Contents

1. Introduction

1.1. Purpose of the Thesis

1.2. Scope of the Thesis

1.3. Structure of the Thesis

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.

2.1.1. Containerization

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

- Portable and Flexible – container can be run on bare metal or virtual machine in cloud regardless of operating system. Only container runtime software like Docker Engine or containerd is required, which allows interacting with the host system.
- Lightweight – container is sharing operating system kernel with host machine, there is no need to install separate operating system inside
- Isolated – does not depend on host's environment or infrastructure
- Standarized – Open Container Initiative standardize 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. Consists of layers. 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].

The Kubernetes Authors says: "The name Kubernetes originates from Greek, meaning helmsman or pilot. K8s as an abbreviation results from counting the eight letters between the "K" and the "s"" [3]. K8s is an open-source orchestration platform capable of managing containers [3]. Key functionalities are [3]:

- Automated rollouts and rollbacks – updates or downgrades version of deployed containers at controller rate, replacing containers incrementally
- Automatic bin packing – allows specifying exact resources needed by container (CPU, Memory) to fit on appropriate node
- Batch execution – possible to create sets of tasks which can be run without manual intervention
- Designed for extensibility – permits adding features using custom resource definitions without changing source code
- Horizontal scaling – scales (replicate) app based of its need for resources
- IPv4/IPv6 dual-stack – allocates IPv4 or IPv6 to pods and services
- Secret and configuration management – allows store, manage and update secrets. Containers do not have to be rebuilt to access updated credentials
- Self-healing – restarts crashed containers or by failure specified by user
- Service discovery and load balancing – advertises a container using DNS name or IP and load balances traffic across all pods in deployment

– Storage orchestration – mounts desired storage like local or shipped by cloud provider and make 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 [4].

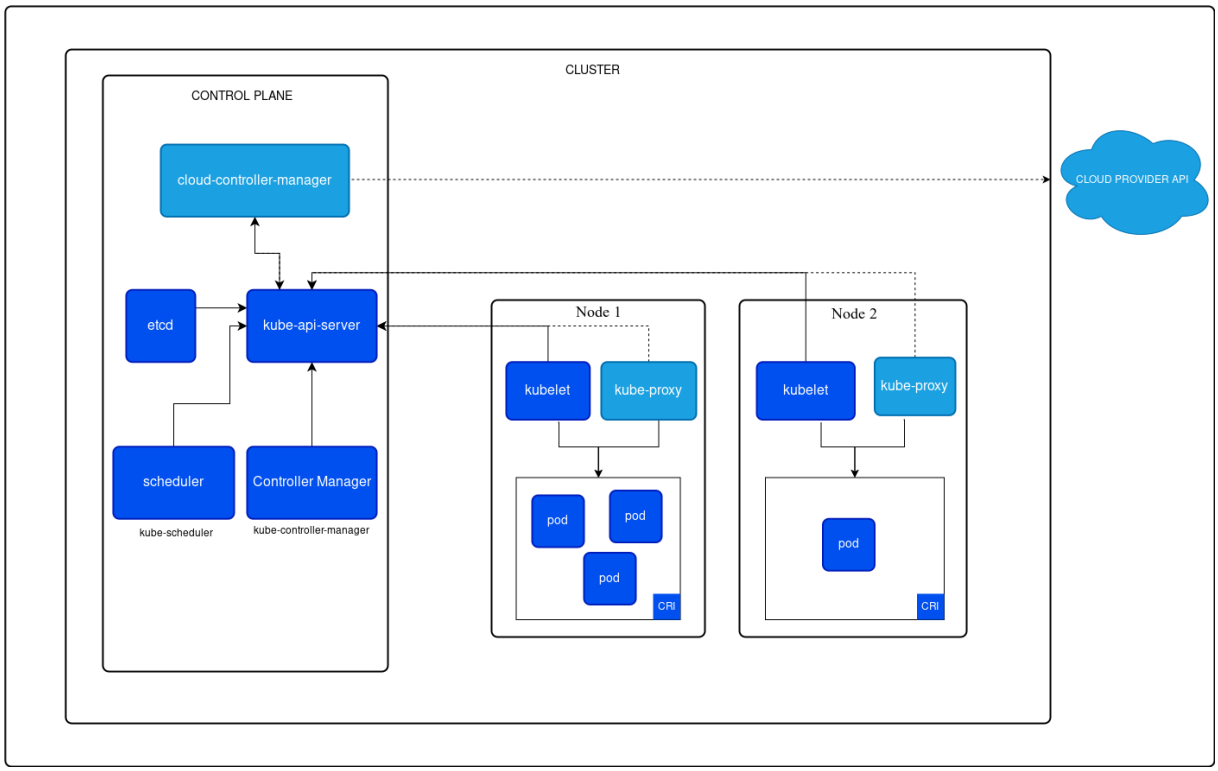


Figure 2.1: Kubernetes Cluster Architecture [4]

On 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 figure in node representation is *kube-proxy* component, which is not mandatory as some networking plugins can provide own

implementation of proxy [4]. 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, a smallest unit managed by K8s that has containers inside [4].

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 envirnoment this component is absent [4].

etcd

Etcd is an open-source distributed key-value store service often used in distribution systems. It is responsible for maintaining both the current state and its previous version in its persistent memory [4][5].

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 [4].

kube-controller-manager

Component which runs controller processes. Its compiled binary consists of multiple controllers. Example controllers are [4]:

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

More controller names can be found in Kubernetes source code.

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 [6].

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 [4].

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. [4].

kubelet

It is responsible for managing containers inside pod on its node. Uses Container Runtime Interface to communicate with containers [4] [7].

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 [8].

Pods

Pods are the smallest deployable objects in Kubernetes. It contains one or more containers, which can communicate with each other using localhost interface. Since they share IP addresses, they cannot use the same ports. It is useful when our service consists of two apps coupled to-

gether. For example, there is a pod which has two containers, one responsible for compiling a code, the second one is creating cache entry from compiled object and uploads to some data storage. It makes more sense, as sharing data among containers in a pod is easier than on node between pods. Scaling is simpler than replicating one pod instead of two. Moreover, communication between apps happens using localhost, in scenario where there are two pods with one container, ClusterIP *Service* is needed. However, the most common approach is to run one container per pod, where pod is just managing wrapper for containerized app. Also, rather than creating pod directly it is more common to use workload resource like *Deployment* [9].

ReplicaSet

Basically *ReplicaSet* consists of pod template and runs desired number of pods [10].

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 [11].

DaemonSet

Running pods using DaemonSet guarantee that every node will have a copy of 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 [12].

StatefulSet

StatefulSet unlike *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 pod on the same node as it was previously [13].

Job

Runs pod that does one task and exists. Kubernetes will retry execution if pod fails specific number of tries set in its configuration [14].

CronJob

Behaviors like *Job* but can regularly run every given time for tasks like database backups or log rotation [15].

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 [16].

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 [17]. There are four types of network communication: [17]:

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 [18].

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 [17]:

- *kubelet* or *cloud-controller-manager*, depending on local or cloud infrastructure allocates IP address for nodes
- *kube-apiserver* allocates IP address for services
- for allocation of IP address to pod is responsible 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* [19].

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 [19][20].

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 [21].

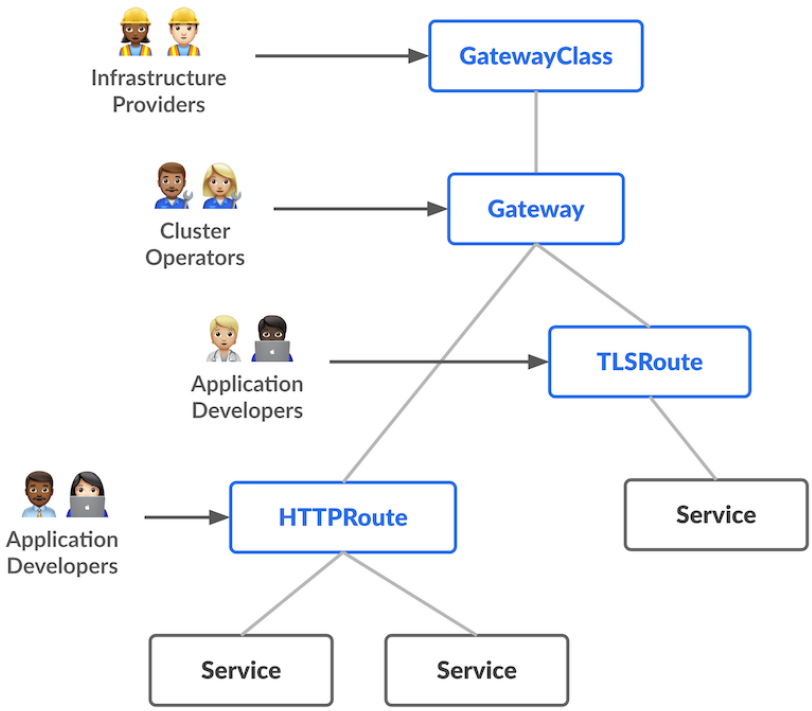


Figure 2.2: Gateway API roles-oriented resource model [21]

The model focuses on 3 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 [21]. The Gateway API creators provide a clear overview of what GatewayClass is: "This resource represents a class of Gateways that can be instantiated.". It defines specific types of load balancing implementations and provides clear explanation of capabilities available in Kubernetes resource model. The functionality is like *Ingress* [21] [22]. There can be more than one GatewayClasss created. [21].

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 [21] [23].

End users specified on Gateway API resource model on 2.2 figure 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. [21] [24].

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" [21].

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. [25] [26].

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 [18].

2.4.1. Overview of Selected CNI Plugins

Table 2.1: Comparison of Antrea and Cilium [27][28][26][29][30].

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 [27]. OvS is a virtual switch with capability of handling traffic flow between virtual machines and containers [31]. 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 [32]. 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 [33].

Cilium, open-source CNI which uses eBPF (extended Barkeley Packer Filter) for packet processing, security and deep observability using Hubble [34]. eBPF is a technology that allows running defined programs, with custom logic inside operating system kernel in privileged context without need of any kernel source code changes or loading modules. Lack of switching between kernel and user space, which reduces latency [35].

As seen on figure 2.3, 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 [36].

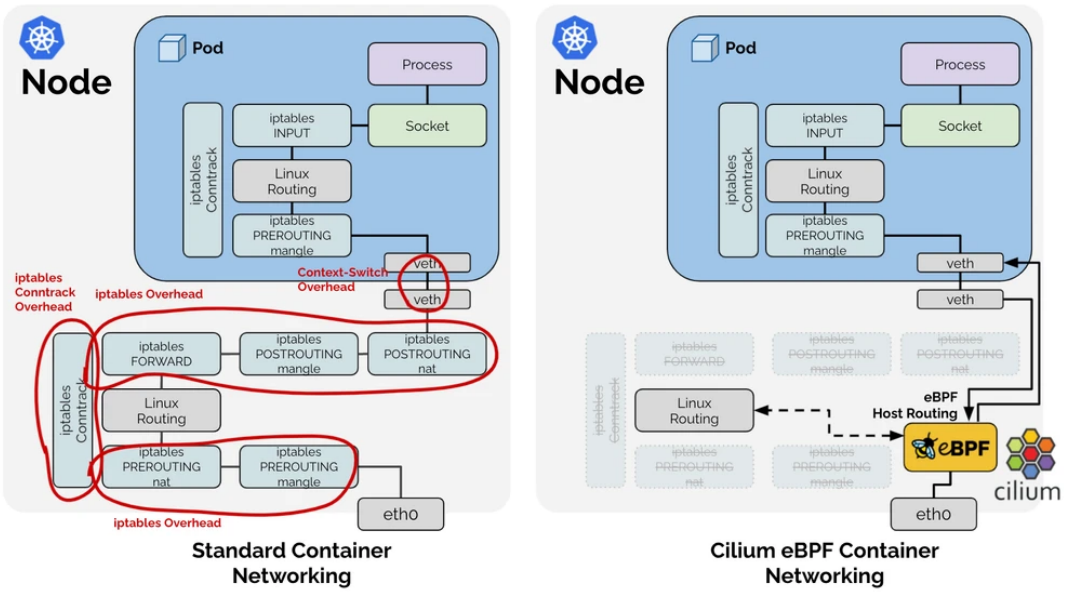


Figure 2.3: Cilium eBPF host-routing [37]

2.5. Related Work

As discussed in [38] 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 [38].

In [39] 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 [39].

Another paper [40] of [39] authors. Functionality, performance, and scalability are in focus,

the scale testbed up to 99 Iperf client and 99 Iperf server pod, mentioning that 100 pods are Kubernetes one node limit [40].

The authors of [41] 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) [41].

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

In modern Kubernetes networking, managing traffic flow into and outside a cluster is important for performance and security reasons. Different CNI plugins offer distinct mechanisms for controlling network traffic, each with its own approach to implementing networking. Understanding how traffic is managed in both ingress and egress scenarios is crucial for enhancing security, performance, and overall network efficiency.

3.1. Egress Scenario: Routing Outgoing Traffic via Egress Gateway

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

The IT department of a financial company manages Kubernetes clusters in their local laboratory. The infrastructure is used to create production-ready, efficient, and secure environment financial services where handling sensitive data and strict regulatory standards are critical. Leaving unmonitored critical traffic leaving the cluster can create vulnerabilities potentially exposing the system to data exfiltration from financial apps. They decided to analyze all outgoing traffic from financial services pods using some intrusion detection system (IDS) software. However, they are also providing some services which do not need 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 take advantage of an egress gateway routing all outgoing traffic from financial services into security tool to monitor and analyze packets.

However, end users started complaining about that, their apps started showing errors like "503 Service Unavailable". IT department administrators started seeking for a problem, and they conclude that created egress gateway is a bottleneck in their cluster. They started searching online for solutions and decided to create separate gateways for each deployment of their service [42]. End users stopped complaining about poor availability of services.

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 their capabilities in Antrea and Cilium CNI plugins, focusing on how they handle outbound traffic, integrate with other networking components. Understanding these implementations is essential for Kubernetes operators to select the right CNI plugin for their specific requirements.

Antrea

Antrea Egress CRD (Custom Resource Definition) API is a resource that controls how Pods in a cluster access external service. The resource specifies what egress IP use to selected pods. When a Pod communicates with an external network, the traffic is routed through the Node that has specified egress IP (egress gateway). The source IP address of traffic will then translate to the configured IP address [43].

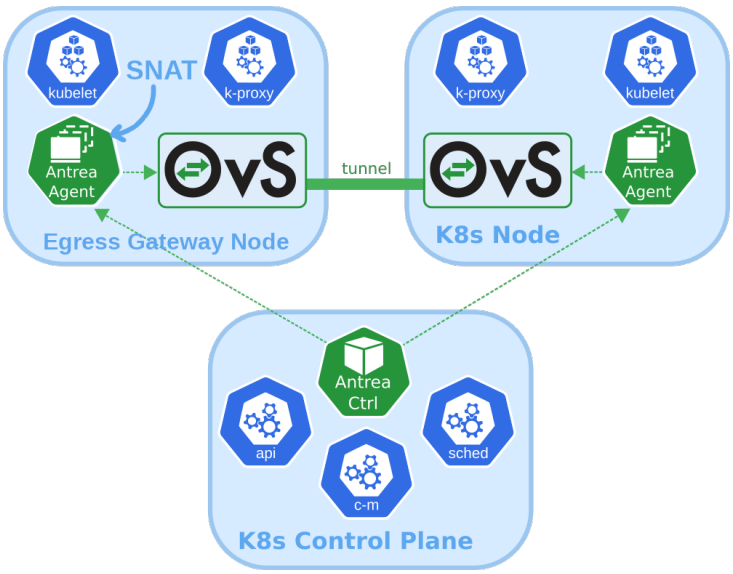


Figure 3.1: Antrea Egress Architecture [43]

Figure 3.1 shows architecture of communication flow when configured egress gateway in Antrea CNI. When a pod running on K8s Node tries to access external service (assume is

labeled to route its outbound traffic through egress node), the traffic is tunneled to gateway node and Antrea Agent is performing SNAT. After translation, the next network peer that is communicating with egress gateway sees its IP as a source IP instead of IP address of a pod [43] [44].

Let's explain egress configuration YAML from Listing 1 [43]:

– 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.
  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 fail-over egress gateway node using Antrea implementation. To do that egressIP and externalIPPool must be set (egressIP is in externalIPPool) and when current egress gateway stops working, another node within externalIPPool will be selected. This infrastructure, with a failover service, is part of a high availability setup for production environments (useful for earlier mentioned IT department) [43].

```
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 [43].

The ExternalIPPool resource from Listing 2 can be configured with the following fields [43]:

– 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

```
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 [43].

Cilium

To get advantage of cilium egress gateway features, eBPF masquerading must be enabled and node's kube-proxy component replaced with cilium implementation [26]. Some environments may not be suitable because Cilium requires a kernel version of 5.4 or higher [45]. 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). 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 [26].

Simmilian to antrea egress resources, cilium has its own CiliumEgressGatewayPolicy present on Listing 3 [26]:

Cilium allows matching the traffic that route through the egress gateway by [26]:

– podSelector – matching pods based of used selector, like previous matching labels in Antrea, or by matching expressions (key operator, values). More than one podSelector can be used

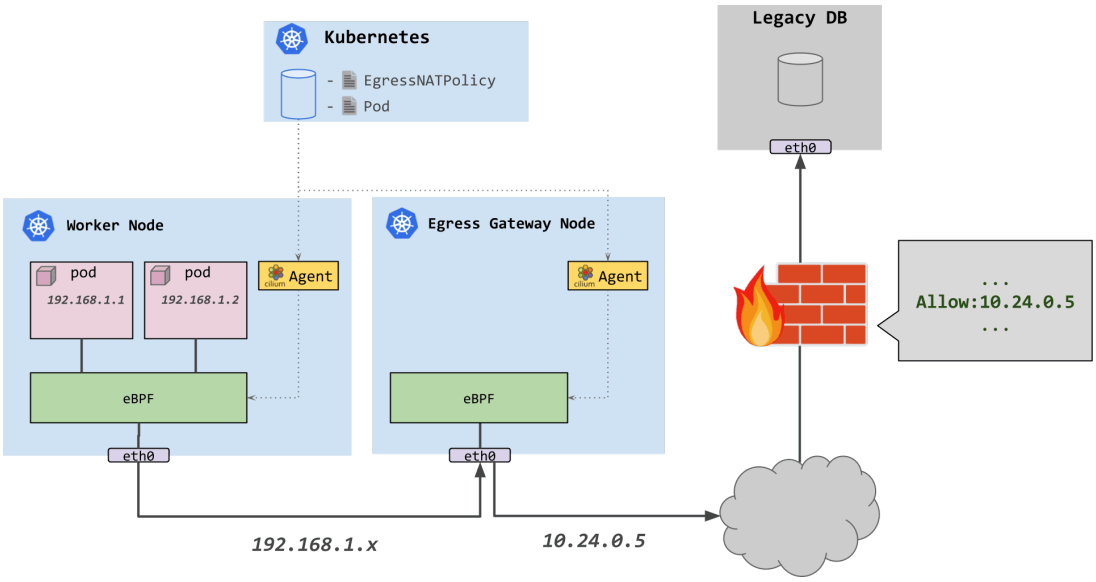


Figure 3.2: Cilium Egress Architecture [46]

– 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.

```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 [43].

Both Antrea and Cilium allows us to configure egress gateway in multiple ways. Cilium has more flexibility in defining which traffic should be routed through egress gateway, unlike

Antrea it can specify traffic by destination CIDR. Although cilium has more capabilities of matching egress traffic, Antrea implementation allows creating fail-over node, which will route traffic if main one fails. Creating high availability egress gateway in cilium is possible using enterprise, paid version of plugin. Cilium also takes advantage by using eBPF which is designed for large-scale clusters [34]. It is not clear which egress gateway CNI implementation to use, every of them has its advantages and drawbacks. Further both gateways will be evaluated using networking tools in creating local environment.

3.2. Ingress Scenario: Splitting Incoming Traffic via Gateway API

The Gateway API as a successor to the Ingress object provides more features for traffic management and a role-oriented approach to separate Kubernetes user/operator concerns. It is capable of traffic splitting, header modification, or URL rewriting. The Gateway API supports key protocols, like HTTP, HTTPS, TCP, UDP and gRPC. Offering a wide range of features, it can be used in some separate ways [47].

Canary Deployment

Canary Deployment is one of most common deployment methods used to roll out a new application version to end users, ensuring that everything is working as expected before full release. The whole point is to release new versions of software only for smalls group of people, leaving most users unaware of new release [48].

This is where traffic splitting feature from Gateway API might be used [49]. There might be five stages of deployment [48]:

- Initial state of app – stable version of application is served
- Canary stage – updated version of application is only visible for 5% of users. As some users can interact with new provisioned software, most common errors should be visible (if any).
- Early stage – second stage of canary deployment where new app is available for 25% of total connections. At this point less frequent bugs might be observed.

– Mid stage – allowing for 50% of end clients. Half of the traffic is routed to the latest version of the app. At this point, the performance of rolled out software is monitored.

– Late stage – most of the traffic (75%) is handled by the recent version of application. Stage that precedes full release of new software.

– Full stage – 100%, new application version is fully deployed for all users

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

Gateway API is not designed for software deployment, it does not have capabilities of rolling back application in automated way. In presented way of use the gateway is used only for weighted traffic splitting.

Traffic Mirroring with Gateway API

A company is offering weather API, not all endpoints are publicly available, some features are secured and paid. Securing these paid interfaces is not as critical as more sensitive and confidential data. Lately company decided to start analyzing incoming traffic on secured endpoints, because they may want to make sure only authorized requests are handled. However, the company does not have the infrastructure capabilities to analyze all incoming traffic on these endpoints. As their services are HTTP based inside Kubernetes cluster, they can get advantage of the Gateway API traffic splitting. The security team decided to route 40% of incoming traffic to a traffic analyzer to evaluate if and how requests might bypass the paywall. Cluster management (infrastructure providers and cluster operators at once in terms of roles in Gateway API model) decided to split traffic using Gateway API, as they need general usage of API Gateway (company offers RESTful APIs). Pure traffic splitting is not a case, because all incoming traffic has to be handled in response. The solution is to split 40% of traffic to a different Kubernetes service, then mirror traffic to analyzer and route back to the pod containing an app. While cluster managers implement second deployment with mirroring requests to traffic analyzer, app developers created HTTPRoute object with appropriate weights for each of services. Figure 3.3 shows how cluster infrastructure might look in this case.

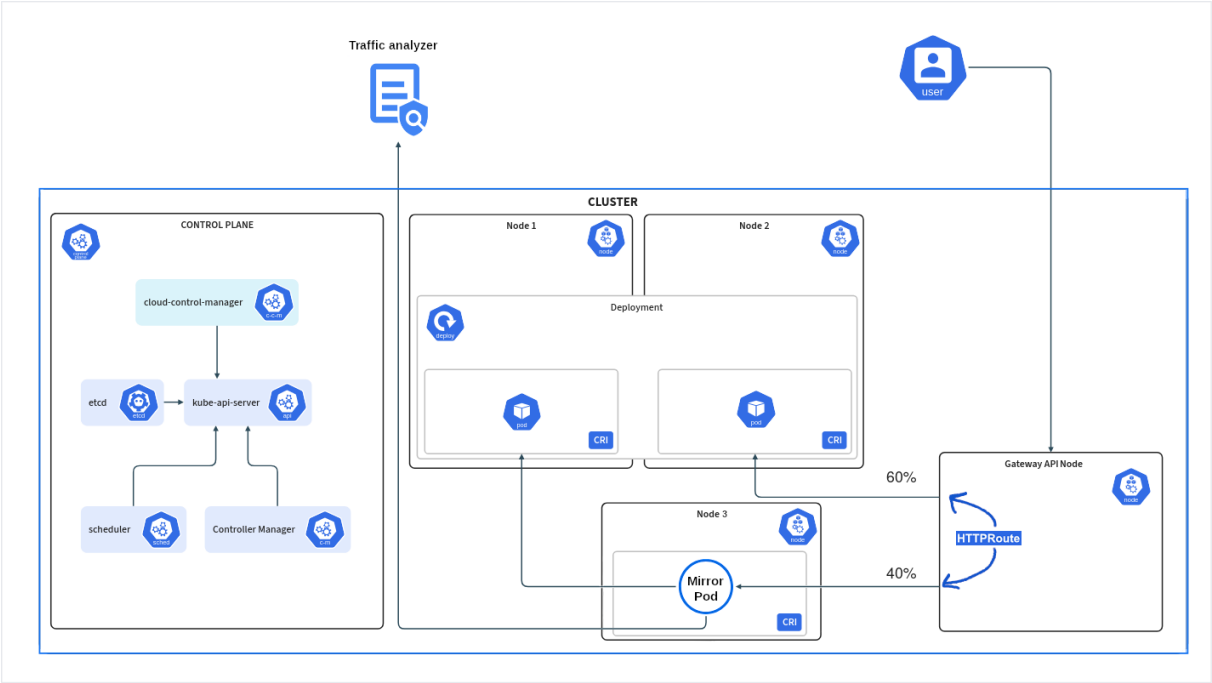


Figure 3.3: Example Kubernetes cluster with traffic mirroring

3.2.1. Traffic Splitting in Selected CNI Plugins

Unfortunately, Antrea plugin does not provide Gateway API implementation, in fact Cilium is the only one which does. To evaluate cluster networking implementation of Antrea CNI, NGINX Gateway Fabric can be used.

Antrea + NGINX

Figure 3.4 shows example cluster which configures Gateway API to work in canary stage. Antrea CNI is installed, antrea-ctrl and antrea-agent pods are deployed on nodes, OvS tunneling among nodes is set up. On the control plane node, the NGINX Gateway API is deployed as a pod. This differs from Cilium, where the Gateway API is not run as a single pod. In this setup, traffic is processed by the NGINX Gateway API pod, while Antrea handles the networking stack that integrates with NGINX to manage traffic routing.

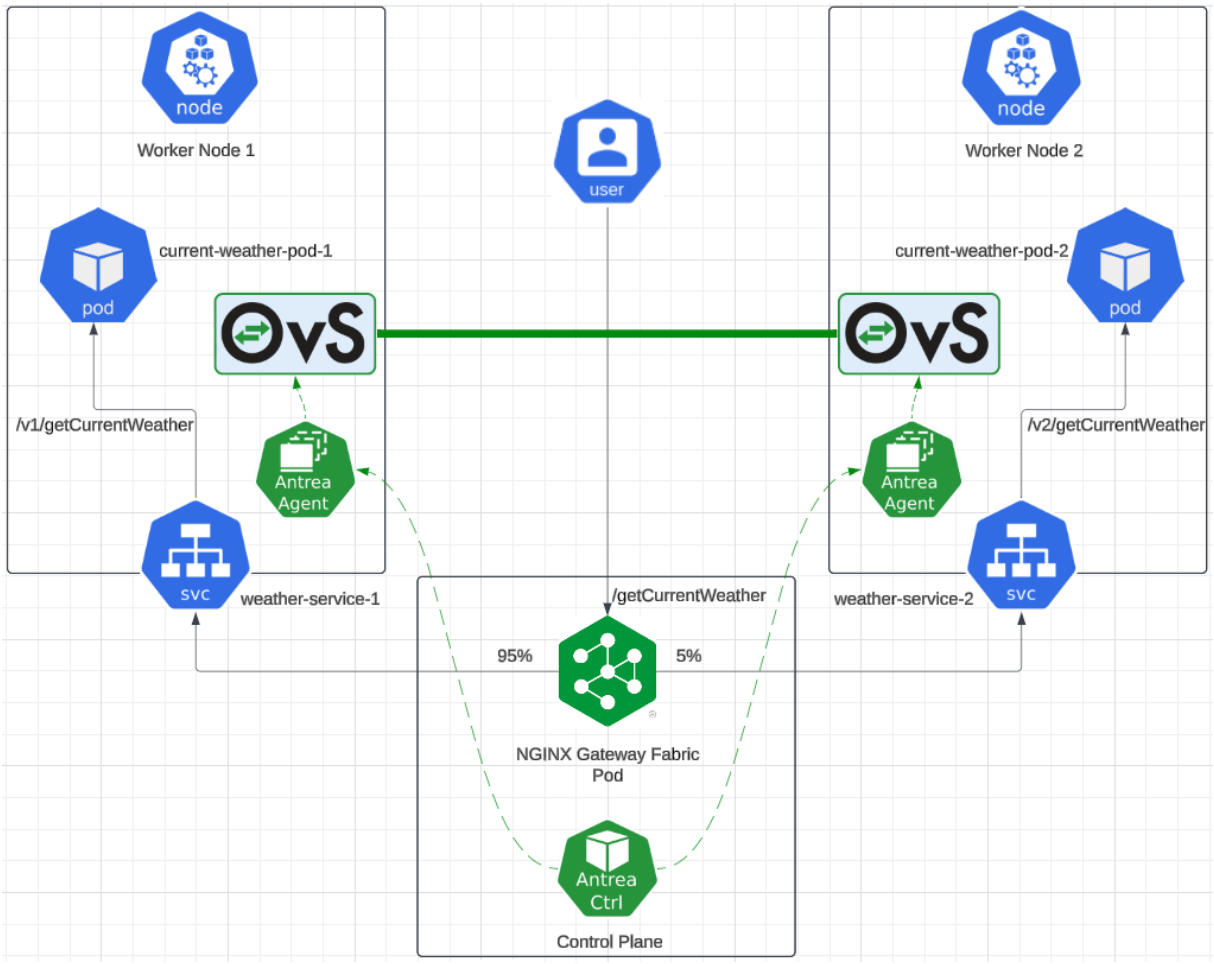


Figure 3.4: Example Kubernetes cluster with Antrea CNI and NGINX Gateway Fabric in canary stage of canary deployment

```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 [43].

Cilium

In figure 3.5 example canary cluster stack is presented using Cilium plugin. The arrows show real data traffic flow, while dashed arrows show configuration flow. HTTPRoutes are pulled by cilium-agent which prepares configuration and injects to Envoy proxy and eBPF. Envoy when sees incoming request, it knows the traffic splitting ratio and decides where to route, to a local pod on to different pod specified in HTTPRoute configuration. HTTPRoute resource for Cilium Gateway API will look almost exactly as in listing 4, the only difference is the parentRefs name, which defines which Gateway is being used.

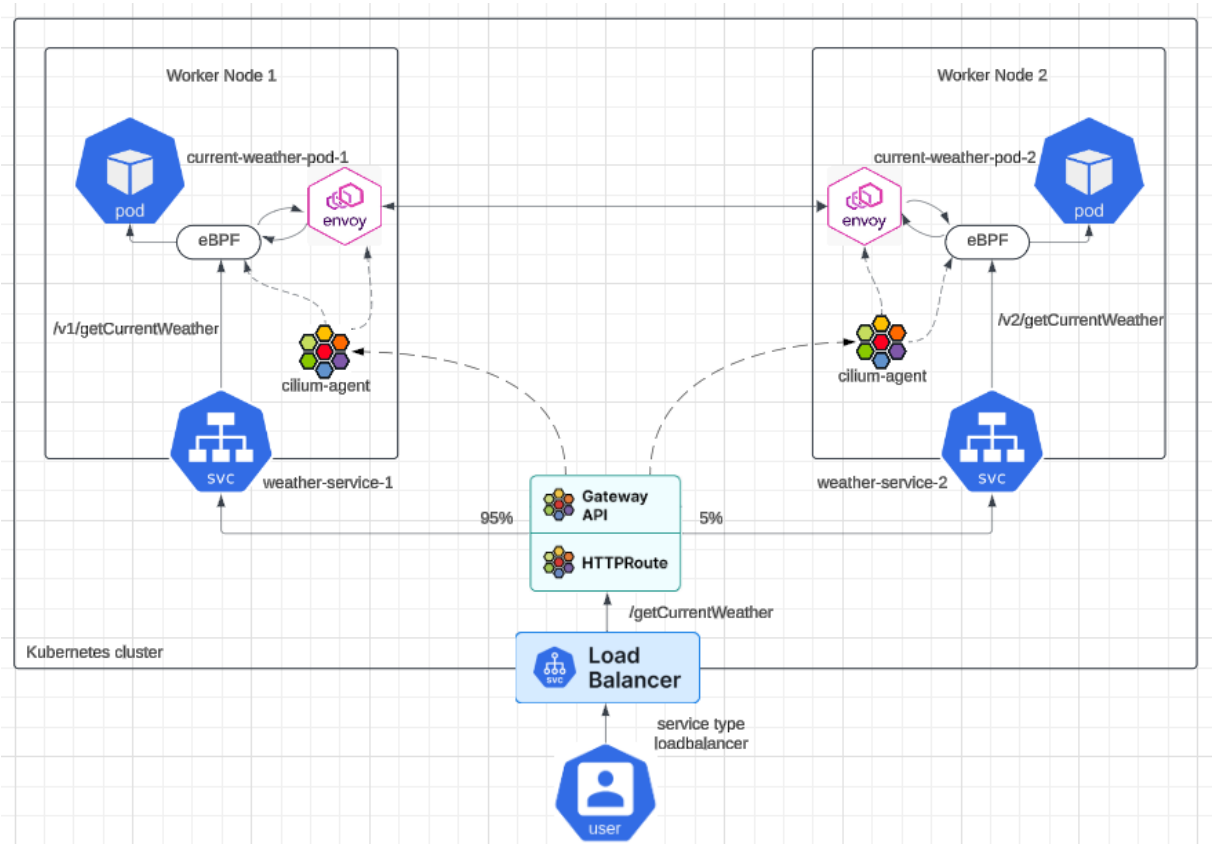


Figure 3.5: Example Kubernetes cluster with Cilium CNI in canary stage of canary deployment

4. Implementing Egress and Ingress Scenarios Using Selected CNI Plugins

This chapter presents the implementation of 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 on the public cloud (Azure). The tools used in this implementation with some 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

Cloud infrastructure consist 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, and Infrastructure as

Code (IaC) tool Terraform will be used to provision and interact with the cluster. Additionally, Ansible will be used for creating, configuring cluster setups along with running terraform and performance tools.

Ansible

Ansible is an open-source tool which can automate provisioning and configuring infrastructure. Configuration in Ansible is written in playbooks, which are YAML files as blueprints that contain 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 desired hosts [50] [51].

```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 [52].

```
[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 [53] [52] [54]

Listing 5 shows example Ansible playbook configuration. The host field defines the group of objects on which configuration script is executed. In this case instances specified in group named openstack_pool in ansible inventory showed on listing 6 will be created when using the playbook. Using var_files is possible to attach file contains, for example authentication variables required to access OpenStack cloud. Become is used to execute script as root user. Tasks and

roles are the place where the actual script is defined. It can be defined directly in tasks, or specified by roles, in this case will use script from ./roles/assign_floating_ip/tasks/main.yaml [51].

Iperf3

Iperf3 is a tool capable of measuring networking metrics. It supports TCP, UDP and SCTP protocols in IPv4 or IPv6 networks. The tool works in client-server architecture. Iperf3 will be used to evaluate throughput and round-trip time in egress scenario.

Kind

Kind is a tool used for creating local Kubernetes cluster. This tool can simulate real communication between nodes within one machine. It creates control plane and worker nodes using Docker containers to enable node to node communication. The important thing is that it does not provide any load balancer for assigning external IP addresses for Kubernetes services. In ingress scenario Gateway API requires outside cluster routable IP address, on local infrastructure MetalLB will be installed and configured to provide IPs [55].

Grafana K6

Grafana K6 is open-source tool designed for load testing by simulating virtual users accessing specified endpoints. The testing configuration is written in a JavaScript file using the k6 library.

MetalLB

MetalLB is an implementation of load balancer for bare metal Kubernetes. Kind does not provide implementation of load balancer. Without external tool like this, load balancer type service will persist in "pending" state. It is mandatory in ingress scenario to create load balancing service (in this case to allocate external IP address) for Gateway API [56].

Node Exporter

A tool that exports the current system's metrics, such as CPU usage, memory utilization, disk I/O, and network statistics. The metrics in OpenMetrics format are exposed on /metrics endpoint [57].

Prometheus

Prometheus is a monitoring and alerting tool, which stores data in time series, any data value is associated with time when it was collected. Stored data can be retrieved using PromQL (query language). The tool collects data by pulling from specified endpoints in configuration [58].

Terraform

Terraform is an open-source infrastructure as code (IaC) tool. It allows provision and manage infrastructure resources, cloud infrastructure, Kubernetes cluster, virtual machines, docker containers, storage, and SaaS features. Configuration files are written in a declarative language called HashiCorp Configuration Language (HCL).

The Terraform workflow is made up of three stages [59]:

1. Write – define in configuration file resources to be created
2. Plan – shows the actual resources that will be created based on the provided configuration and checks for any errors in the code
3. Apply – provision resources or applies changes defined in write stage to the infrastructure

4.2. Egress scenario implementation

The egress scenario compares Antrea and Cilium egress gateway implementation performance. The test involves using iperf3 in TCP mode to measure network performance. The iperf3 is located in a pod inside Kubernetes cluster, on the other hand iperf server runs on personal computer which launches cluster. The network resources are collected by iperf3, and CPU/memory usage is monitored by a node exporter. These 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 a cluster will leave cluster through Antrea egress gateway node implementation

2. antrea base – traffic leaves cluster using node on which pod is deployed

3. cilium egress gateway – cilium implementation of an egress gateway will route outbound traffic to the outside

4. cilium base – no redirecting at all

4.2.1. Antrea

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 on 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: Kind config used in both scenarios [60].

Four file containing variables included in the script:

1. common.yml – contains shared variables like Ansible become password to access root privileges on machine

2. egress_gateway.yml – scenario name or node name on which deploy gateway
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, like node names, job name for Prometheus and env type

The actual playbook from listing 7 consist of eight steps to perform automate provisioning infrastructure, running test and store output:

1. create_kind_cluster – creates kind cluster using config YAML
2. install_antrea – applies massive Antrea YAML containing custom resource definitions which define cluster networking
3. wait_until_antrea_installed – uses kubectl wait command and stops script until Antrea controller deployment is available
4. get_ip_for_egress_node – retrieves IP address of node by name specified in variable files on which deploy egress gateway in the next step
5. deploy_antrea_egress_gateway – turns on 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, deploys Prometheus Deployment and Node Exporter DaemonSet
7. terraform_run_egress_iperf – runs iperf3 server on laptop, saves current timestamp and runs iperf3 client Pod using terraform.
8. scrap_prometheus_data – part of playbook responsible for pulling CPU and memory metrics stored by Prometheus

The playbook from listing 7 produces following infrastructure:

Kubernetes cluster visible in figure 4.1 is provisioned on personal computer using Kind and configured using Ansible and Terraform. The cluster consists of three nodes, a control plane, a worker node, and an egress gateway, which blong to docker network. When role terraform_run_egress_iperf begins its execution and created iperf3 Pod is ready the test begins,

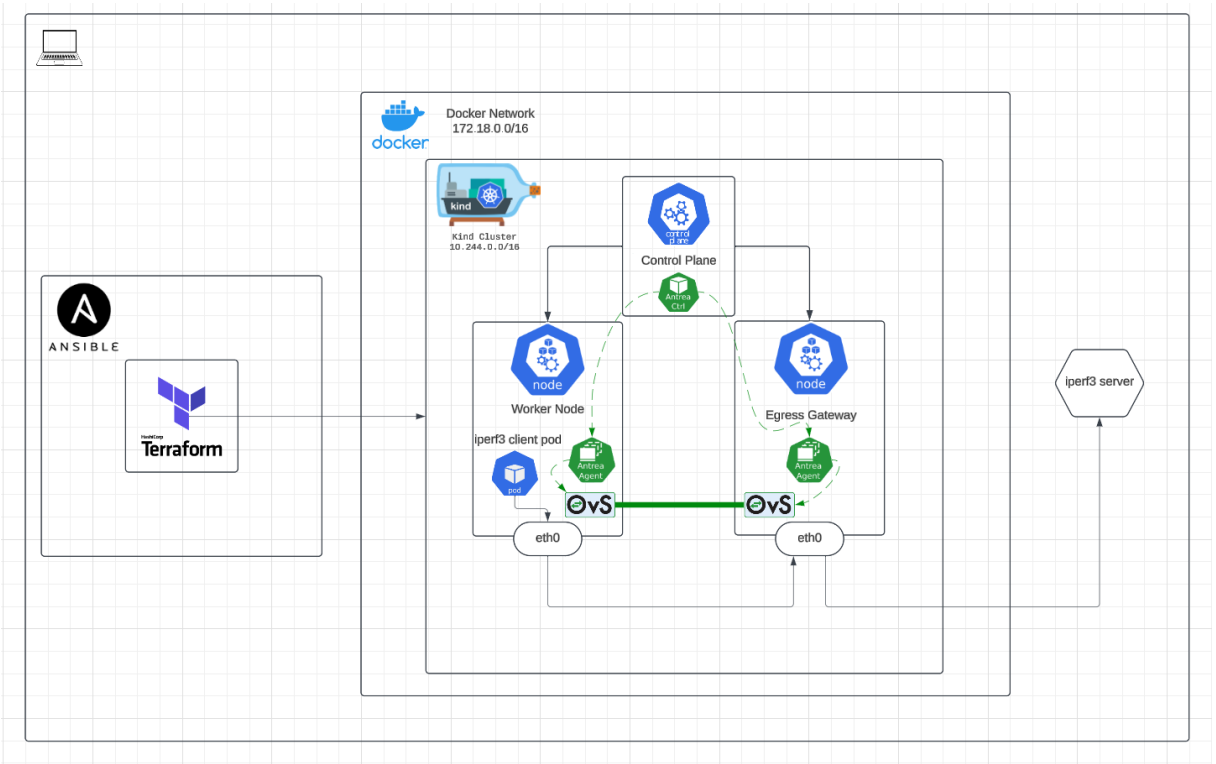


Figure 4.1: Antrea Egress Scenario infrastructure.

client using TCP protocol is sending data to server (outside cluster), which is routed through egress gateway. The Iperf3 server discovers that traffic is incoming from egress gateway node (it sees IP address of Egress Gateway Node as a source), because traffic from the Pod is SNATed. The iperf3 client sends data packets using TCP protocol, after receiving acknowledgment from server, networking metrics are stored in its memory and at the end of the test, JSON file with gathered data is saved inside a Pod. The node exporter is constantly scraping the metrics, which are pulled by a Prometheus to its database. After the measurement is over, unformatted data is downloaded from Prometheus and saved to CSV file.

4.2.2. Cilium

The playbook for cilium is really like the one which creates egress scenario using Antrea CNI. The Ansible roles are designed to be reused in different playbooks, the only differences are, CNI installation, Egress Gateway deployment and in cilium desired node is labelled with egress-node=true. The infrastructure can be seen in figure 4.2

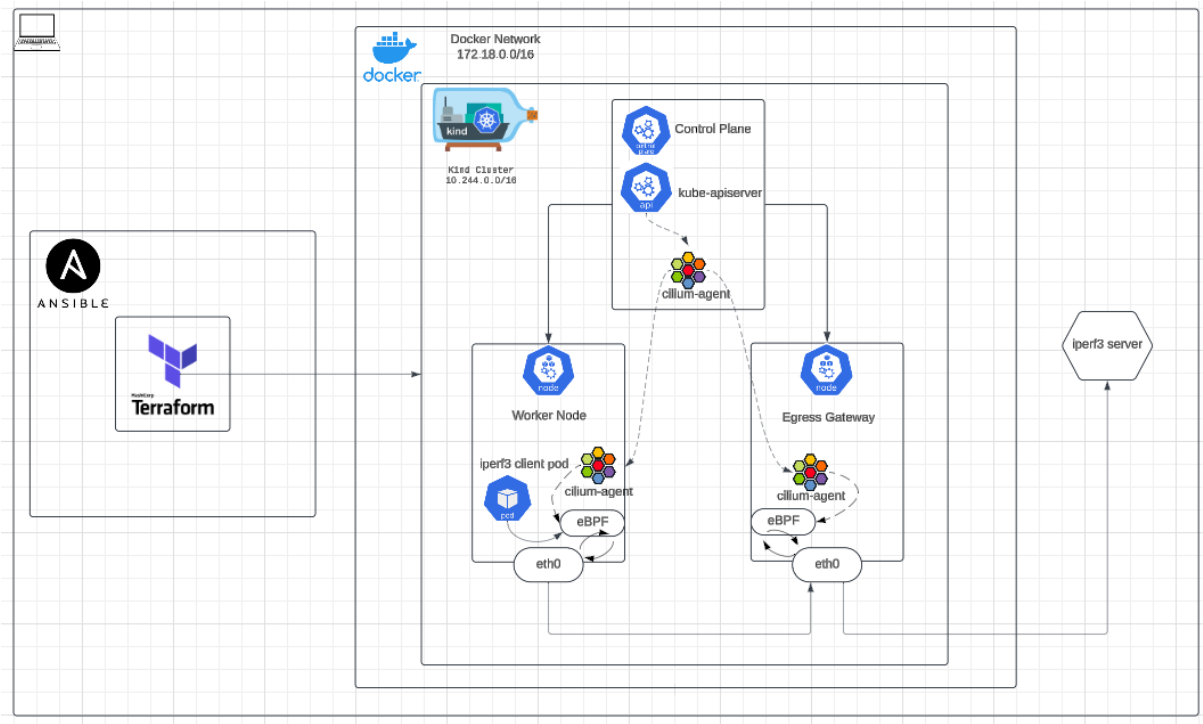


Figure 4.2: Cilium Egress Scenario infrastructure.

4.3. Ingress scenario implementation

The ingress scenario evaluates the Antrea and Cilium Container Networking Plugins CPU and memory usage while using Gateway API in handling weighted traffic routing. The experiment involves using the Gateway API to route 40% of incoming requests to one Pod and evaluating 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 by allocating different number of virtual users talking to the Gateway API. These are one, ten, a hundred and a thousand virtual users. The traffic is initiated using k6 tool, which runs inside container on personal computer. The network generator is performing HTTP request at Gateway API and saves received response extracting Pod name to the text file (one per line). Having a list of responses containing names of two pods, accuracy of Gateway API traffic splitting is calculated. CPU and memory usage is monitored with a Node Exporter during whole test and fetched at the end of 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 configured properly to be authenticated with an azure account. The script shown on listing 8 is responsible for creating Kubernetes cluster in Azure Services. It is important to choose appropriate location for cluster, define default node pool (virtual machine type and node count) and to get rid of default CNI by setting network_plugin in network_profile to "none" if different networking plugin is preferred [61].

```
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 [61].

4.3.2. Antrea

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

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 util is fully installed

6. deploy_antrea_ingress_scenario – deploys ingress scenario (echo Pods and Gateway API) using terraform

7. monitoring – enables monitoring with Node Exporter and Prometheus

8. register_gateway_api_ip – registers IP address of Gateway API for k6 tool

9. run_k6 – creates container with k6 which generates HTTP traffic accessing Gateway API

10. scrap_prometheus_data – downloads data about CPU and memory utilization

```yaml
- 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: Kind config used in both scenarios [60].

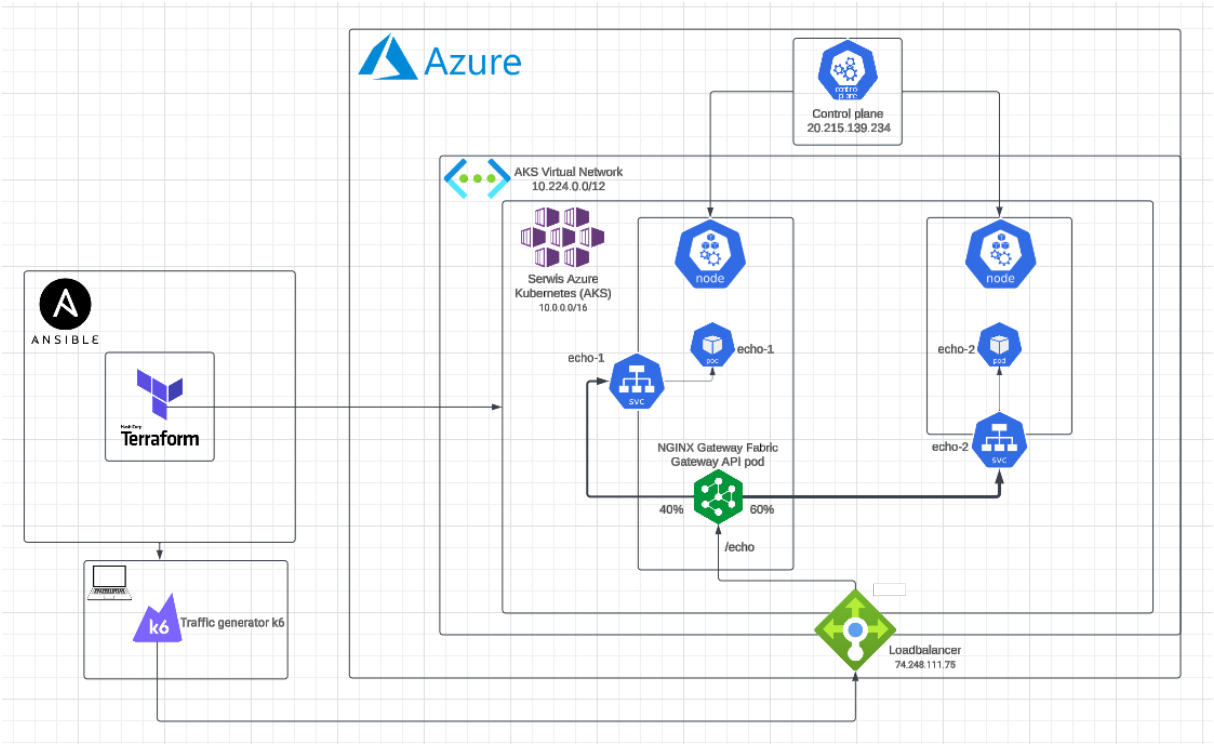


Figure 4.3: Antrea Ingress Scenario infrastructure.

4.3.3. Cilium

The ingress scenario with the Cilium CNI plugin does not need to install the NGINX Gateway Fabric, as it utilizes its own built-in implementation.

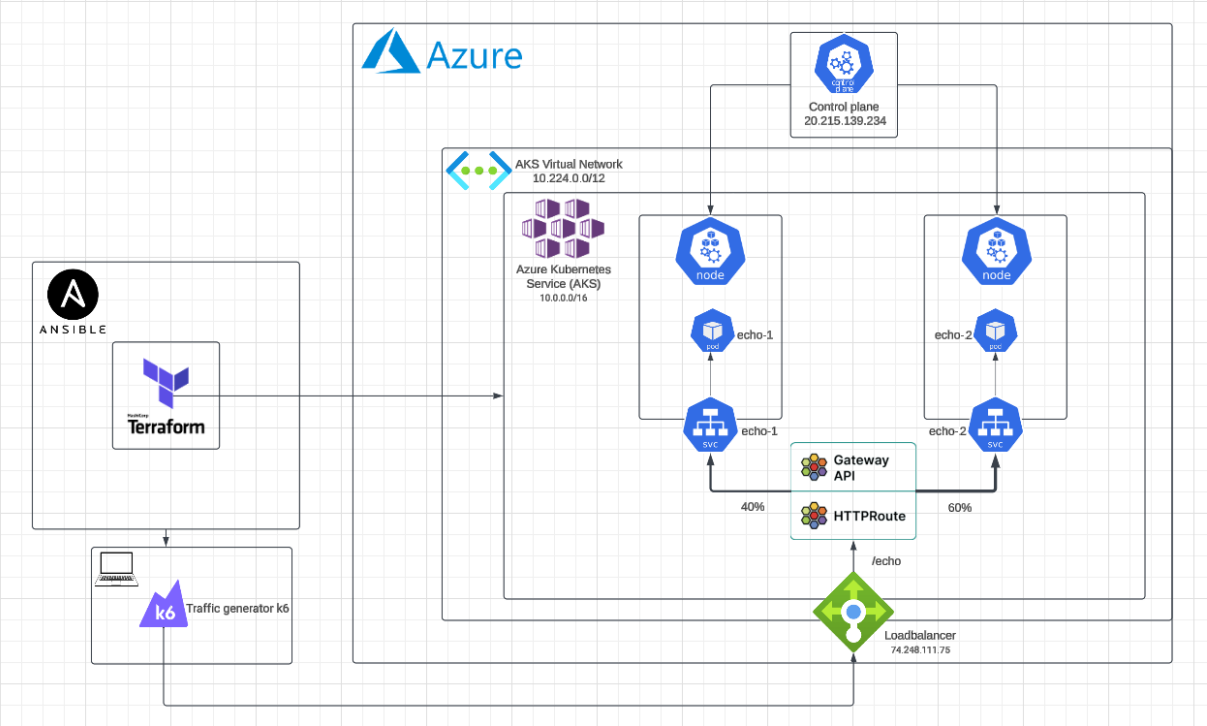


Figure 4.4: Cilium Ingress Scenario infrastructure.

4.3.4. 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 is always leaving 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 more distinct separation between the control plane and worker nodes. This architectural difference can affect the measurements (in comparison to 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 exclusively used by the cluster itself, not by the client. However, when running local stack, the client is part of a laptop on which cluster is running, what might influence measurements.

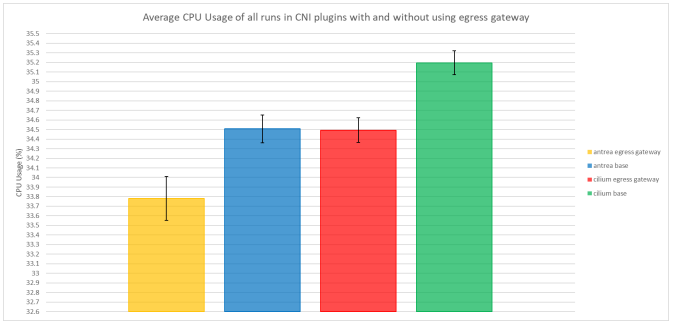
5. Performance comparison

In this chapter, test results for both ingress and egress scenarios will be presented and analyzed, with a focus on comparing CPU and memory usage in each case. For ingress, the analysis will cover weighted traffic splitting through the Gateway API, evaluating its impact on resource utilization and accuracy of traffic distribution. For egress, the evaluation will include traffic routing through an egress gateway, comparing throughput and round-trip time. The results will highlight differences in resource efficiency and networking performance among Antrea and cilium CNI plugins, identifying which CNI offers better optimization for these scenarios.

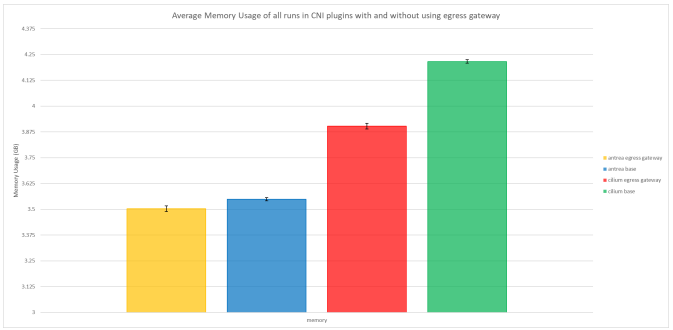
5.1. Egress Scenario

Resource consumption

The figures 5.1a and 5.1b show resource utilization is lower when using egress gateway than without redirecting outgoing packets via additional node. This might 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 is translated), allowing the node to leave CPU time for the pod which runs iperf3 client. In the case where an egress gateway is not used, the individual nodes are responsible for handling more operations, which can lead to higher resource allocation. In figures 5.2a and 5.2b there are showed all eight runs in all four test cases. The sixth run in figure 5.2a during cilium egress gateway case is an outliner and was not used to calculate total average. Overall CPU usage is lower when cluster uses Antrea networking. If it comes to memory usage, Antrea uses 10% less while using egress, up to almost 20% less without additional node in comparison to cilium.

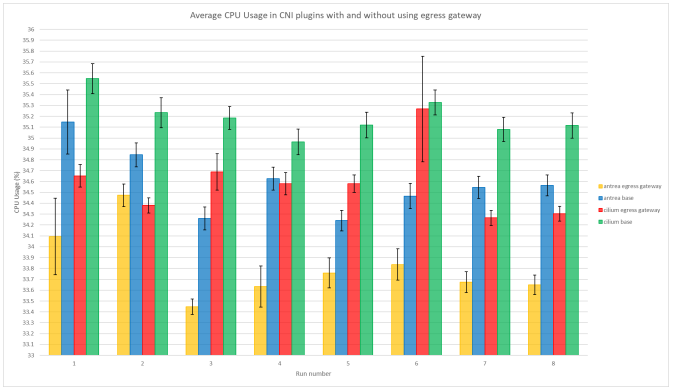


(a)

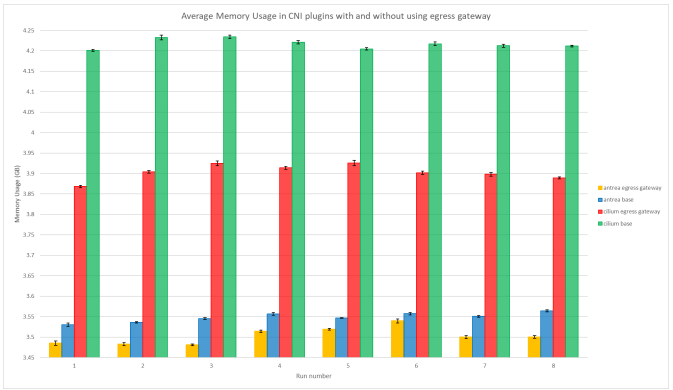


(b)

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

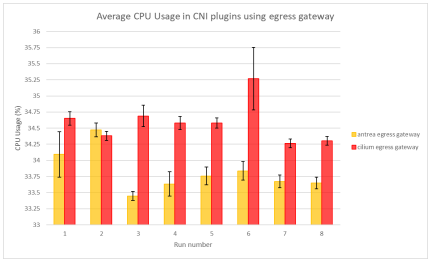


(a)

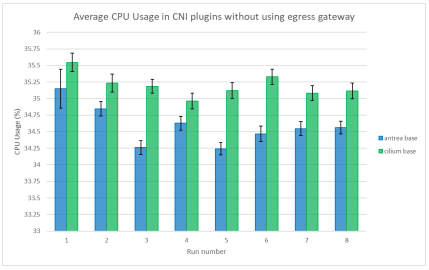


(b)

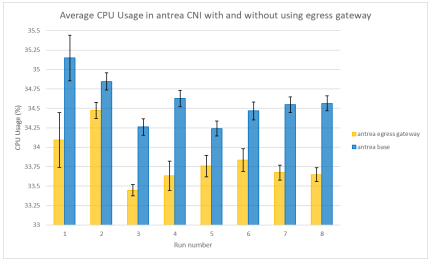
Figure 5.2: Average resource comsumption in egress scenario in each run, (a) CPU, (b) Memory



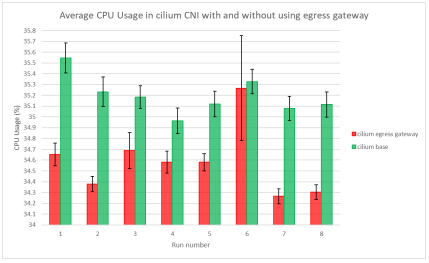
(a)



(b)

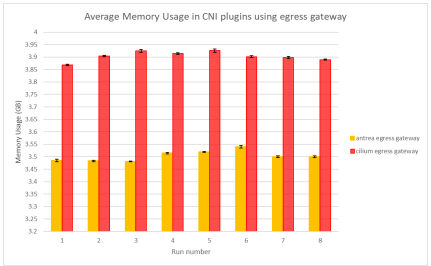


(c)

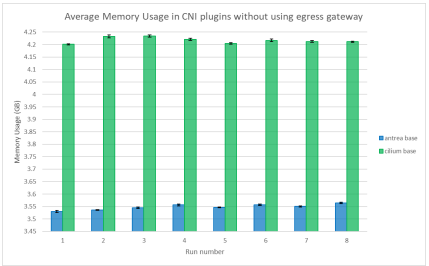


(d)

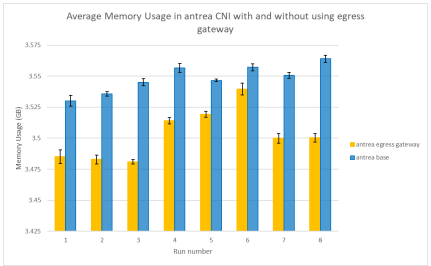
Figure 5.3: CPU usage in all four test cases.



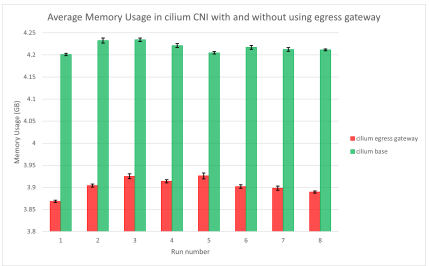
(a)



(b)



(c)



(d)

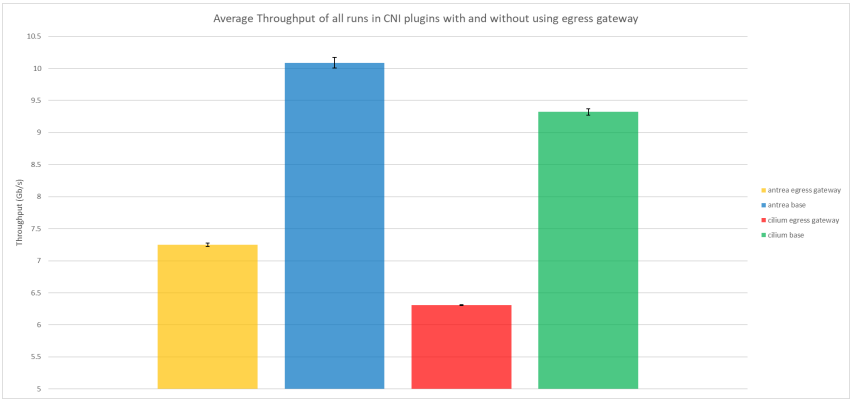
Figure 5.4: Memory usage in all four test cases.

Figures 5.3a, 5.3b, 5.4a, 5.4b compare average CPU and memory usage between cilium and Antrea CNI in each of eight runs. It shows how both container network interface plugins perform under the same test conditions, what helps to evaluate which plugin consumes less resources. In only second run Antrea has not used less CPU than cilium used. Comparing resource utilization

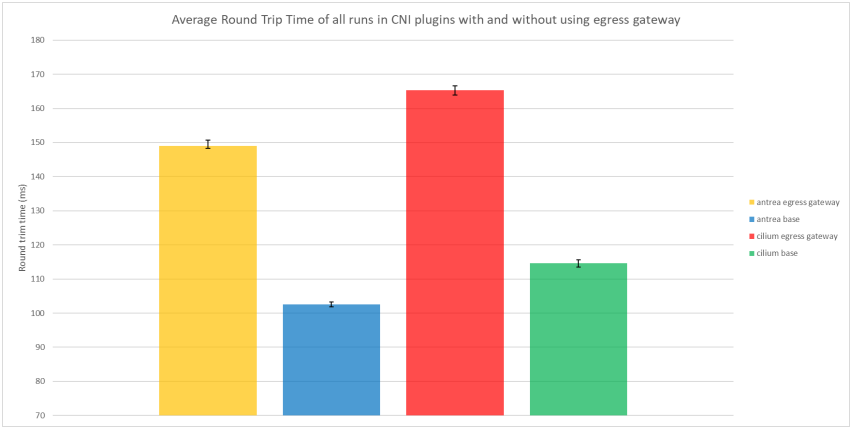
showed on figures 5.3c, 5.3d, 5.4c, 5.4d, highlights how using an egress gateway with each CNI plugins affects resource usage compared to not using one.

Networking performance

The figures 5.5a and 5.5b provide overall performance summary as an average of all runs. Antrea handles traffic more efficiently, with lower round trip time regardless of using egress gateway or not. 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.

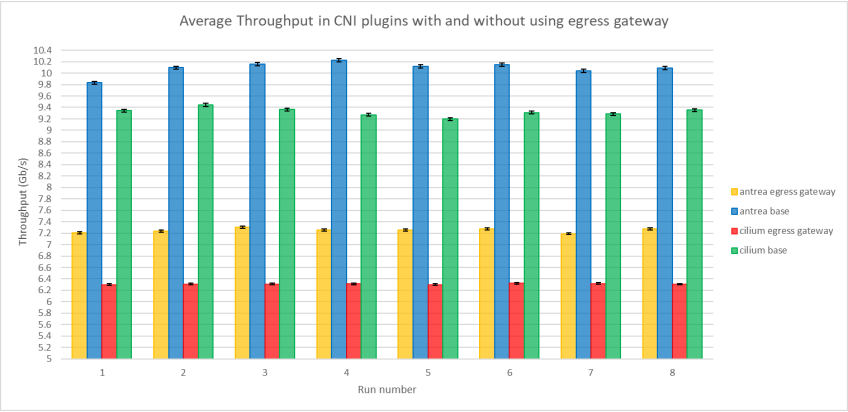


(a)

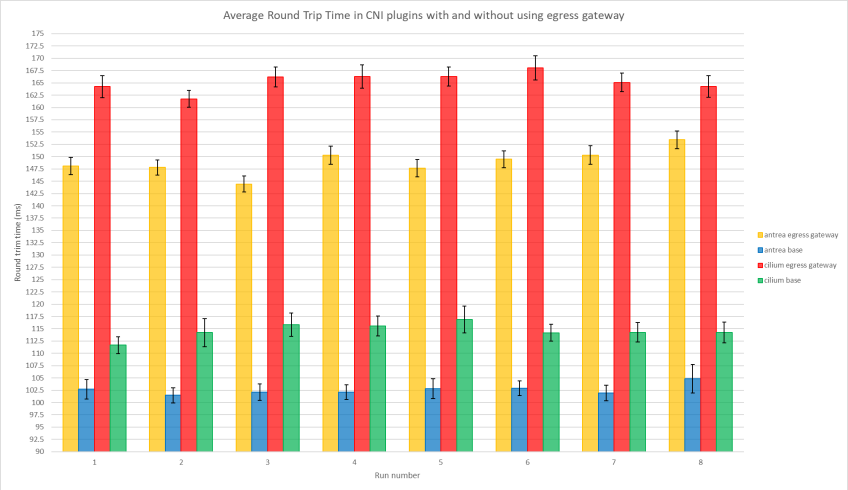


(b)

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



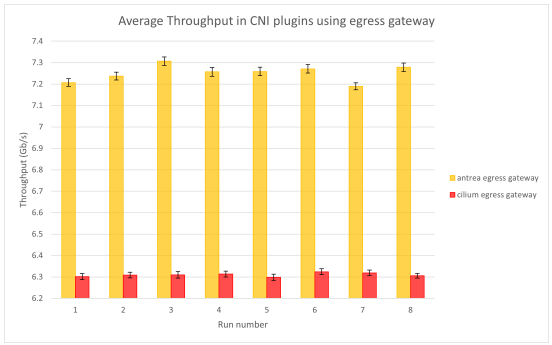
(a)



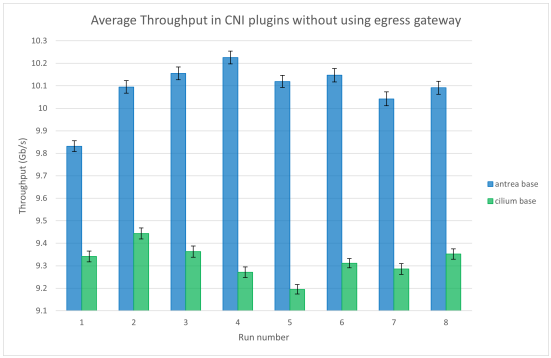
(b)

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

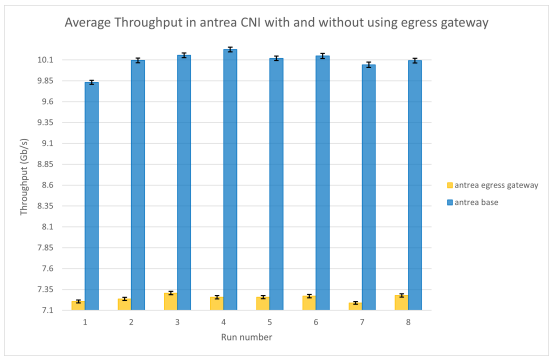
Figures 5.7a, 5.7b, 5.8a, 5.8b illustrate average throughput and round trip time in eight runs, comparing the performance of networking plugins. Throughput with cilium egress gateway remains stable, hovering around 6.3 Gb/s and with tighter confidence intervals indicating more stability. Every single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



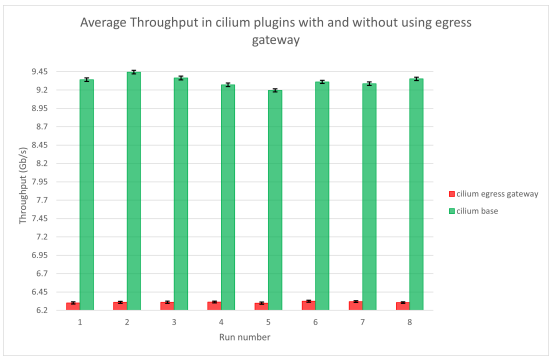
(a)



(b)

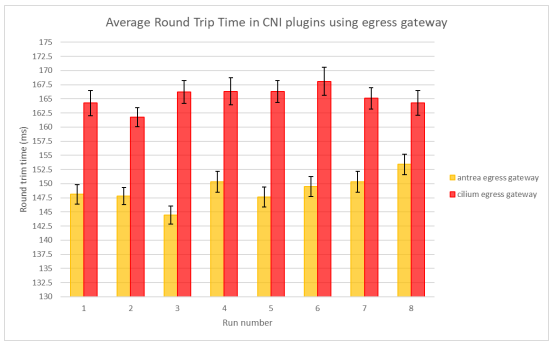


(c)

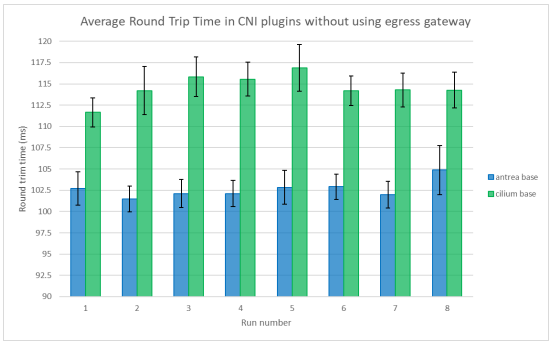


(d)

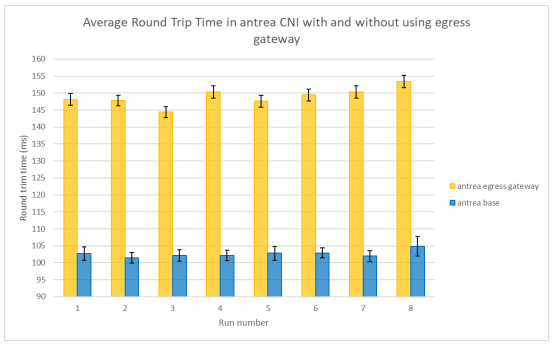
Figure 5.7: Throughput in all four test cases.



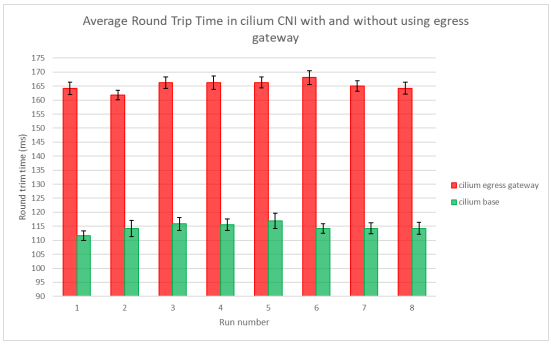
(a)



(b)



(c)

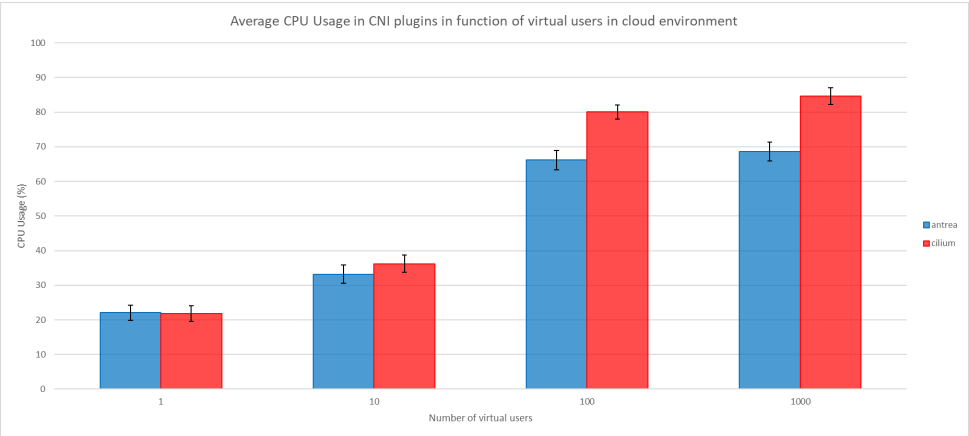


(d)

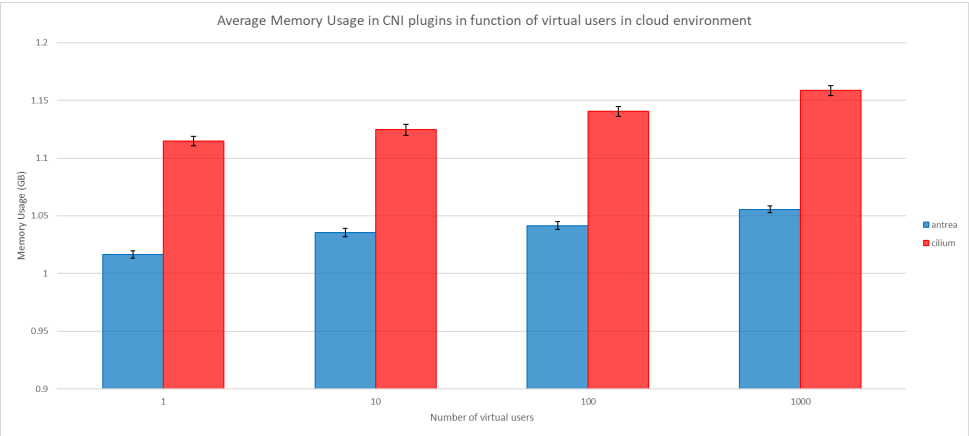
Figure 5.8: Round Trip Time in all four test cases.

5.2. Ingress Scenario

Resource consumption

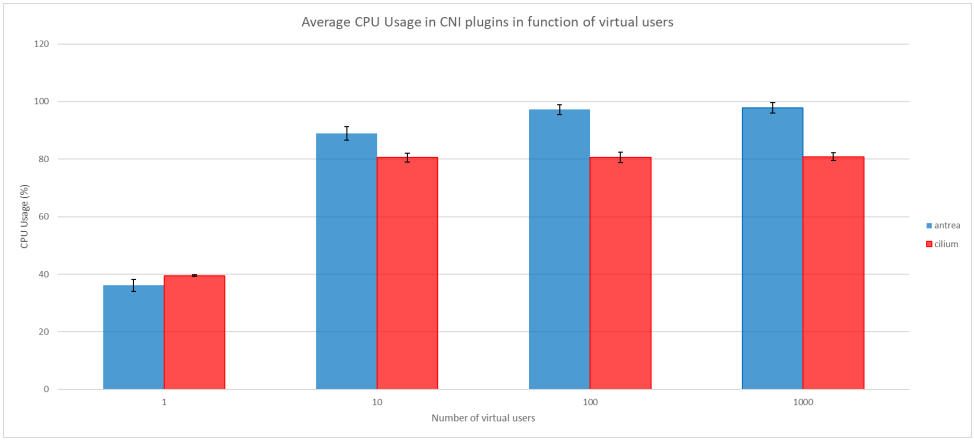


(a)

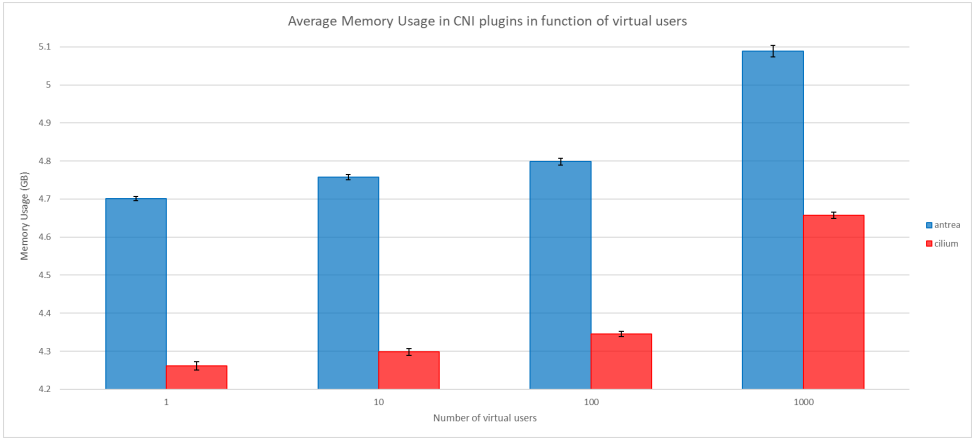


(b)

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



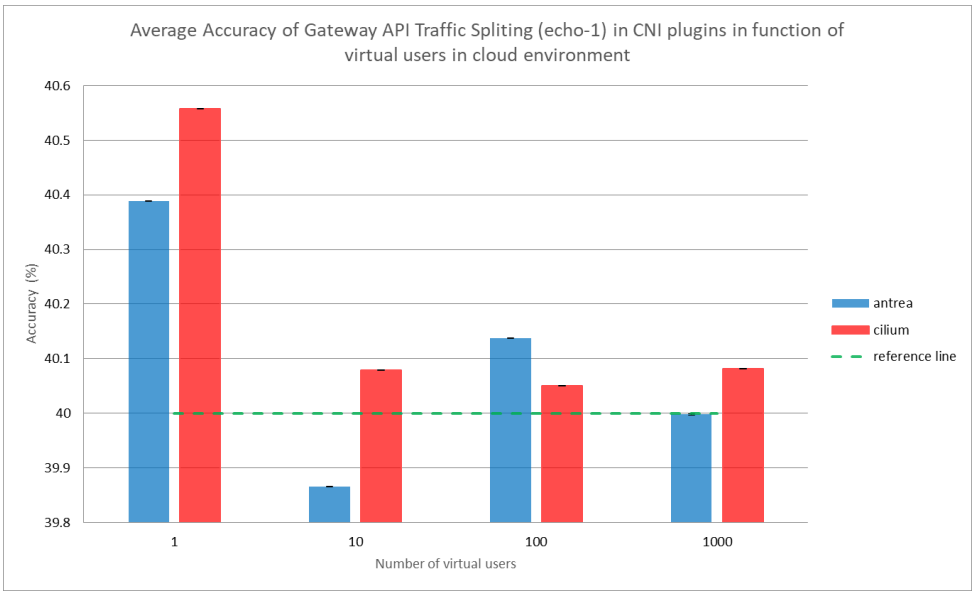
(a)



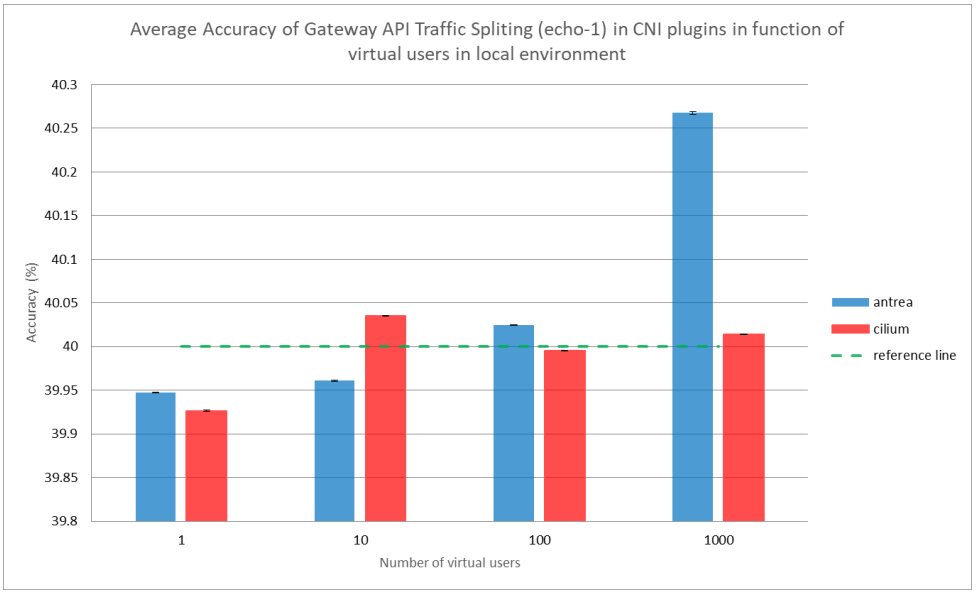
(b)

Figure 5.10: Average resource utilization in ingress scenario with increasing virtual users in local 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 the memory usage shows lower increases. In the cloud environment, Antrea demonstrates better resource efficiency, whereas in the cloud environment, Cilium consumes less resources.



(a)



(b)

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

When comparing traffic weighting accuracy 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 thousand users generate massive load. In a local environment, Cilium is more accurate in splitting traffic to a value specified in HTTPRoute in every scenario except for the case of a single user. Antrea in local environment has wider confidential intervals, and distance from reference point is high in comparison to cilium, making it less effective during high load period.

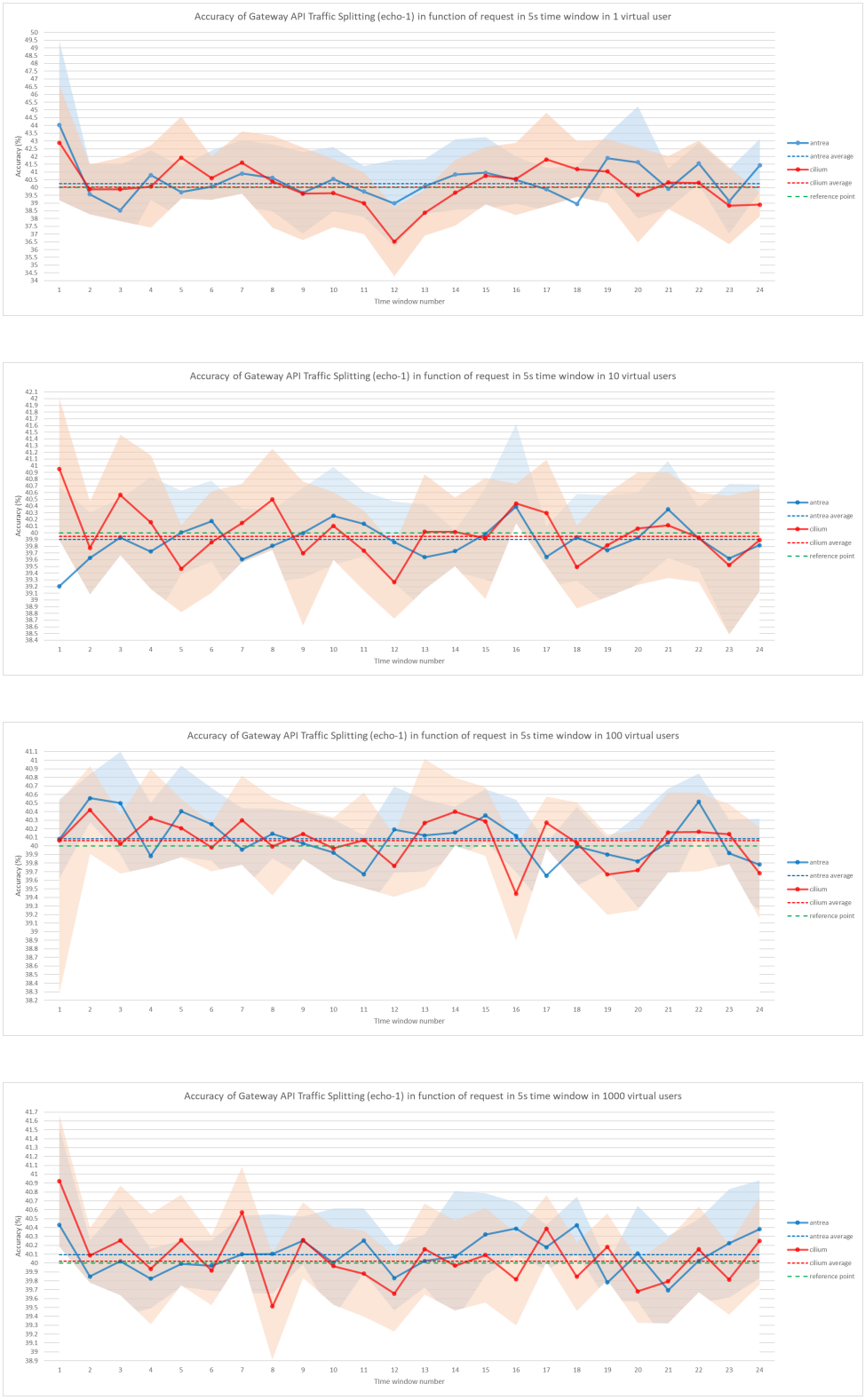
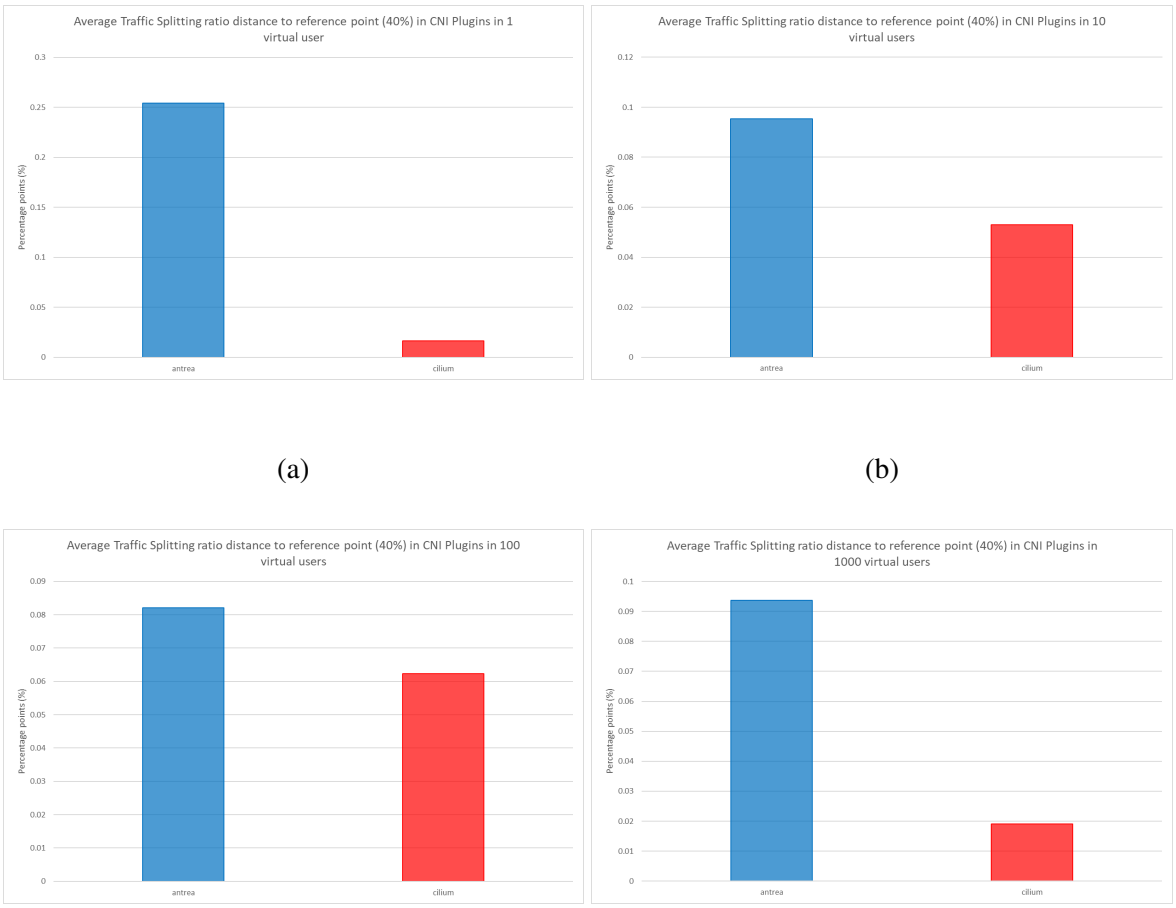


Figure 5.12: Average traffic splitting ratio in function of five second request time window with increasing virtual users; one, ten, houndred and tousand



(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

Analyzing plots in figure 5.12 show overlapping confidence intervals in all cases, indicating some variability in the results. However, when averaging the values, it becomes clear that cilium performs better, especially as the number of VUs is one or a thousand. The distance to reference points can be seen in figure 5.13. The overlapping confidence intervals show statistical uncertainty, but the average values demonstrate that cilium performs higher accuracy traffic weighting compared to Antrea and nginx test case.

Bibliography

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

[2] Docker Inc. What is an image? https://docs.docker.com/get-started/docker-concepts/building-images/. Accessed, 07-Dec-2024.

[3] The Kubernetes Authors. Overview. https://kubernetes.io/docs/concepts/overview/. Accessed, 07-Dec-2024.

[4] The Kubernetes Authors. Cluster Architecture. https://kubernetes.io/docs/concepts/architecture/. Accessed, 07-Dec-2024.

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

[6] The Kubernetes Authors. Kubernetes Scheduler. https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/. Accessed, 08-Dec-2024.

[7] The Kubernetes Authors. Container Runtimes. https://kubernetes.io/docs/setup/production-environment/container-runtimes/. Accessed, 08-Dec-2024.

[8] The Kubernetes Authors. Namespaces. https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/. Accessed, 10-Dec-2024.

[9] The Kubernetes Authors. Pods. https://kubernetes.io/docs/concepts/workloads/pods/. Accessed, 08-Dec-2024.

[10] The Kubernetes Authors. ReplicaSet. https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/. Accessed, 10-Dec-2024.

[11] The Kubernetes Authors. Deployments. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. Accessed, 10-Dec-2024.

[12] The Kubernetes Authors. DaemonSet. https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/. Accessed, 10-Dec-2024.

[13] The Kubernetes Authors. StatefulSets. https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/. Accessed, 10-Dec-2024.

[14] The Kubernetes Authors. Jobs. https://kubernetes.io/docs/concepts/workloads/controllers/job/. Accessed, 10-Dec-2024.

[15] The Kubernetes Authors. CronJob. https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/. Accessed, 10-Dec-2024.

[16] The Kubernetes Authors. Service. https://kubernetes.io/docs/concepts/services-networking/service/. Accessed, 10-Dec-2024.

[17] The Kubernetes Authors. Cluster Networking. https://kubernetes.io/docs/concepts/cluster-administration/networking/. Accessed, 10-Dec-2024.

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

[19] The Kubernetes Authors. Ingress. https://kubernetes.io/docs/concepts/services-networking/ingress/. Accessed, 10-Dec-2024.

[20] The Kubernetes Authors. Ingress Controllers. https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/. Accessed, 10-Dec-2024.

[21] The Linux Foundation. Introduction. https://gateway-api.sigs.k8s.io/. Accessed, 11-Dec-2024.

[22] The Linux Foundation. GatewayClass. https://gateway-api.sigs.k8s.io/api-types/gatewayclass/. Accessed, 11-Dec-2024.

[23] The Linux Foundation. Gateway. https://gateway-api.sigs.k8s.io/api-types/gateway/. Accessed, 11-Dec-2024.

[24] The Linux Foundation. HTTPRoute. https://gateway-api.sigs.k8s.io/api-types/httproute/. Accessed, 11-Dec-2024.

[25] Inc. Tigera. Kubernetes egress. https://docs.tigera.io/calico/latest/about/kubernetes-training/about-kubernetes-egress. Accessed, 13-Dec-2024.

[26] Cilium Authors. Egress Gateway. https://docs.cilium.io/en/stable/network/egress-gateway/egress-gateway/. Accessed, 12-Dec-2024.

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

[28] Cilium Authors. Cilium BGP Control Plane. https://docs.cilium.io/en/stable/network/bgp-control-plane/bgp-control-plane/. Accessed, 12-Dec-2024.

[29] Cilium Authors. Gateway API Support. https://docs.cilium.io/en/stable/network/servicemesh/gateway-api/gateway-api/. Accessed, 12-Dec-2024.

[30] Cilium Authors. Transparent Encryption. https://docs.cilium.io/en/stable/security/network/encryption/. Accessed, 12-Dec-2024.

[31] A Linux Foundation Collaborative Project. Production Quality, Multilayer Open Virtual Switch. https://www.openvswitch.org/. Accessed, 12-Dec-2024.

[32] The Kubernetes Authors. Network Policies. https://kubernetes.io/docs/concepts/services-networking/network-policies/. Accessed, 12-Dec-2024.

[33] Inc. Tigera. Kubernetes policy, demo. https://docs.tigera.io/calico/latest/network-policy/get-started/kubernetes-policy/kubernetes-demo. Accessed, 12-Dec-2024.

[34] Jeremy Colvin. Introduction to Cilium & Hubble. https://docs.cilium.io/en/latest/overview/intro/. Accessed, 12-Dec-2024.

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

[36] 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.

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

[38] 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.

[39] 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.

[40] 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.

[41] 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.

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

[43] The Antrea Contributors. Documentation. https://antrea.io/docs/v1.3.0/docs/egress/#what-is-egress. Accessed, 15-Dec-2024.

[44] The Antrea Contributors. Documentation. https://antrea.io/docs/v1.13.0/docs/noencap-hybrid-modes/. Accessed, 15-Dec-2024.

[45] Cilium Authors. Cilium Quick Installation. https://docs.cilium.io/en/stable/gettingstarted/k8s-install-default/. Accessed, 15-Dec-2024.

[46] Cilium Authors. Egress Gateway. https://cilium.io/use-cases/egress-gateway/. Accessed, 15-Dec-2024.

[47] Cilium Authors. Gateway API. https://cilium.io/use-cases/gateway-api/. Accessed, 16-Dec-2024.

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

[49] Cilium Authors. Traffic Splitting Example. https://docs.cilium.io/en/stable/network/servicemesh/gateway-api/splitting/. Accessed, 16-Dec-2024.

[50] Inc. Red Hat. Introduction to Ansible. https://docs.ansible.com/ansible/latest/getting_started/introduction.html. Accessed, 17-Dec-2024.

[51] Inc. Red Hat. Creating a playbook. https://docs.ansible.com/ansible/latest/getting_started/get_started_playbook.html. Accessed, 17-Dec-2024.

[52] Inc. Red Hat. openstack.cloud.server module – Create/Delete Compute Instances from OpenStack. https://docs.ansible.com/ansible/latest/collections/openstack/cloud/server_module.html. Accessed, 17-Dec-2024.

[53] Inc. Red Hat. How to build your inventory. https://docs.ansible.com/ansible/latest/inventory_guide/intro_inventory.html. Accessed, 17-Dec-2024.

[54] Inc. Red Hat. Special Variables. https://docs.ansible.com/ansible/latest/reference_appendices/special_variables.html. Accessed, 17-Dec-2024.

[55] The Kubernetes Authors. Docs. https://kind.sigs.k8s.io/. Accessed, 17-Dec-2024.

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

[57] Cloud Native Computing Foundation. Node exporter. https://github.com/prometheus/node_exporter. Accessed, 18-Dec-2024.

[58] Cloud Native Computing Foundation. Node exporter. https://prometheus.io/docs/introduction/overview/. Accessed, 18-Dec-2024.

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

[60] The Kubernetes Authors. Docs. https://kind.sigs.k8s.io/docs/user/configuration/. Accessed, 17-Dec-2024.

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