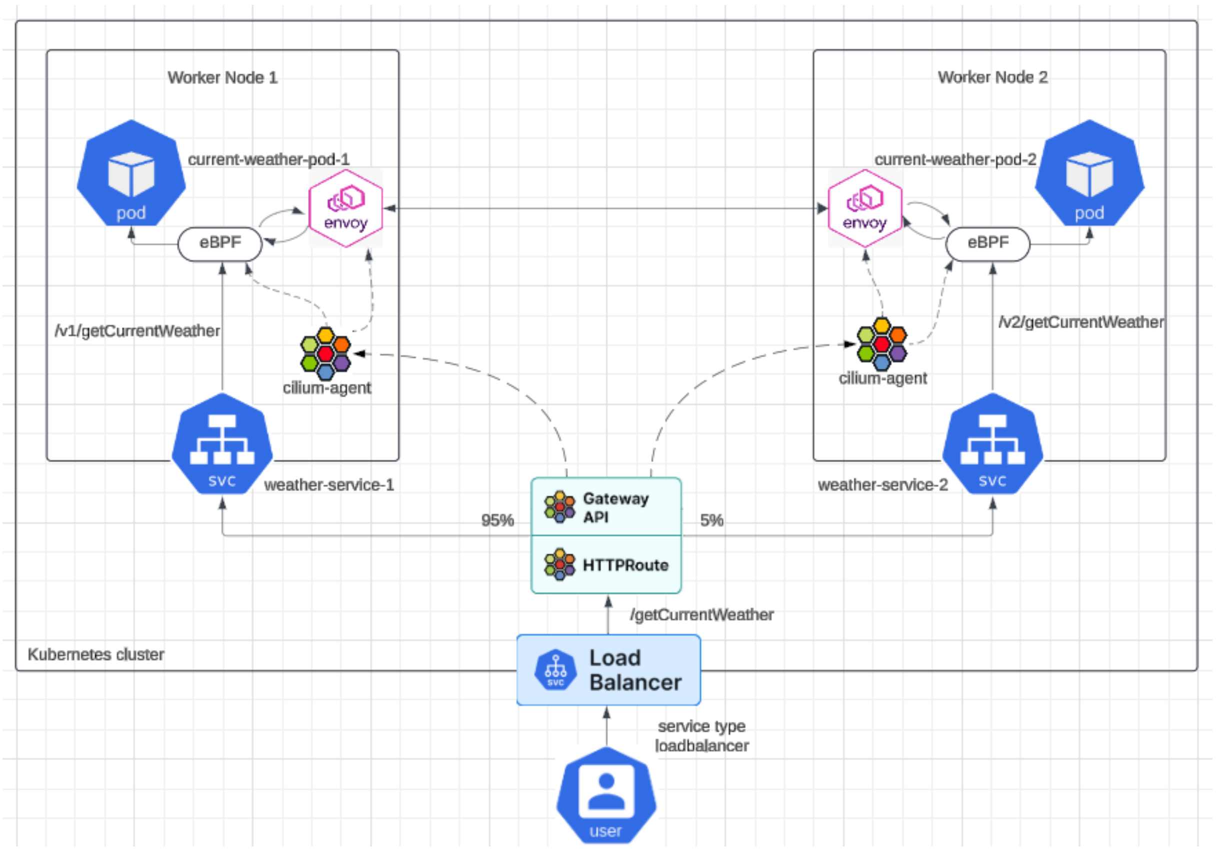


Figure 3.5: Example Kubernetes cluster with Cilium CNI in canary stage of canary deployment [19].

4. Implementing Egress and Ingress Scenarios Using Selected CNI Plugins

This chapter presents the implementation of both egress and ingress scenarios. The egress scenario will be executed locally, while the ingress scenario will be deployed both on local infrastructure (personal laptop) and in the public cloud (Azure). The tools used in these implementations, along with example configurations, will be described.

The Kubernetes cluster will run on a local laptop with the following specifications:

- CPU: AMD Ryzen 5 3500U 8 CPUs
- RAM: 20 GB
- Storage: 256 GB SSD
- Operating System: Fedora 40 with kernel 6.10.11

The cloud infrastructure consists of two AKS nodes (virtual machines) of type Azure Standard_A2_v2. Each VM has the following specifications:

- CPU: 2 vCPUs
- RAM: 4 GB
- Storage: Standard SSD
- Operating System: Ubuntu 22.04

4.1. Tools and automation

In this section, the tools used to provision the egress and ingress implementations will be described. A Kubernetes cluster will be created to simulate the scenarios, with Terraform, an

Infrastructure as Code (IaC) tool, used to provision and interact with the cluster. Additionally, Ansible will be used to create and configure the cluster setup, as well as to run Terraform and performance tools.

Ansible

Ansible is an open-source tool that automates the provisioning and configuration of infrastructure. Configuration in Ansible is written in playbooks, which are YAML files that serve as blueprints containing a set of instructions to be executed. Each playbook consists of one or more plays, and each play describes a set of tasks to be performed on a group of targeted hosts [22].

```yaml
- name: Create openstack instance and assign floating ip
  hosts: "{{ openstack_pool | default('localhost') }}"
  var_files:
    - ./vars/auth.yml
  become: yes

  tasks:
    - name: Create the OpenStack instance
      openstack.cloud.server:
        state: present
        name: " {{ inventory_hostname }}"
        key_name: "{{ key_name }}"
        network: "{{ network_name }}"
        auth:
          auth_url: "{{ auth_url }}"
          username: "{{ username }}"
          password: "{{ password }}"
          project_name: "{{ project_name }}"

  roles:
    - assign_floating_ip

```

Listing 5: Example Ansible playbook [22].

```ini
[openstack_pool]
instance-1.example.com key_name=ansible_key network_name=my-network ansible_host=10.10.10.10
instance-2.example.com key_name=ansible_key network_name=my-network ansible_host=10.10.10.20
```

Listing 6: Example Ansible inventory [22].

Listing 5 shows an example Ansible playbook configuration. The hosts field defines the group of objects on which the configuration script will be executed. In this case, instances specified in the openstack_pool group in the Ansible inventory, shown in Listing 6, will be created when the playbook is run. The vars_files option allows for attaching files that contain variables, such as authentication credentials required to access the OpenStack cloud. The become directive is used to execute the script as the root user. The tasks and roles sections define the actual tasks. These can be specified directly in the tasks section or through roles. In this example, the script is referenced from ./roles/assign_floating_ip/tasks/main.yaml [22].

Iperf3

Iperf3 is a tool used for measuring network performance metrics. It supports TCP, UDP, and SCTP protocols in both IPv4 and IPv6 networks. The tool operates on a client-server architecture, making it ideal for evaluating throughput and round-trip time. In this scenario, Iperf3 will be used to assess the network performance in the egress setup.

Kind

Kind (Kubernetes in Docker) is a tool for creating local Kubernetes clusters. It simulates real communication between nodes within a single machine by using Docker containers to create control plane and worker nodes. This setup allows for node-to-node communication in a local environment. One important limitation of Kind is that it does not provide a load balancer for assigning external IP addresses to Kubernetes services. In the ingress scenario, where the Gateway API requires an externally routable IP address, MetalLB will be installed and configured on the local infrastructure to provide the necessary external IPs [23].

Grafana K6

Grafana K6 is an open-source load testing tool designed to simulate virtual users interacting with specified endpoints. The test configurations are written in JavaScript using the k6 library, allowing for the creation of detailed performance tests to evaluate system behavior under load. MetalLB

MetalLB

MetalLB is a load balancer implementation for Kubernetes on bare metal environments. Since Kind does not provide a built-in load balancer, services of type "LoadBalancer" would remain in a "pending" state without an external solution like MetalLB. In the ingress scenario, it is essential to deploy a load balancing service, such as MetalLB, to allocate an external IP address for the Gateway API [24].

Node Exporter

Node Exporter is a tool that exposes the current system's metrics, including CPU usage, memory utilization, disk I/O, and network statistics. These metrics are presented in the OpenMetrics format and can be accessed via the /metrics endpoint [25].

Prometheus

Prometheus is a monitoring and alerting toolkit that stores data in time-series format, associating each data point with the exact time it was collected. Prometheus retrieves data by pulling from specified endpoints based on its configuration. The collected data can be queried using PromQL, Prometheus's query language, allowing users to analyze and visualize system performance [26].

Terraform

Terraform is an open-source Infrastructure as Code (IaC) tool that enables the provisioning and management of a wide range of infrastructure resources, including cloud infrastructure, Kubernetes clusters, virtual machines, Docker containers, storage, and SaaS features. Configuration files are written in HashiCorp Configuration Language (HCL), which is a declarative language designed for describing infrastructure [27].

The Terraform workflow is made up of three stages [27]:

1. Write – Define the resources to be created in a configuration file.
2. Plan – Preview the resources that will be created based on the provided configuration and check for any errors in the code.
3. Apply – Provision the resources or apply changes to the infrastructure as defined in the write stage.

4.2. Egress scenario implementation

The egress scenario compares the performance of Antrea and Cilium egress gateway implementations. The test uses Iperf3 in TCP mode to measure network performance. The Iperf3 client is deployed inside a pod within the Kubernetes cluster, while the Iperf3 server runs on a personal computer that launches the cluster. Network performance metrics are collected by

Iperf3, while CPU and memory usage are monitored by a Node Exporter. The metrics being monitored include:

- CPU – The processing power utilized by the cluster.
- Memory – The amount of RAM consumed by the infrastructure.
- Throughput – The volume of data successfully transmitted per unit of time.
- RTT (Round-Trip Time) – The time taken for a data packet to travel to its destination and back.

The overall resource utilization and performance will be evaluated in four test cases:

1. antrea egress gateway – All traffic generated by a pod inside the cluster will leave the cluster through the Antrea egress gateway node implementation.
2. antrea base – Traffic will leave the cluster using the node on which the pod is deployed, without egress gateway involvement.
3. cilium egress gateway – The Cilium implementation of an egress gateway will route outbound traffic to the outside.
4. cilium base – No redirection of traffic, it will leave the cluster directly from the node where the pod is running.

In this part of the scenario, Antrea CNI is installed on a locally hosted Kubernetes cluster using Kind. An Ansible playbook automates the process by creating the cluster, installing the CNI, deploying the egress gateway, and running the test. The script used for this setup is shown below in listing 7:

```
- name: Create antrea egress scenario with egress gateway
  hosts: "{{ target | default('localhost') }}"
  vars_files:
    - ./vars/antrea.yml
    - ./vars/common.yml
    - ./vars/egress_gateway.yml
    - ./vars/local.yml

  roles:
    - create_kind_cluster
    - install_antrea
    - wait_until_antrea_installed
    - get_ip_for_egress_node
    - deploy_antrea_egress_gateway
    - monitoring
    - terraform_run_egress_iperf
    - scrap_prometheus_data
```

Listing 7: Ansible playbook used to deploy Antrea with Egress Gateway [22].

Four file containing variables included in the script:

1. common.yml – Contains shared variables, such as Ansible become password to access root privileges on machine.
2. egress_gateway.yml – Scenario name or node name on which the gateway is deployed.
3. antrea.yml – CNI name for later use, such as specifying the cluster name and the folder path where test results are stored.
4. local.yml – Information about local infrastructure, such as node names, the job name for Prometheus and the environment type

The actual playbook from listing 7 consists of eight steps to automate infrastructure provisioning, run test and store the output:

1. create_kind_cluster – Creates Kind cluster using config YAML.
2. install_antrea – Applies the Antrea YAML containing custom resource definitions that define cluster networking.
3. wait_until_antrea_installed – Uses the kubectl wait command and pauses the script execution until Antrea controller deployment is available.

4. get_ip_for_egress_node – Retrieves the IP address of the node by name specified in the variable files, on which the egress gateway will be deployed in the next step .

5. deploy_antrea_egress_gateway – Enables egress support in Antrea CNI using config map and creates static egress gateway by setting egressIP field to previously obtained IP address.

6. monitoring – Applies monitoring in cluster, deploying the Prometheus Deployment and Node Exporter DaemonSet.

7. terraform_run_egress_iperf – Runs the Iperf3 server on laptop, saves current timestamp and runs the Iperf3 client pod using Terraform.

8. scrap_prometheus_data – This part of playbook is responsible for pulling CPU and memory metrics stored it Prometheus database.

The playbook from listing 7 produces following infrastructure:

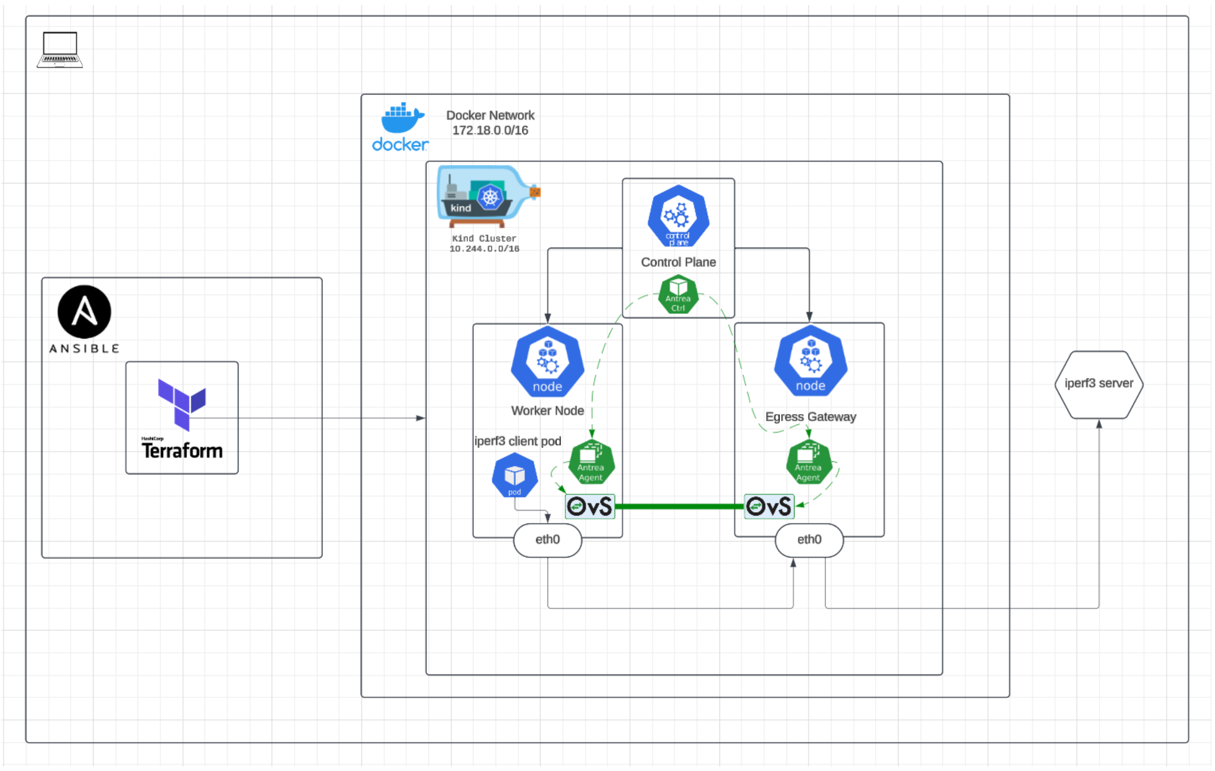


Figure 4.1: Antrea Egress Scenario infrastructure [9][21].

The Kubernetes cluster, as shown in Figure 4.1, is provisioned on a personal computer using Kind and configured with Ansible and Terraform. The cluster consists of three nodes: a control plane, a worker node, and an egress gateway, all of which are part of the Docker network. When the role terraform_run_egress_iperf starts execution and the created Iperf3 pod is ready, the test begins. The client, using the TCP protocol, sends data to the server (outside the cluster), which is routed through the egress gateway. The Iperf3 server detects that the traffic is coming from the egress gateway node (it sees the IP address of the egress gateway node as the source), because the traffic from the pod is SNATed.The Iperf3 client sends data packets using the TCP protocol, and after receiving acknowledgment from the server, the networking metrics are stored in memory. At the end of the test, a JSON file containing the gathered data is saved inside the pod. This file is saved on a volume shared with the host (personal laptop), ensuring that the data remains accessible after the test ends and the pod is terminated. Meanwhile, the Node Exporter continuously scrapes the metrics, which are pulled by Prometheus into its database. After the measurement is complete, the unformatted data is downloaded from Prometheus and saved as a CSV file.

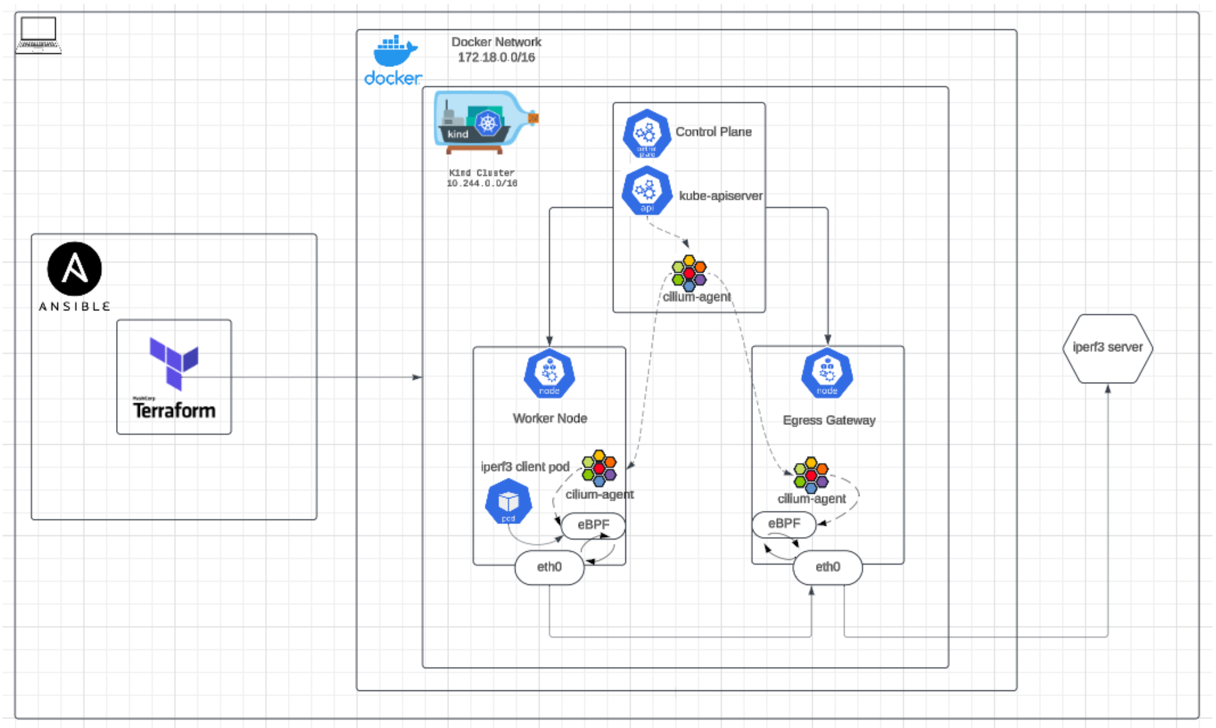


Figure 4.2: Cilium Egress Scenario infrastructure [8].

The playbook for Cilium is similar to the one used to create the egress scenario with the Antrea CNI. The Ansible roles are designed to be reusable across different playbooks. The only differences are the CNI installation, the deployment of the Egress Gateway, and in Cilium, the desired node is labeled with egress-node=true [8]. The infrastructure can be seen in Figure 4.2. The Iperf3 pod starts generating traffic, which is encapsulated and sent to the egress gateway.

The gateway performs IP address translation and forwards the data to the Iperf3 server outside the cluster.

4.3. Ingress scenario implementation

The ingress scenario evaluates the CPU and memory usage of the Antrea and Cilium Container Networking Plugins while using the Gateway API to handle weighted traffic routing. The experiment involves using the Gateway API to route 40% of incoming requests to one pod and evaluating the accuracy of traffic splitting by two different Gateway APIs. The test setup includes the K6 load testing tool running outside the Kubernetes cluster, generating traffic towards the Gateway API. The simulated traffic has four intensity levels, determined by allocating different numbers of virtual users interacting with the Gateway API. These levels are one, ten, one hundred, and one thousand virtual users. The traffic is initiated using the K6 tool, which runs inside a container on a personal computer. The network generator performs HTTP requests to the Gateway API and saves the received responses, extracting the pod name to a text file (one per line). By compiling a list of responses containing the names of two pods, the accuracy of the Gateway API traffic splitting is calculated. CPU and memory usage are monitored with a Node Exporter throughout the test and retrieved at the end of the scenario.

4.3.1. Cluster provisioning

Creating a local cluster using Kind is a straightforward process, as described earlier. However, when setting up a Kubernetes cluster on Azure, the azurerm Terraform provider must be properly configured to authenticate with an Azure account. The script shown in Listing 8 is responsible for creating the Kubernetes cluster in Azure Services. It is important to choose an appropriate location for the cluster, define the default node pool (including virtual machine type and node count), and remove the default CNI by setting the network_plugin in the network_profile to "none" if a different networking plugin is preferred [28].

4.3.2. Infrastructure

The process of creating an ingress scenario with Antrea CNI on Azure Kubernetes Services is fully automated showed in listing 9. The script provisions the infrastructure seen in Figure 4.3. The steps in the scripts are:

```
resource "azurerm_resource_group" "rg" {
  location = var.resource_group_location
  name     = "rg${var.common_infix}"
}

resource "azurerm_kubernetes_cluster" "k8s" {
  location            = azurerm_resource_group.rg.location
  name                = "cluster${var.common_infix}"
  resource_group_name = azurerm_resource_group.rg.name
  dns_prefix          = "dns${var.common_infix}"

  identity {
    type = "SystemAssigned"
  }

  default_node_pool {
    name       = "agentpool"
    vm_size    = var.vm_type
    node_count = var.node_count
  }

  linux_profile {
    admin_username = var.username

    ssh_key {
      key_data = azapi_resource_action.ssh_public_key_gen.output.publicKey
    }
  }

  network_profile {
    network_plugin    = "none"
    load_balancer_sku = "standard"
  }
}
```

Listing 8: Terraform Azure Kubernetes Service creation script [28].

1. create_azure_cluster – Runs Terraform to provision infrastructure in the cloud and configures the local environment to allow kubectl to interact with the cluster.

2. install_antrea – Installs Antrea CNI plugin.

3. wait_until_antrea_installed – Waits until Antrea is installed.

4. install_gateway_api_crd – Applies custom definition resources, Gateway, GatewayClass, HTTPRoutes etc.

5. install_nginx_gateway_fabric – Installs NGINX Gateway Fabric using Helm and waits until is ready.

6. deploy_antrea_ingress_scenario – Deploys the ingress scenario (echo pods and Gateway API) using Terraform.

7. monitoring – Enables monitoring with Node Exporter and Prometheus.

8. register_gateway_api_ip – Registers the IP address of the Gateway API for the K6 tool.

9. run_k6 – Creates a container with K6, which generates HTTP traffic accessing Gateway API.

10. scrap_prometheus_data – Downloads data about CPU and memory utilization.

```
- name: Create antrea ingres scenario with gateway api
  hosts: "{{ target | default('localhost') }}"
  vars_files:
    - ./vars/antrea.yml
    - ./vars/cloud.yml
    - ./vars/common.yml
    - ./vars/traffic_splitting.yml

  roles:
    - create_azure_cluster
    - install_antrea_cloud
    - wait_until_antrea_installed
    - install_gateway_crd
    - install_nginx_gateway
    - deploy_antrea_ingress_scenario
    - monitoring
    - register_gateway_api_ip
    - run_k6
    - scrap_prometheus_data
```

Listing 9: Ansible playbook used to deploy Antrea with Gateway API [22].

Figure 4.3 shows the cluster created using the script from Listing 9. In the cloud environment, the NGINX Gateway API pod is located on one of the nodes, which also runs the echo pod. Communication between the two pods within the node occurs via the Open vSwitch (OvS) bridge [9]. In the local infrastructure, the gateway is forced to be deployed on the control plane node, so it uses Node-to-Node communication when routing traffic. Since Cilium uses the Gateway API, which does not run in a pod, the resource utilization is distributed between both nodes to Envoy.

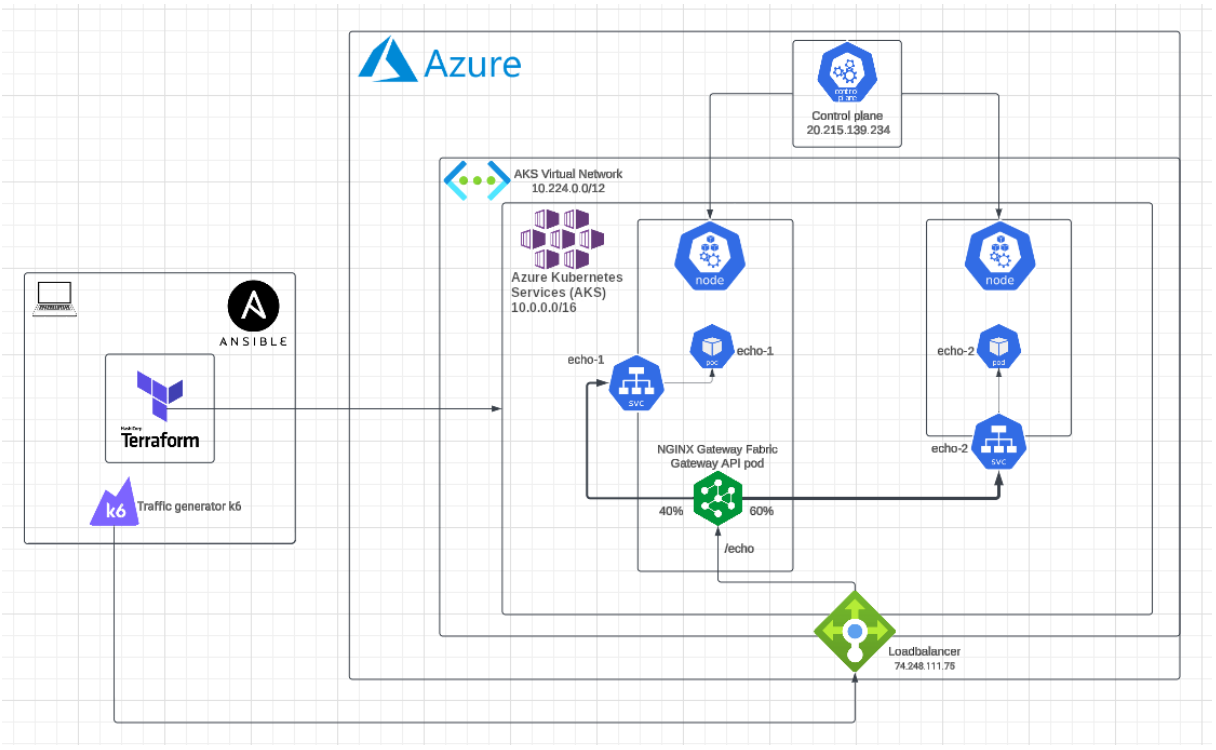


Figure 4.3: Antrea Ingress Scenario infrastructure [29][21].

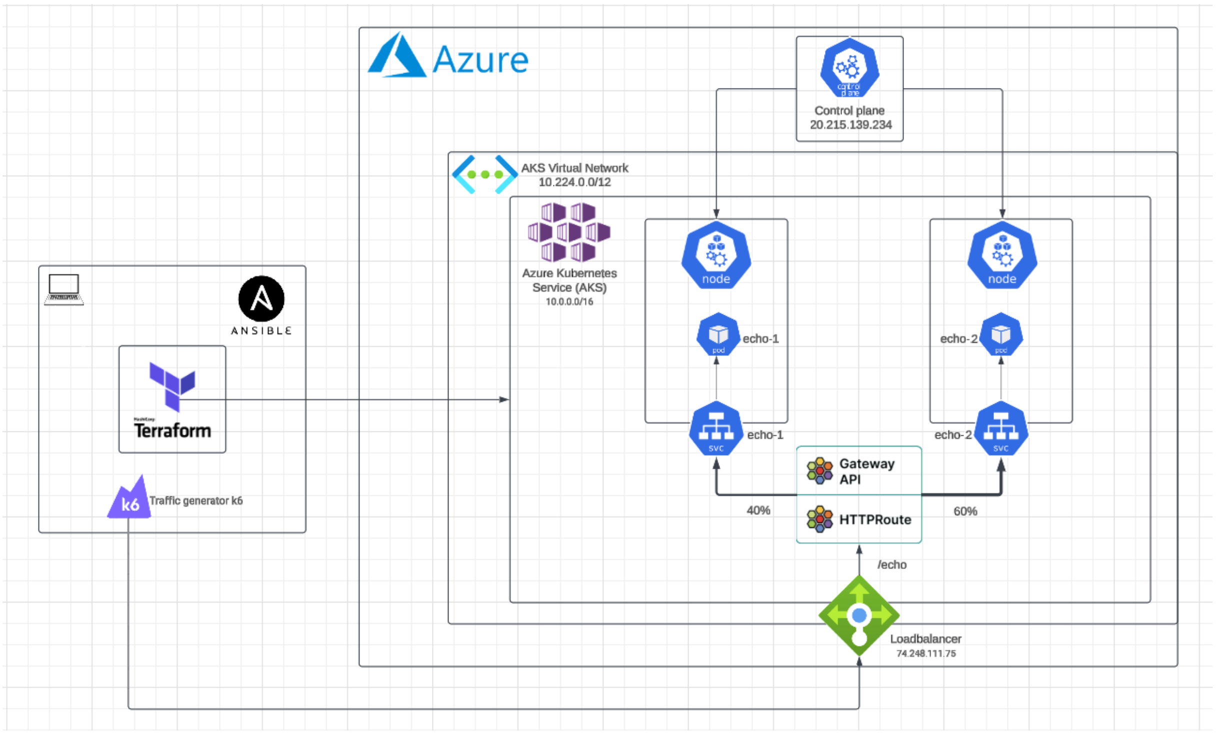


Figure 4.4: Cilium Ingress Scenario infrastructure [8][19].

Both test cases, shown in Figures 4.3 and 4.4, consist of two nodes, each running echo pods. The pods host an application designed to return their pod name when a request is made to the "/echo" endpoint. The Grafana K6 tool sends requests to the exposed Gateway API IP address and, in return, logs which pod responded to the request. This information is used to calculate the accuracy of traffic weighting in both configurations. The load balancer shown at the bottom of the figures exposes the Gateway API to a public IP address so that the K6 client, located on a personal computer, can perform the requests. It does not split the traffic; that is done by the Gateway API.

The ingress scenario using the Cilium CNI plugin does not require the installation of NGINX Gateway Fabric, as it utilizes its own built-in implementation.

4.3.3. The differences between cloud and local runs

Control plane

In the local environment, the control plane node is created in the same way as the worker nodes, running as a container. However, when using NGINX with the Antrea CNI, the NGINX Gateway Fabric pod is deployed on a separate control plane node. This setup allows the routing of traffic between two different nodes, ensuring that traffic always exits the node. In contrast, when using AKS (Azure Kubernetes Service), the control plane is not part of the node pool; it is a separate managed service outside the node pool, providing a clearer separation between the control plane and worker nodes. This architectural difference can affect the measurements (compared to the local stack), as resource utilization of the control plane node is not gathered.

Client traffic generator

In the cloud setup, the client running on a personal computer generates HTTP requests to the public Gateway API IP address exposed by Azure Cloud. Resource utilization within the cluster is exclusive to the cluster itself, not the client. However, when running the local stack, the client is part of the laptop on which the cluster is running, which could influence the measurements.

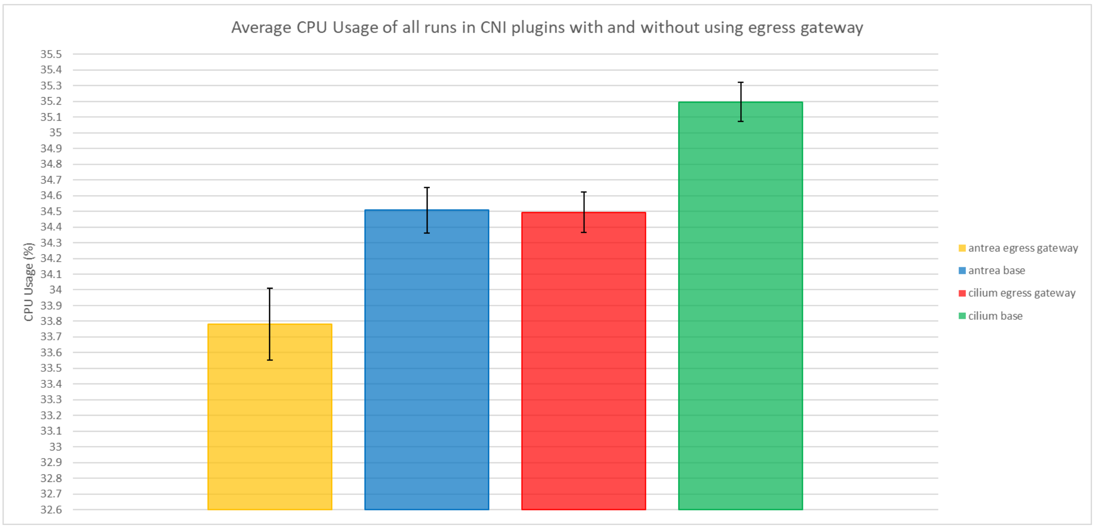
5. Performance comparison

In this chapter, the test results for both ingress and egress scenarios will be presented and analyzed, focusing on comparing CPU and memory usage in each case. For the ingress scenario, the analysis will cover weighted traffic splitting through the Gateway API, evaluating its impact on resource utilization and the accuracy of traffic distribution. The test was conducted sixteen times, each lasting only two minutes, because during traffic generation by a thousand virtual users, the logs reached up to half a million lines. For the egress scenario, the evaluation will include traffic routing through an egress gateway, comparing throughput and round-trip time. The egress test was conducted eight times, each lasting twelve minutes. The results will highlight differences in resource efficiency and networking performance between the Antrea and Cilium CNI plugins, identifying which CNI offers better optimization for these scenarios.

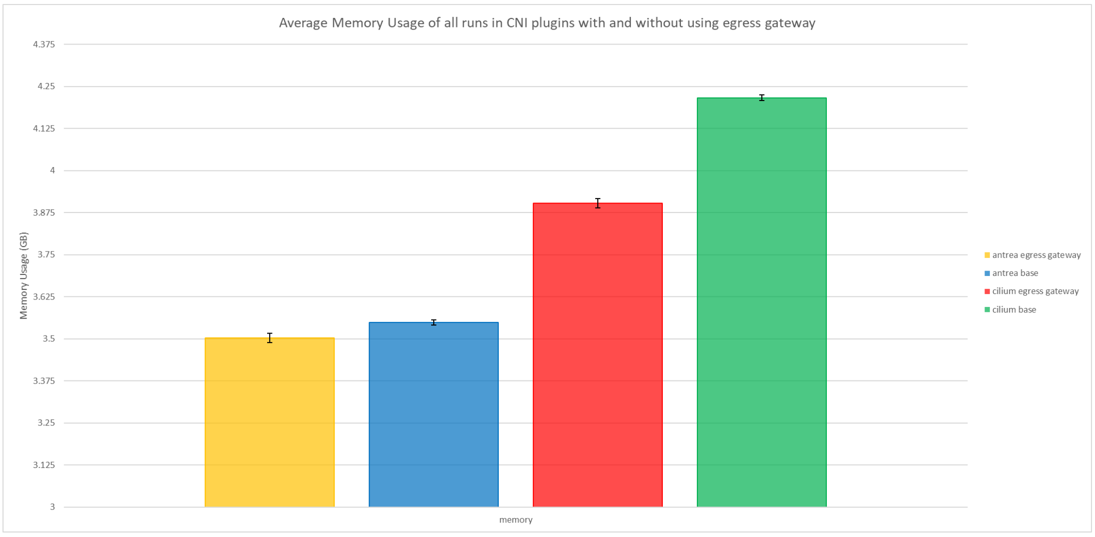
5.1. Egress Scenario

Resource consumption

Figures 5.1a and 5.1b show that resource utilization is lower when using an egress gateway compared to scenarios without redirecting outgoing packets via an additional node. This could be because all traffic management (routing table, NAT rules, and maps) is offloaded to the egress gateway. In this setup, the node running the Iperf3 pod does not need to evaluate routing decisions. Instead, it simply forwards all traffic to the egress gateway (where address translation occurs), allowing the node to allocate more CPU resources to the pod running the Iperf3 client. Without the egress gateway, the individual nodes are responsible for handling more operations, which can lead to higher resource usage. Figures 5.2a and 5.2b show the results of all eight runs across the four test cases. The sixth run in Figure 5.2a, during the Cilium egress gateway case, is an outlier and was excluded from the calculation of the total average (using a truncated mean). Overall, CPU usage is lower when the cluster uses Antrea networking. As for memory usage, Antrea consumes 10% less memory when using the egress gateway and up to almost 20% less when not using an additional node, compared to Cilium.

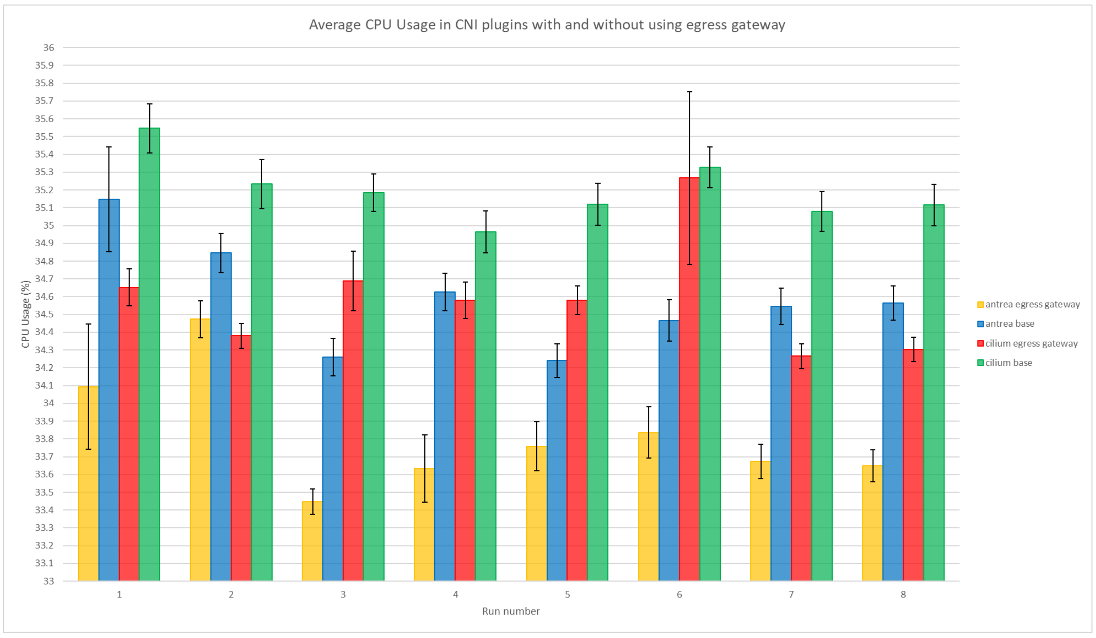


(a)

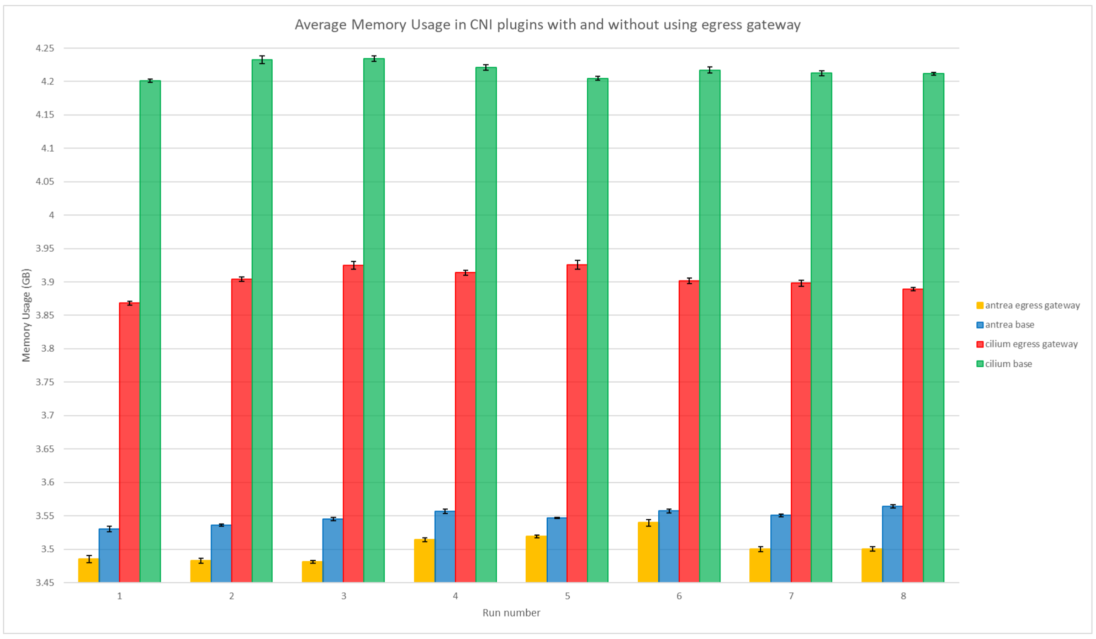


(b)

Figure 5.1: Average resource consumption in egress scenario, (a) CPU, (b) Memory



(a)



(b)

Figure 5.2: Average resource consumption in egress scenario in each run, (a) CPU, (b) Memory

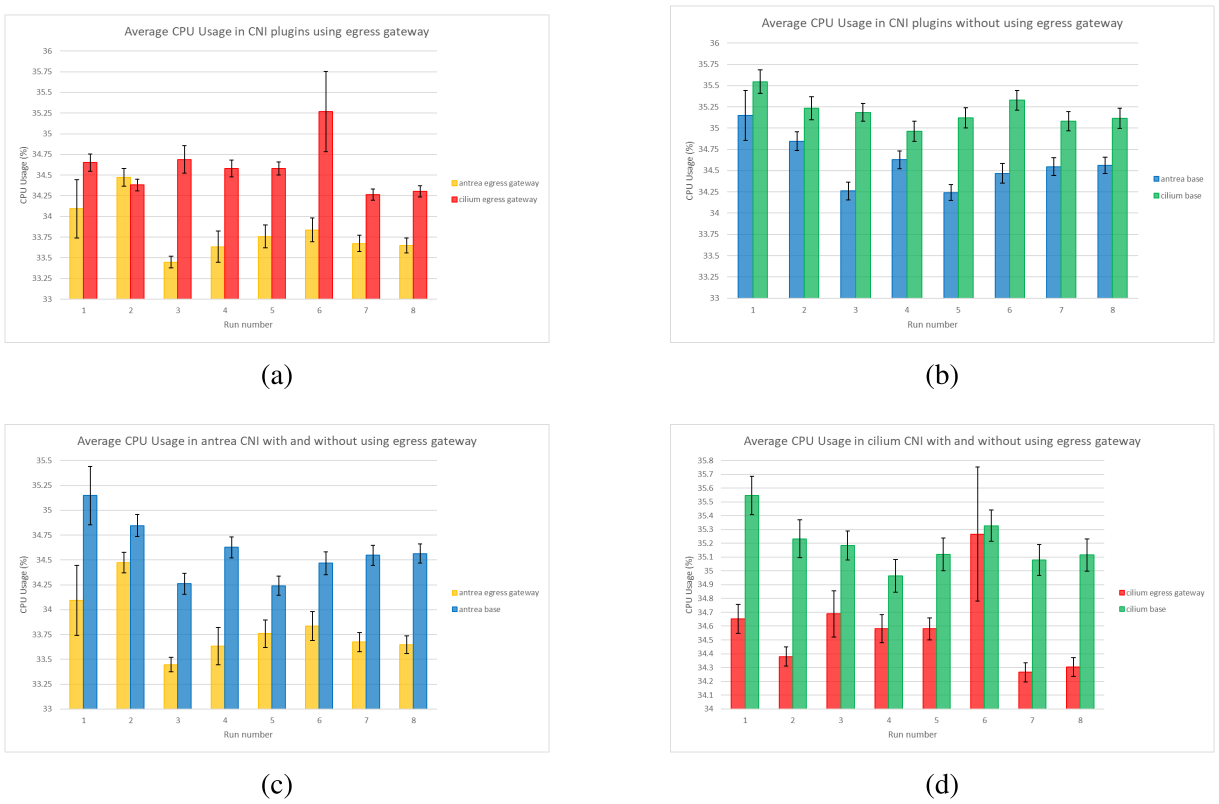
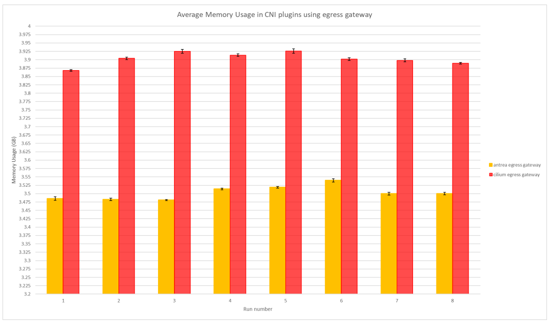
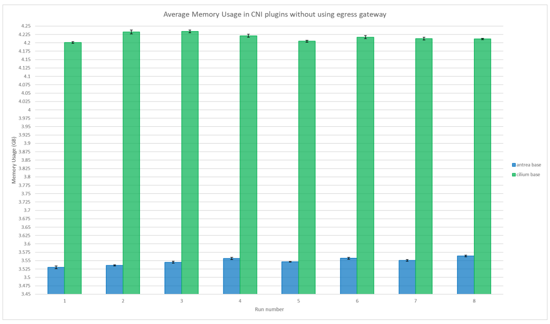


Figure 5.3: CPU usage in all four test cases.

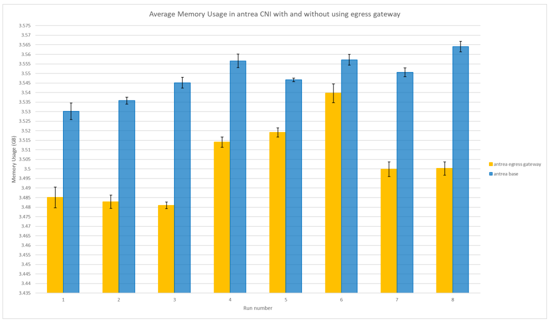
Figures 5.3a, 5.3b, 5.4a, and 5.4b compare the average CPU and memory usage between Cilium and Antrea CNI across all eight runs. These figures demonstrate how both container network interface plugins perform under the same test conditions, helping to evaluate which plugin consumes fewer resources. In only the second run did Antrea not use less CPU than Cilium. Figures 5.3c, 5.3d, 5.4c, and 5.4d highlight how using an egress gateway with each CNI plugin affects resource usage, compared to scenarios where an egress gateway is not used.



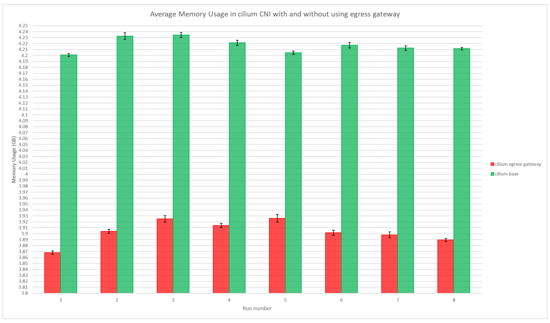
(a)



(b)



(c)

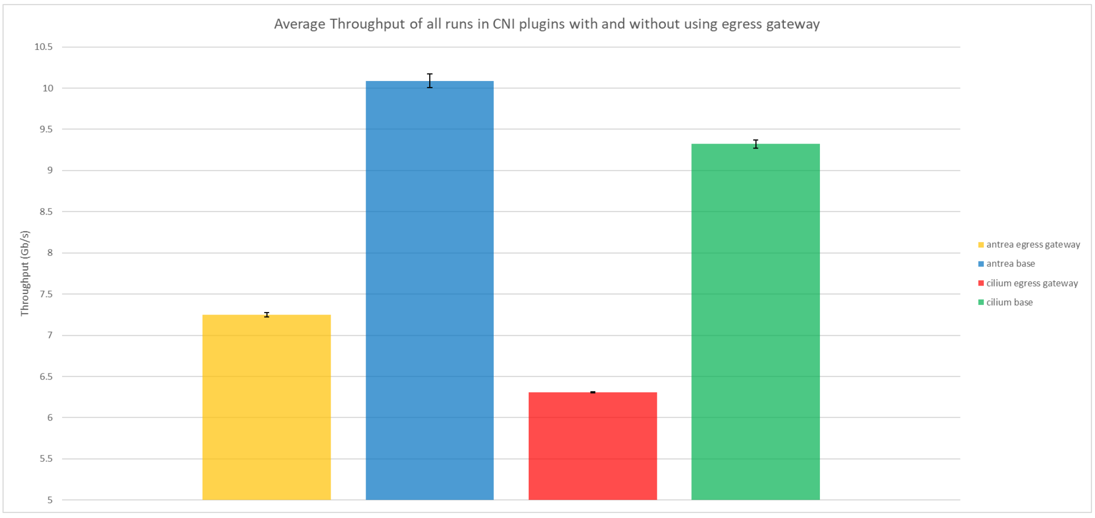


(d)

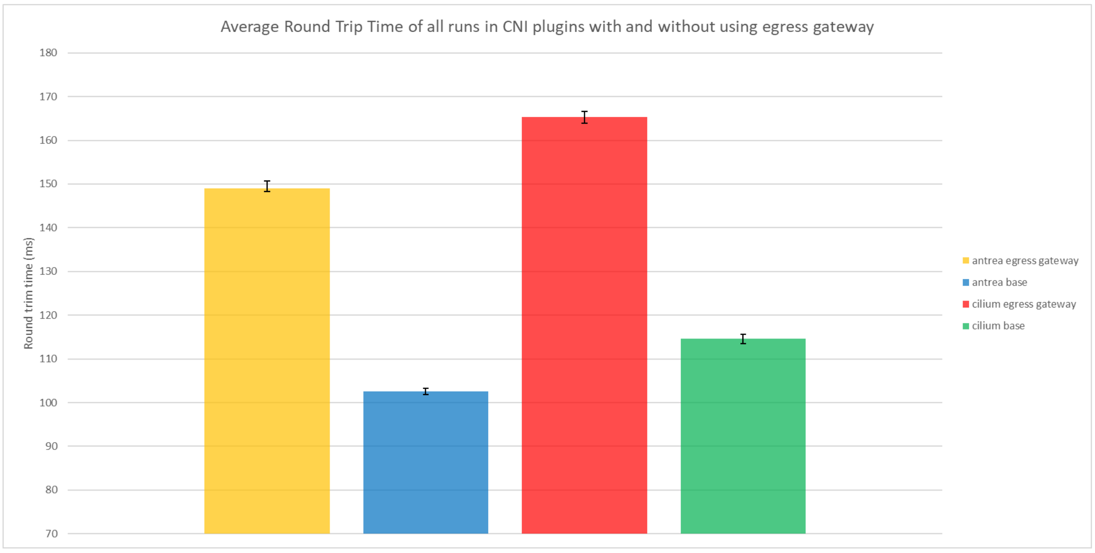
Figure 5.4: Memory usage in all four test cases.

Networking performance

Figures 5.5a and 5.5b provide an overall performance summary as the average of all runs. Antrea handles traffic more efficiently, with lower round-trip times, regardless of whether an egress gateway is used. The plots in 5.6a and 5.6b display the results of all eight runs for each test case, showing that Antrea outperforms Cilium in every single run, achieving higher throughput and lower bidirectional latency.

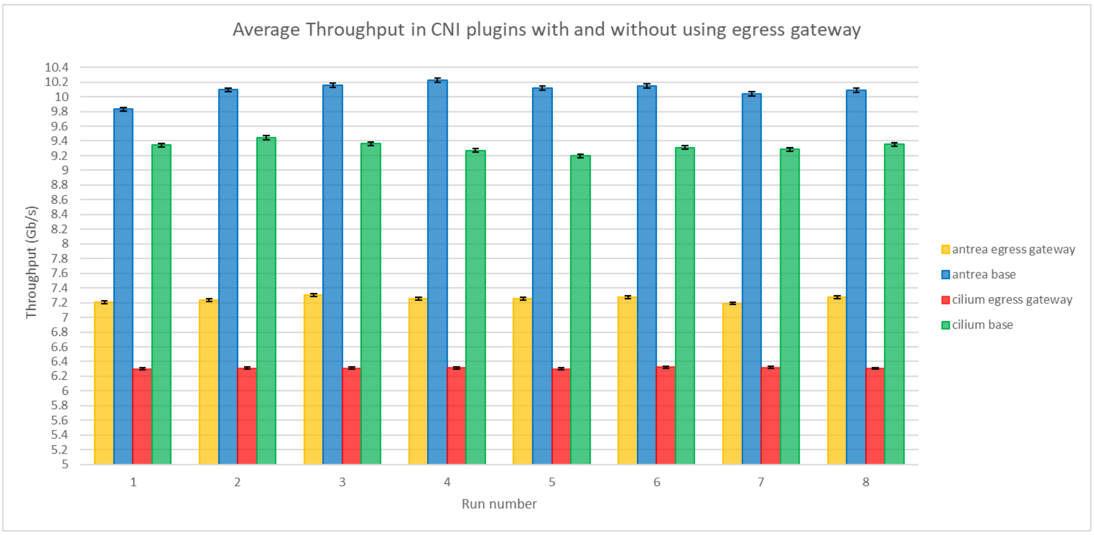


(a)

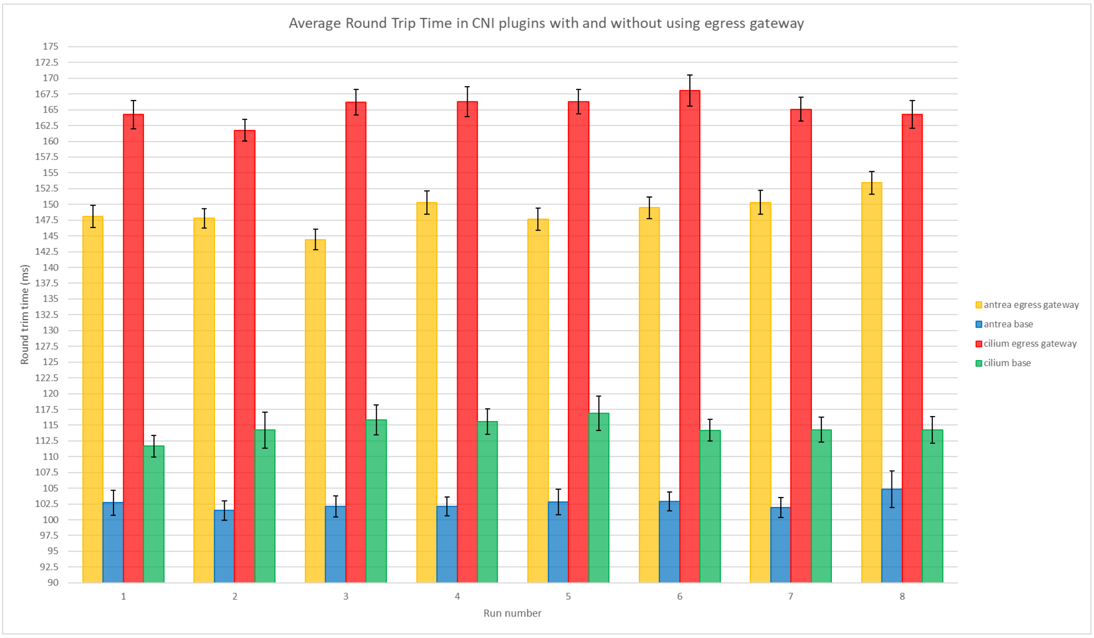


(b)

Figure 5.5: Average networking performance in egress scenario, (a) Throughput, (b) Round Trip Time



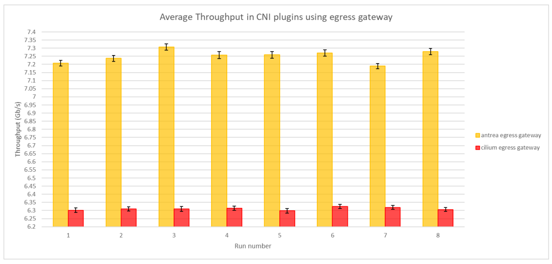
(a)



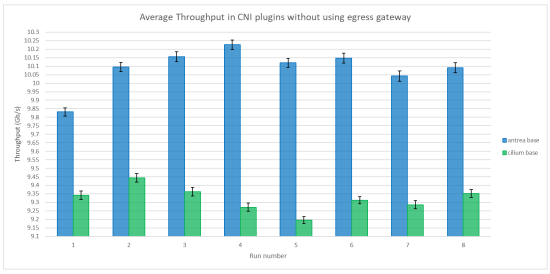
(b)

Figure 5.6: Average networking performance in egress scenario in each run, (a) Throughput, (b) Round Trip Time

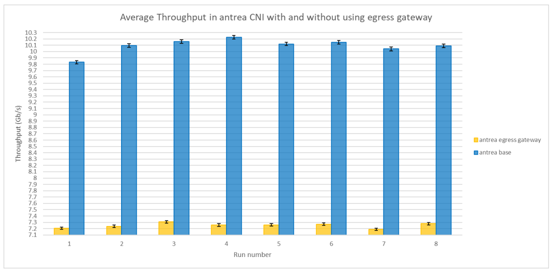
Figures 5.7a, 5.7b, 5.8a, and 5.8b illustrate the average throughput and round-trip time across eight runs, comparing the performance of the networking plugins. Throughput with the Cilium egress gateway remains stable, hovering around 6.3 Gb/s, with tighter confidence intervals indicating greater stability. In every run, Antrea demonstrates an advantage in both data transfer rate and RTT. Figures 5.7c, 5.7d, 5.8c, and 5.8d compare the performance of the CNI plugins with and without the use of an egress gateway. They highlight whether using an egress gateway is beneficial, showing significant differences in throughput and round-trip time.



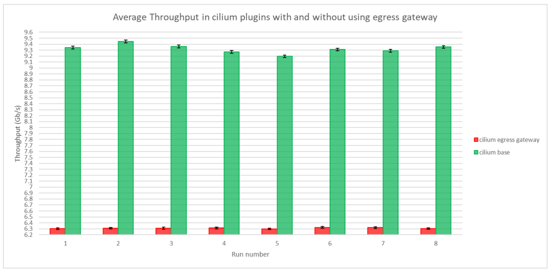
(a)



(b)

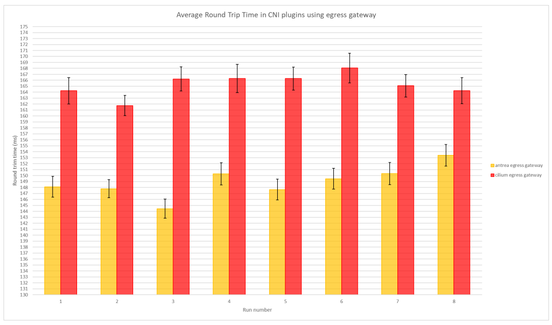


(c)

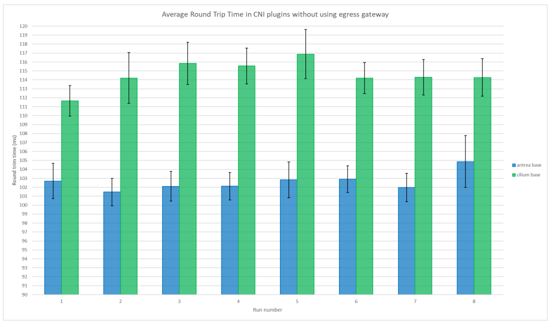


(d)

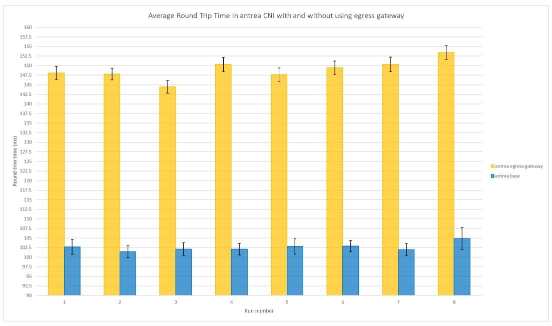
Figure 5.7: Throughput in all four test cases.



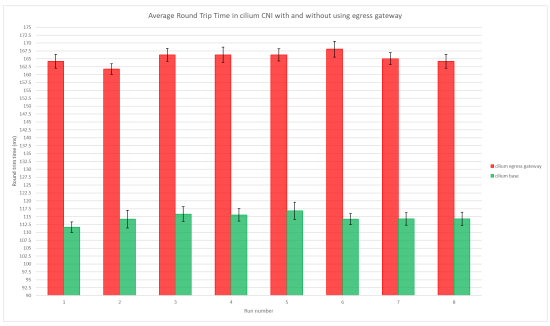
(a)



(b)



(c)



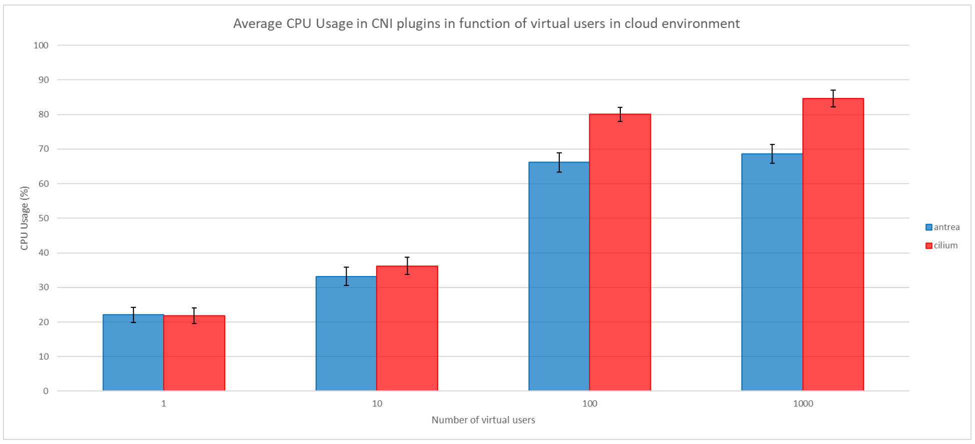
(d)

Figure 5.8: Round Trip Time in all four test cases.

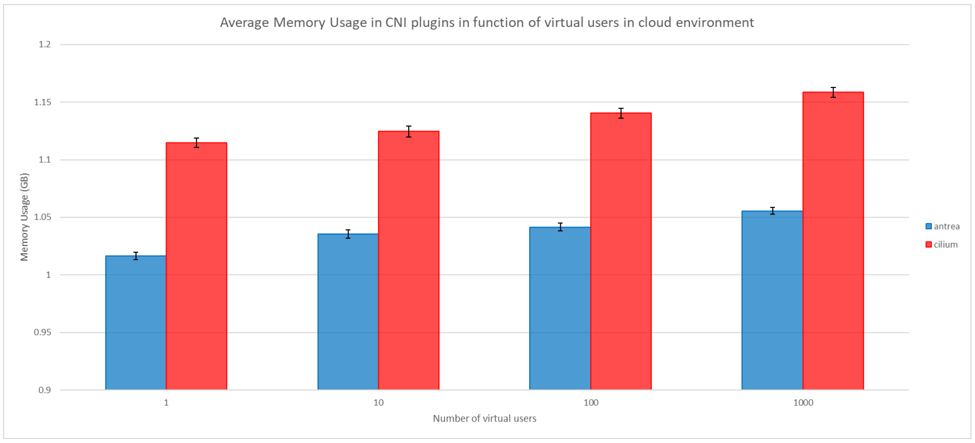
This performance advantage of Antrea could be attributed to the fact that Cilium uses eBPF, which was designed for large-scale clusters. Since the current cluster consists of only a few pods, eBPF features are not fully leveraged, and Antrea's networking simplicity results in better data rates.

5.2. Ingress Scenario

Resource consumption



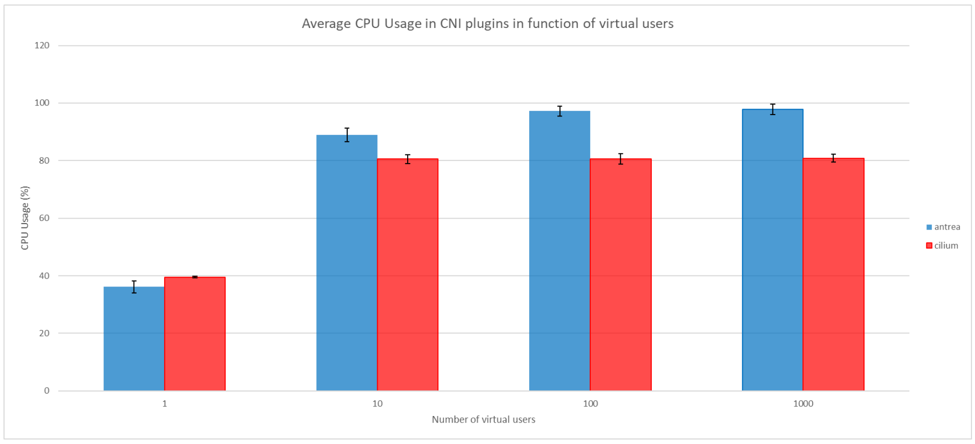
(a)



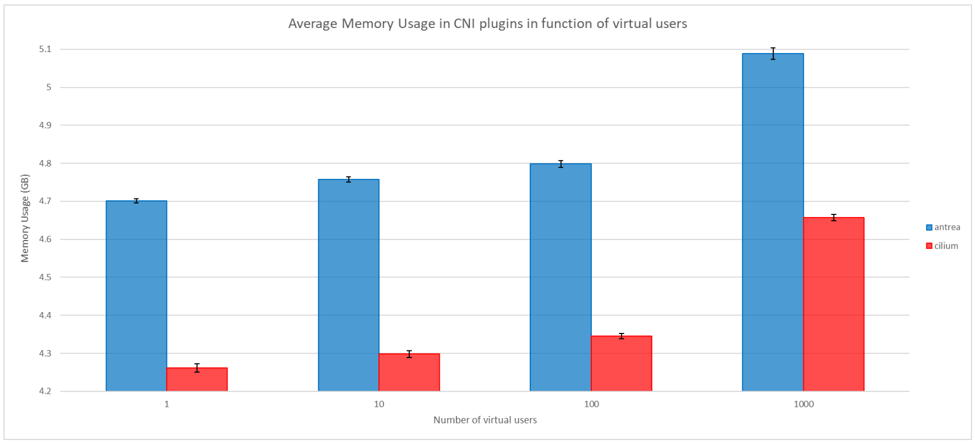
(b)

Figure 5.9: Average resource utilization in ingress scenario with increasing virtual users in cloud environment, (a) CPU, (b) Memory

Comparing Figures 5.9 and 5.10, it is evident that traffic management primarily impacts CPU usage rather than memory consumption, as memory usage shows egresser increases. In the cloud environment, Antrea demonstrates better resource efficiency, while in the local environment, Cilium consumes fewer resources. Antrea might use more CPU locally because there is an additional node, and the Gateway API NGINX pod is located on a different node (on the control plane) than the pod it forwards data to. This requires more resources to handle traffic across the cluster rather than within a single node.



(a)

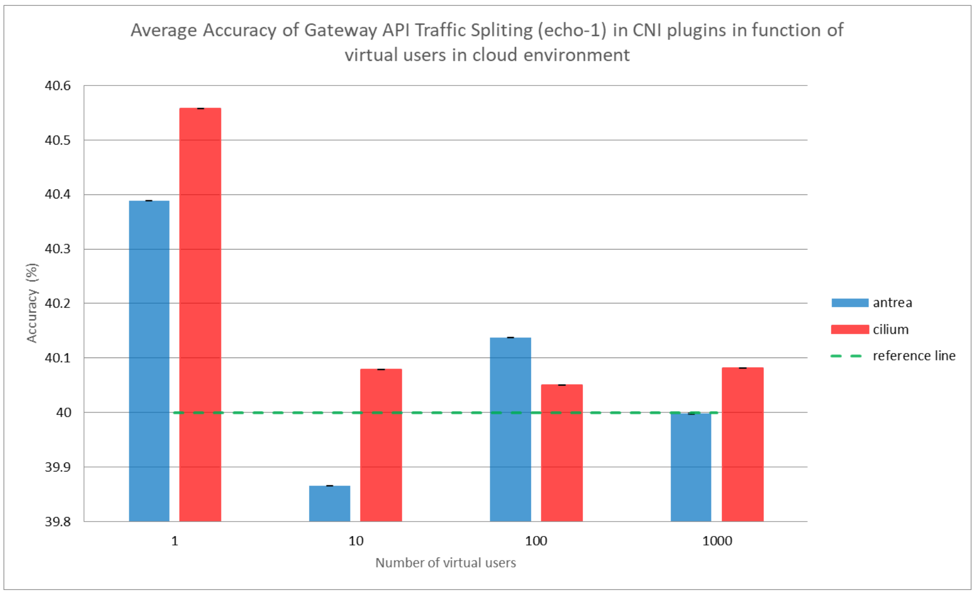


(b)

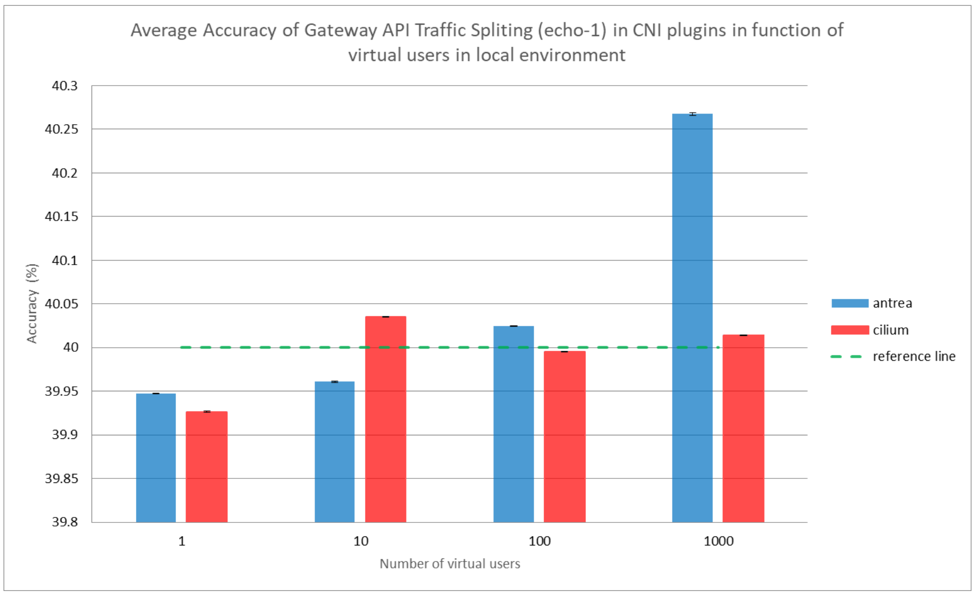
Figure 5.10: Average resource utilization in ingress scenario with increasing virtual users in local environment, (a) CPU, (b) Memory

Traffic splitting

When comparing traffic weighting accuracy in Figure 5.11 with an increasing number of virtual users in a cloud environment, Antrea proves to be more precise, both when a single user makes a request and when a thousand users generate massive load. In a local environment, Cilium is more accurate in splitting traffic to the value specified in the HTTPRoute in every scenario except when only a single user is involved. Antrea, in the local environment, shows wider confidence intervals, and the distance from the reference point is higher compared to Cilium, making it less effective during high-load periods.



(a)



(b)

Figure 5.11: Average traffic splitting accuracy in ingress scenario with increasing virtual users (a) Cloud, (b) Local

Analyzing the plots in Figure 5.12, we observe overlapping confidence intervals in all cases, indicating some variability in the results. However, when averaging the values, it becomes clear that Cilium performs better, particularly when the number of virtual users (VUs) is either one or a thousand. The distance to reference points is illustrated in Figure 5.13. Although the overlapping confidence intervals suggest statistical uncertainty, the average values show that Cilium achieves higher accuracy in traffic weighting compared to Antrea and the NGINX test case.

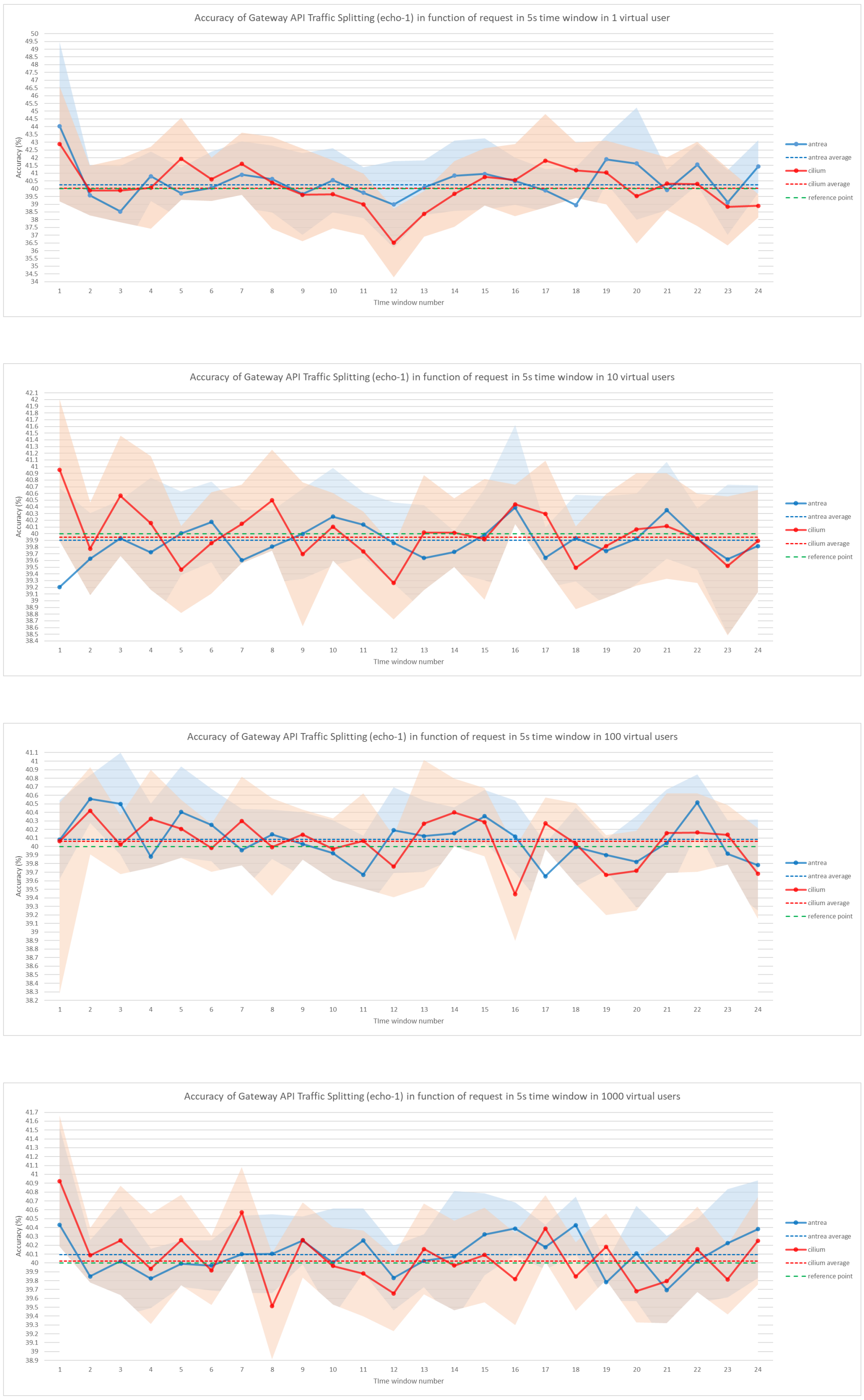
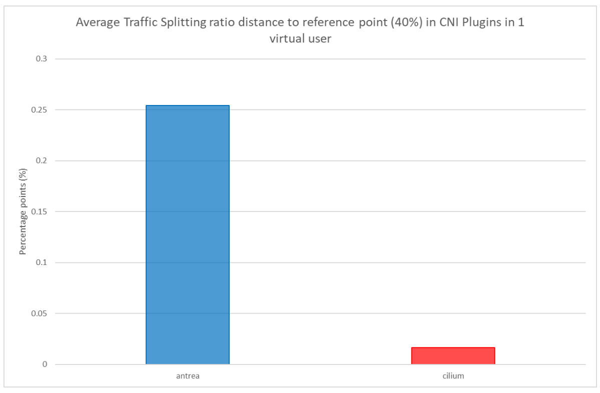
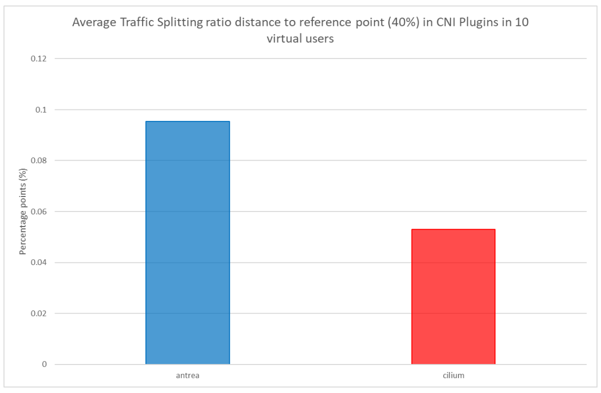


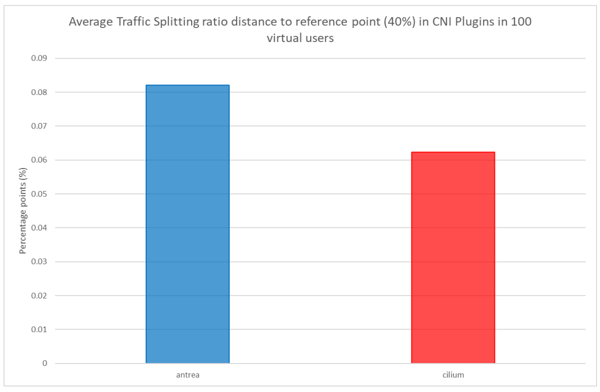
Figure 5.12: Average traffic splitting ratio in function of five second request time windows with increasing virtual users; one, ten, a hundred and a thousand



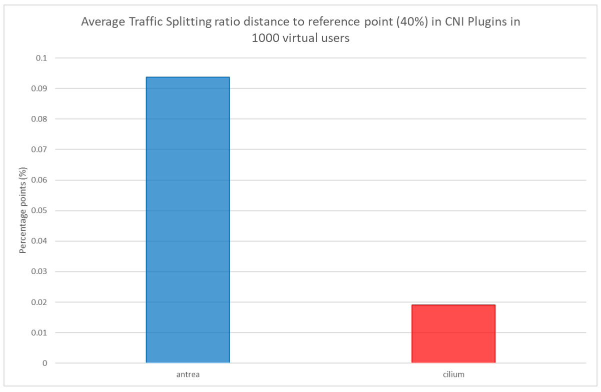
(a)



(b)



(c)



(d)

Figure 5.13: Average traffic splitting ratio distance to reference point in increasing virtual users, (a) one, (b) ten, (c) hundred, (d) thousand

6. Conclusions

In conclusion, the evaluation of both egress and ingress scenarios highlights the strengths of Antrea and Cilium CNI plugins. In the egress scenario, Antrea demonstrates lower resource usage, with higher throughput and lower round-trip time. However, Cilium's egress gateway implementation offers more control over routing outgoing traffic, with additional options for matching traffic to flow through the gateway. Both CNIs provide high availability for the egress gateway, though Cilium requires an enterprise version of the plugin. For the ingress scenario, Cilium consumes fewer resources in a local environment where Kubernetes nodes are created as Docker containers. On the other hand, Antrea, when combined with NGINX, exhibits lower resource usage in cloud environments. Cilium, however, achieves better traffic weighting accuracy (with some statistical uncertainty) compared to Antrea in both environments.

Overall, Cilium offers greater configurability and a robust set of features, including its own Gateway API and Egress Gateway implementation. Additionally, it supports eBPF, which can provide significant advantages in larger clusters. In contrast, the Antrea CNI plugin offers an efficient Egress Gateway with lower resource usage, making it a better option for smaller clusters with limited resources, where the need for both a Gateway API and an Egress Gateway provided by a single CNI may not be necessary.

Future work could focus on evaluating the performance and resource utilization of Antrea and Cilium in more complex and larger clusters. As noted by the authors of Cilium, eBPF offers greater efficiency in large-scale environments. Expanding future scenarios to include multiple Gateway APIs and egress gateways, with traffic generated by thousands or even tens of thousands of pods, would provide deeper insights into the performance of container networking plugins under highly demanding conditions.

Bibliography

[1] What is containerization? https://www.redhat.com/en/topics/cloud-native-apps/what-is-containerization. Accessed, 07-Dec-2024.

[2] https://docs.docker.com/get-started/docker-concepts/building-images/. Accessed, 07-Dec-2024.

[3] Kubernetes Documentation. https://kubernetes.io/docs/home/. Accessed, 07-Dec-2024.

[4] Data model. https://etcd.io/docs/v3.5/learning/data_model/. Accessed, 08-Dec-2024.

[5] Stephanie Susnjara and Ian Smalley. What is Kubernetes networking? https://www.ibm.com/topics/kubernetes-networking/. Accessed, 11-Dec-2024.

[6] Kubernetes Gateway API Documentation. https://gateway-api.sigs.k8s.io/. Accessed, 11-Dec-2024.

[7] Calico Documentation. https://docs.tigera.io/calico/. Accessed, 12-Dec-2024.

[8] Cilium Documentation. https://docs.cilium.io/en/stable/. Accessed, 12-Dec-2024.

[9] Antrea Documentation. https://antrea.io/docs/v2.2.0/. Accessed, 12-Dec-2024.

[10] Open Virtual Switch Documentation. https://www.openvswitch.org/. Accessed, 12-Dec-2024.

[11] eBPF Documentation. https://ebpf.io/what-is-ebpf/. Accessed, 12-Dec-2024.

[12] Jeremy Colvin. What is Kube-Proxy and why move from iptables to eBPF? https://isovalent.com/blog/post/why-replace-iptables-with-ebpf/. Accessed, 12-Dec-2024.

[13] Thomas Graf. CNI Benchmark: Understanding Cilium Network Performance. https://cilium.io/blog/2021/05/11/cni-benchmark/. Accessed, 12-Dec-2024.

[14] V. Dakić, J. Redžepagić, M. Bašić, and L. Žgrablić. Performance and latency efficiency evaluation of kubernetes container network interfaces for built-in and custom tuned profiles. *Electronics*, 13(3972), 2024.

[15] Shixiong Qi, Sameer G Kulkarni, and K. K. Ramakrishnan. Understanding container network interface plugins: Design considerations and performance. In *2020 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN*, pages 1–6, 2020.

[16] Shixiong Qi, Sameer G. Kulkarni, and K. K. Ramakrishnan. Assessing container network interface plugins: Functionality, performance, and scalability. *IEEE Transactions on Network and Service Management*, 18(1):656–671, 2021.

[17] David Soldani, Petrit Nahi, Hami Bour, Saber Jafarizadeh, Mohammed F. Soliman, Leonardo Di Giovanna, Francesco Monaco, Giuseppe Ognibene, and Fulvio Risso. ebpf: A new approach to cloud-native observability, networking and security for current (5g) and future mobile networks (6g and beyond). *IEEE Access*, 11:57174–57202, 2023.

[18] Kubernetes egress. https://www.tigera.io/blog/using-calico-egress-gateway-and-access-controls-to-secure-traffic/. Accessed, 15-Dec-2024.

[19] Egress Gateway. https://cilium.io/use-cases. Accessed, 15-Dec-2024.

[20] What Are Canary Deployments? Process and Visual Example. https://codefresh.io/learn/software-deployment/what-are-canary-deployments/. Accessed, 16-Dec-2024.

[21] Inc F5. Gateway architecture. https://docs.nginx.com/nginx-gateway-fabric/overview/gateway-architecture/. Accessed, 23-Dec-2024.

[22] Ansible Documentation. https://docs.ansible.com/ansible. Accessed, 17-Dec-2024.

[23] Kind Documentation. https://kind.sigs.k8s.io/. Accessed, 17-Dec-2024.

[24] Cloud Native Computing Foundation. MetalLB Documentation. https://metallb.io/. Accessed, 18-Dec-2024.

[25] Node Exporter Documentation. https://github.com/prometheus/node_exporter. Accessed, 18-Dec-2024.

[26] Prometheus Documentation. https://prometheus.io/docs/introduction/overview/. Accessed, 18-Dec-2024.

[27] What is Terraform? https://developer.hashicorp.com/terraform/intro. Accessed, 18-Dec-2024.

[28] Quickstart: Deploy an Azure Kubernetes Service (AKS) cluster using Teraform. https://learn.microsoft.com/en-us/azure/aks/learn/quick-kubernetes-deploy-terraform. Accessed, 18-Dec-2024.

[29] Grafana K6 Documentation. https://grafana.com/docs/k6/latest/. Accessed, 17-Dec-2024.

[30] LucidApp. https://lucid.app. Accessed, 23-Dec-2024.

[31] Microsoft Excel. https://www.microsoft.com/microsoft-365/excel. Accessed, 28-Dec-2024.

[32] https://cdn-icons-png.freepik.com/256/11454/11454089.png?semt=ais_hybrid. Accessed, 23-Dec-2024.

[33] https://e7.pngegg.com/pngimages/901/1/png-clipart-cloud-computing-content-delivery-network-cloud-storage-web-
png. Accessed, 23-Dec-2024.

[34] https://static-00.iconduck.com/assets.00/etcd-Icon-1024x990-d2wbvuqt.png. Accessed, 23-Dec-2024.

[35] https://diagrams.mingrammer.com/docs/nodes/k8s. Accessed, 23-Dec-2024.

[36] https://kind.sigs.k8s.io/logo/logo.png. Accessed, 23-Dec-2024.

[37] https://media.istockphoto.com/id/1332100919/vector/man-Icon-black-Icon-person-symbol.jpg?s=612x612&w=0&k=20&c=AVVJkvxQQCuBhawHrUhDRTCeNQ3Jgt0K1tXjJsFy1eg=. Accessed, 23-Dec-2024.

[38] https://s3-ap-southeast-2.amazonaws.com/content-prod-529546285894/2020/03/tf.png. Accessed, 23-Dec-2024.

[39] https://upload.wikimedia.org/wikipedia/commons/0/05/Ansible_Logo.png. Accessed, 23-Dec-2024.

[40] https://static-00.iconduck.com/assets.00/docker-Icon-2048x1753-uguk29a7.png. Accessed, 23-Dec-2024.

[41] https://learn.microsoft.com/en-us/azure/architecture/icons/. Accessed, 23-Dec-2024.

All figures in this document were created using Lucidchart for diagrams and Microsoft Excel for data visualization [30][31]. Icon sources [32][33][34][35][36][37][38][39][40][41].