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]:

– Protable 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 to interact 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 depends 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 new changes. Consists of layers. The layer con-

tains one modification made to an image. All layers are cachable and can be reused when building an image. The mechanism is really 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 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 to specify 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 to add 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 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 one of each is needed to run a container, but to provide more robust and reliable production cluster is better to use two to three control plane nodes [4].

Figure 2.1: Kubernetes Cluster Architecture [4]
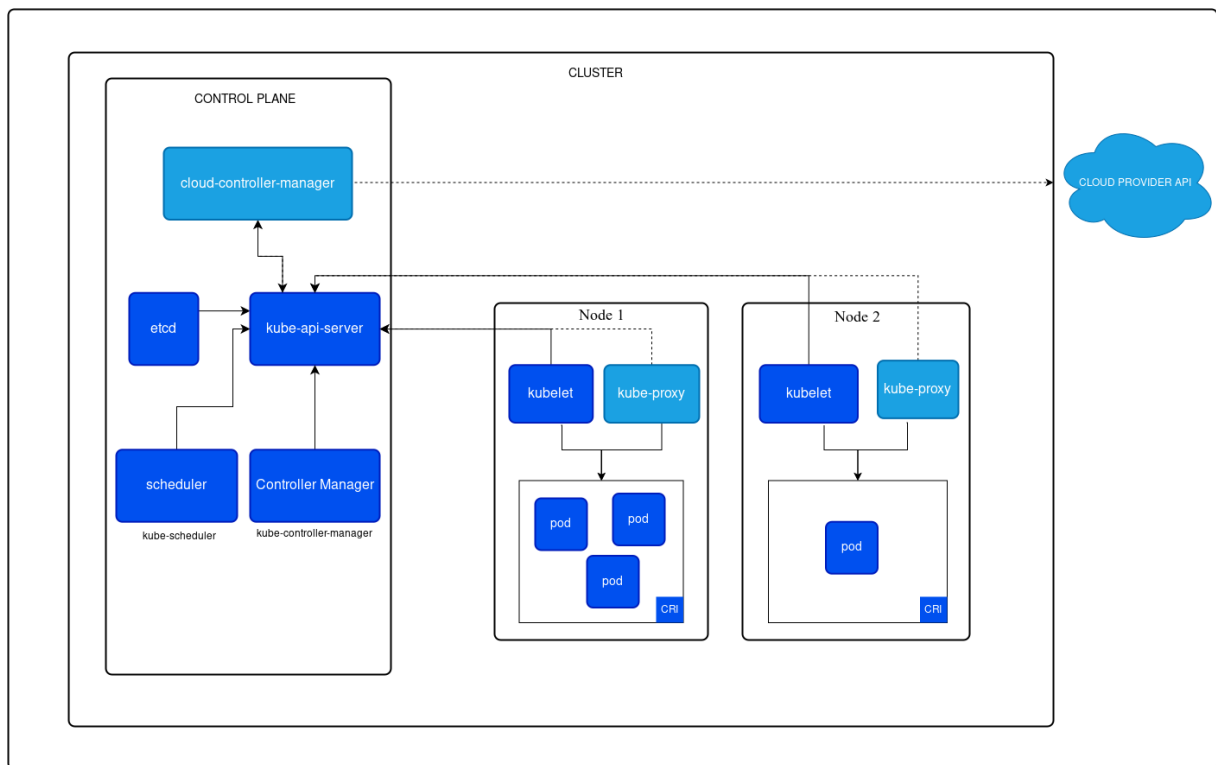
On figure 2.1 there is graphical representation of Kubernetes cluster. Not all of components shown in the figure are mandatory for Kubernetes to work correctly. At the control plane part, *cloud-controller-manager* might not be mandatory, in on-premises configurations where interacting with cloud provider 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 cluster 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 distributed 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 of 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 to communicate 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 cluster 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 really useful, when our service consists of two apps which are coupled together. 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 rather 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 has the ability to 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 and 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 is able to run regularly 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 from time to time (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's 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 specified port acts like a proxy to 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 have to 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 mainly 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].
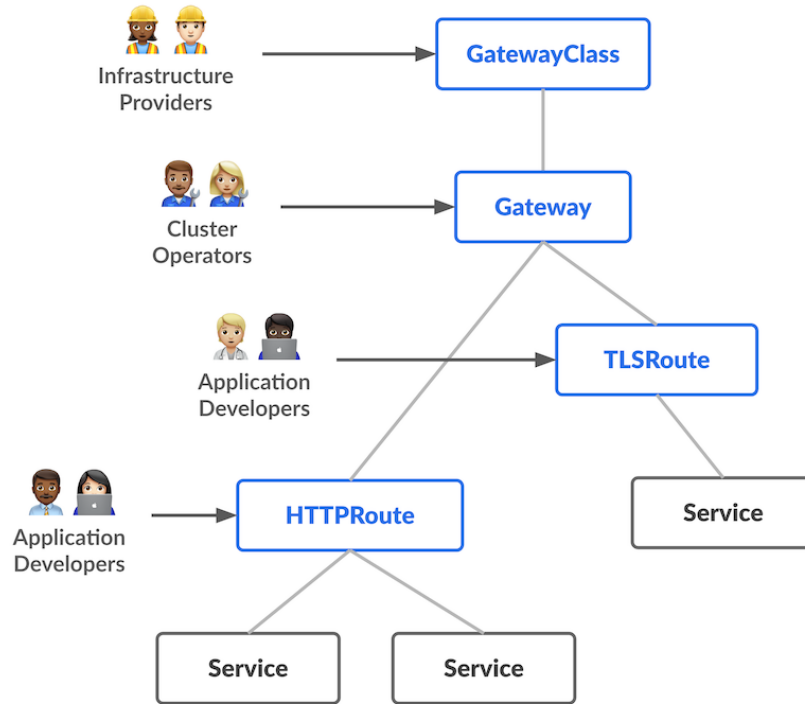

Figure 2.2: Gateway API roles-oriented resource model [21]

The model focuses on 3 separate groups of people who interact with a cluster on different levels.

On a top of figure 2.2 there are infrastructure providers, who provide GatewayClass resource. They are responsible for the overall multiple clusters infrastructure, 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 loadbalancing implementations and provides clear explenation of capabilities availabe in Kubernetes resource model. The functionality is similiar to *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 loadbalancing system is provisioned by GatewayClasss. Gateway resource defines specific instance which will handle

incoming traffic. Allows to define 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 object like service. It is able to split traffic using "weight" as a key, which represents the percentage of the total traffic to be routed. GRPCRoute is simmiliar, 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. Generally speaking, 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 main 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].
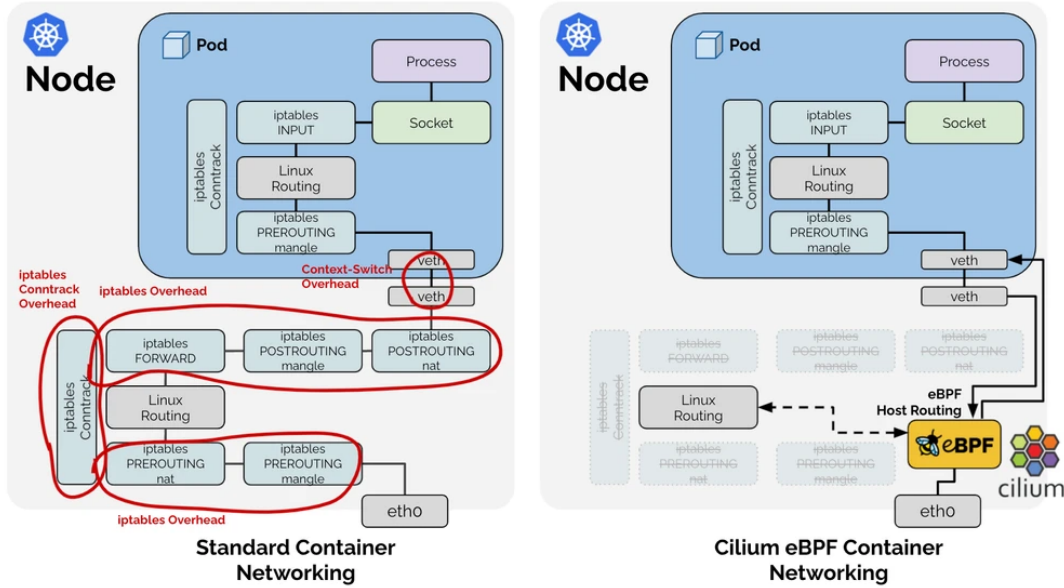

Figure 2.3: Cilium eBPF host-routing [37]

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 differ 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 main 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, fith 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 these expectation is eBPF (extended Barkeley Packet Filter) [41].

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

Egress and no egress TODO

3.1. Egress Scenario: Routing Outgoing Traffic via Egress Gateway

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

The IT departament of a financial company manages Kubernetes cluster 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 a robust security. Redirecting every request to the traffic analyzer would add unnecessary overhead to exposed applications and cause higher latency. Cluster operators decided to get 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 departament administrators started seeking for a problem and they conclude that created egress gateway is a bottleneck in their cluster. They started searching on-

line for solutions and decided to create separate gateways for each deployment of their service [42]. End users stopped complaining about poor avalaibility of services.

3.1.1. Egress Gateway in Selected CNI Plugins

Container Network Interface (CNI) plugins implement 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 resource that controls how Pods in a cluster access external services. 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 translated to the configured IP address [43].

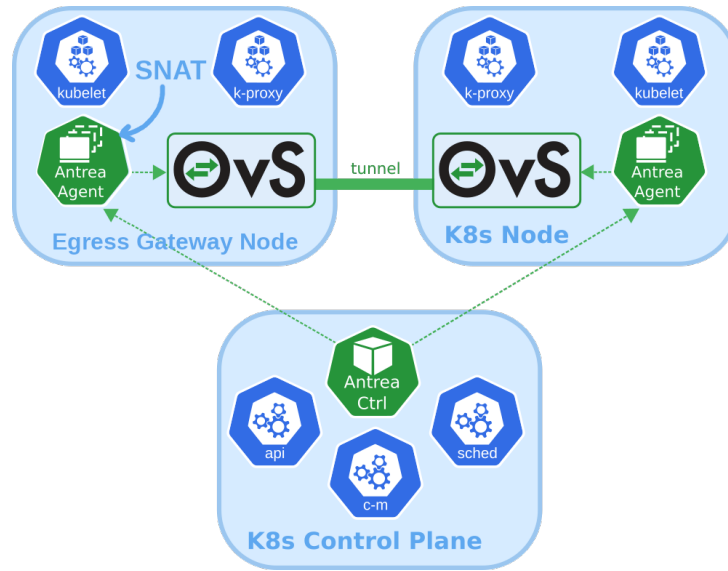Figure 3.1: Antrea Egress Architecture [43]

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 tunnelled to gateway node and Antrea Agent is performing SNAT. After translation, the next network peer that is

comuunicating with egress gateway sees its IP as a source IP instead of IP address of a pod [43] [44].

Lets 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 tunnelled

– 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 envirnoments (useful for earlier mentioned IT departament) [43].

```
1  apiVersion: crd.antrea.io/v1alpha2
2  kind: Egress
3  metadata:
4      name: egress-prod-web
5  spec:
6      appliedTo:
7      namespaceSelector:
8          matchLabels:
9          env: prod
10     podSelector:
11         matchLabels:
12         role: web
13     egressIP: 10.10.0.8
14     externalIPPool: prod-external-ip-pool
15  status:
16     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 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: 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 selecter, like previous matching labels in 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.
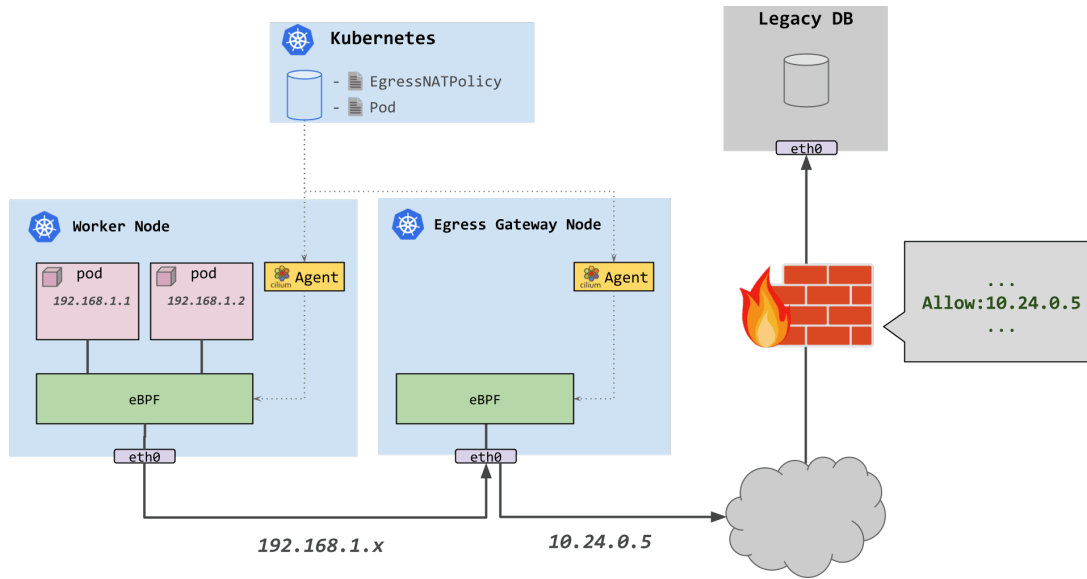
Figure 3.2: Cilium Egress Architecture [46]

Selecting an egress gateway can be done in 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 Anrea and Cilium allows 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 trafic by destination CIDR. Although cilium has more capabilities of matching egress traffic, antrea implementation allows to create fail-over node, which will route traffic if main one fails. Cilium also takes advantage by using eBPF which is designed for large-scale

clusters [34]. It is not clear which egress gateway CNI implementation 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 role-oriented approach to separate Kubernetes user/operator concerns. It is capable of teraffic splitting, header modification, or URL rewriting. The Gateway API supports key protocols, like HTTP, HTTPS, TCP, UDP and gRPC. Offering wide range of features, can be used in some different ways [47].

Canary Deployment

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

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

– initial state of app – stable version of application is served

– canary stage – new version of application is only visible for 5% of users. As some users are able to interact with new provisioned software, most common errors should be visible (if any exist).

– 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. A half of traffic is routed to new version of app. At this point performance of rolledout software is monitored.

– late stage – most of traffic (75%) is handled by new 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 enpoints 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 incomming trafic on secured enpoints, because they make want to make sure only authorized requests are handled. However the company does not have infrastructure capabilities to analyze all incomming traffic on these enpoints. As their services are HTTP based inside Kuberentes 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 usege of API Gateway (company offers RESTful APIs). Pure traffic splitting is not a case, because all incomming traffic have to be handled with 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 show how cluster infrastucture might look in this case.
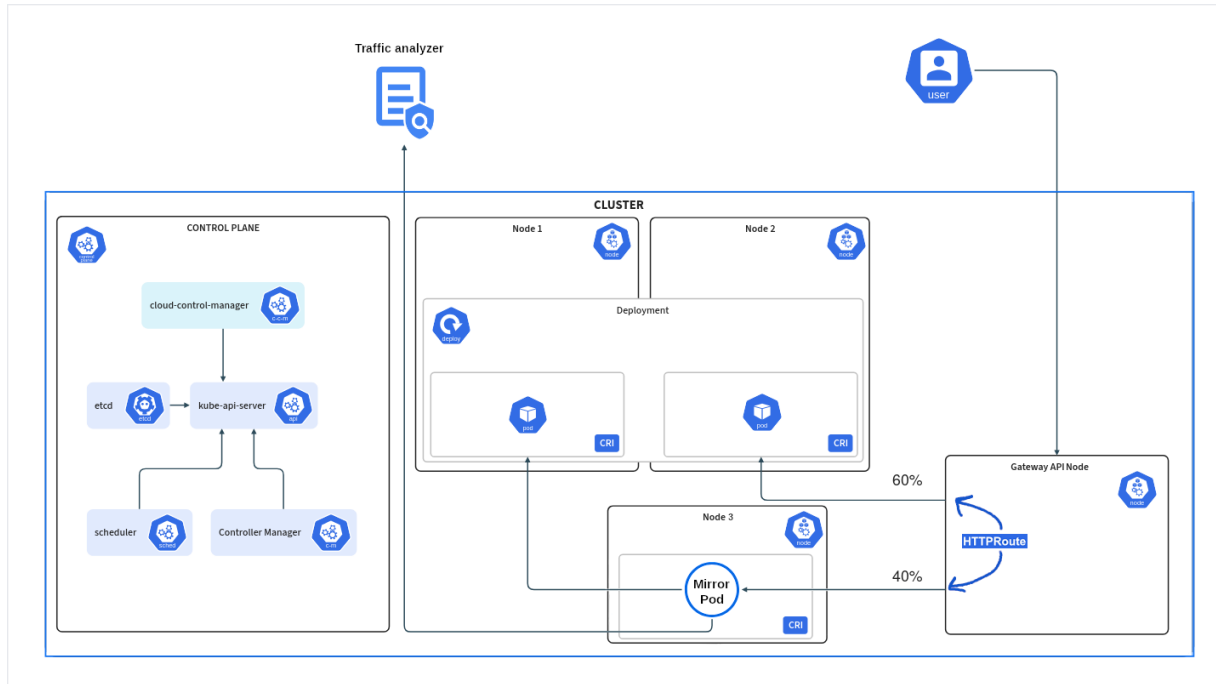
Figure 3.3: Example Kuberenetes cluster with traffic mirroring

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. For purpose of evaluating 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 tunelling among nodes is set up. On control plane node there is also NGINX Gateway API deployed as a pod, what is in different to cilium, where Gateway API will not be running as a pod.
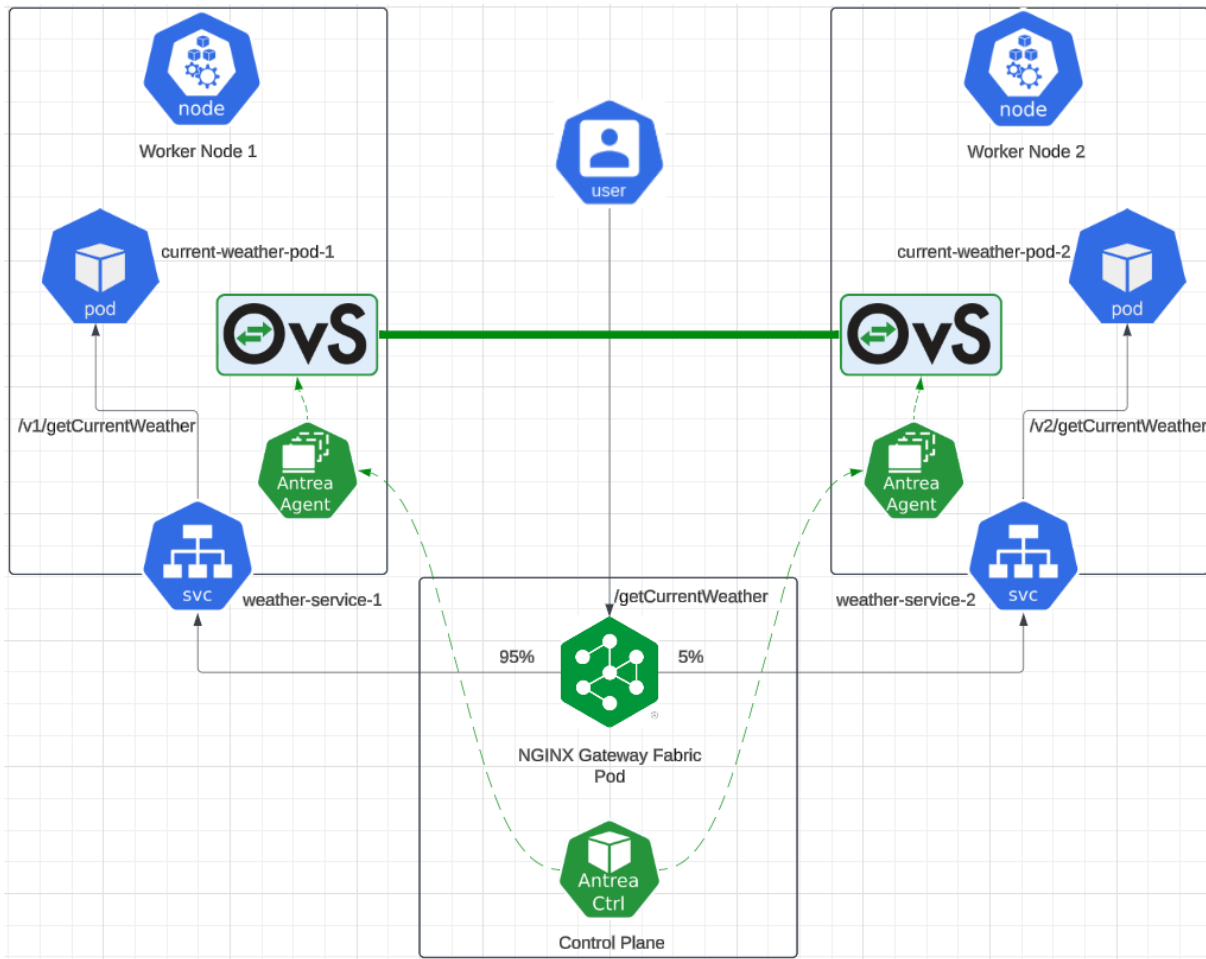
Figure 3.4: Example Kuberenetes cluster with Antrea CNI and NGINX Gateway Fabric in canary stage of canary deployment

```yaml
1  apiVersion: gateway.networking.k8s.io/v1
2  kind: HTTPRoute
3  metadata:
4      name: curren-weather-route
5  spec:
6      parentRefs:
7      - name: nginx-gw
8      rules:
9      - matches:
10          - path:
11              type: PathPrefix
12              value: /getCurrentWeather
13          backendRefs:
14          - kind: Service
15              name: current-weather-pod-1
16              port: 8080
17              weight: 95
18          - kind: Service
19              name: current-weather-pod-2
20              port: 8090
21              weight: 5
```

Listing 4: Egress resource example [43].

Cilium

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 8CPUs

– RAM: 20 GB

– Storage: 256 GB SSD

– Operating System: Fedora 40

Cloud infrastrucure 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 opensource tool which is able to 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
 1  - name: Create openstack instance and assign floating ip
 2    hosts: "{{ openstack_pool | default('localhost') }}"
 3    var_files:
 4      - ./vars/auth.yml
 5    become: yes
 6
 7    tasks:
 8      - name: Create the OpenStack instance
 9        openstack.cloud.server:
10          state: present
11          name: " {{ inventory_hostname }}"
12          key_name: "{{ key_name }}"
13          network: "{{ network_name }}"
14          auth:
15            auth_url: "{{ auth_url }}"
16            username: "{{ username }}"
17            password: "{{ password }}"
18            project_name: "{{ project_name }}"
19
20    roles:
21      - assign_floating_ip
22
```

Listing 5: Example ansible playbook [52].

```
1  [openstack_pool]
2  instance-1.example.com key_name=ansible_key network_name=my-network ansible_host=10.10.10.10
3  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. Hosts field define 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 containg, for example authentication variables required to access openstack cloud. Become is used to execute script as root user. Tasks and roles is the place, where actual script is defined. It can be defined directly in tasks, or specified by a roles, in this case will use script from ./roles/assign_floating_ip/tasks/main.yaml [51].

Iperf3

Iperf3 is a tool capable of measure 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 evalutate throughput and round trip time in egress scenario. Listings 7 and 8 shows how to run iperf three seconds measurement within localhost interface [55].

```
$ iperf3 --server
-----------------------------------------------------------
Server listening on 5201 (test #1)
-----------------------------------------------------------
Accepted connection from 127.0.0.1, port 40496
[  5] local 127.0.0.1 port 5201 connected to 127.0.0.1 port 40502
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-1.00   sec  5.55 GBytes  47.6 Gbits/sec
[  5]   1.00-2.00   sec  5.97 GBytes  51.3 Gbits/sec
[  5]   2.00-3.00   sec  5.50 GBytes  47.3 Gbits/sec
[  5]   3.00-3.00   sec  4.50 MBytes  38.7 Gbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-3.00   sec  17.0 GBytes  48.7 Gbits/sec                  receiver
```

Listing 7: Running iperf3 server command [56].

```
$ iperf3 --client 127.0.0.1 --time 3
Connecting to host 127.0.0.1, port 5201
[  5] local 127.0.0.1 port 40502 connected to 127.0.0.1 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec  5.55 GBytes  47.6 Gbits/sec    0   1.31 MBytes
[  5]   1.00-2.00   sec  5.97 GBytes  51.3 Gbits/sec    0   1.50 MBytes
[  5]   2.00-3.00   sec  5.50 GBytes  47.2 Gbits/sec    0   1.50 MBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-3.00   sec  17.0 GBytes  48.7 Gbits/sec    0               sender
[  5]   0.00-3.00   sec  17.0 GBytes  48.7 Gbits/sec                    receiver

iperf Done.
```

Listing 8: Running iperf3 client command [56].

Kind

Kind is a tool used for creating local Kuberentes 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 [57]. Listing 9 shows kind configuration used in both egress and ingress scenation on local infrastructure. One control plane node and two worker nodes are created. It also disables default CNI plugin which is essentail in this case.

```yaml
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
networking:
    disableDefaultCNI: true
nodes:
    - role: control-plane
      extraPortMappings:
        - containerPort: 80
          hostPort: 80
        - containerPort: 443
          hostPort: 443
    - role: worker
    - role: worker
```

Listing 9: Kind config used in both scenarios [58].

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. Listing 10 demonstrates k6 script used in the ingres scenario. It is written as an Ansible template. During the ingress test Ansible injects variables into the script, allowing k6 to probe the Gateway API [59].

```
1   import http from 'k6/http';
2
3   export const options = {
4     vus: "{{ number_of_vusers }}",
5     duration: "{{ test_duration }}s",
6   };
7
8   export default function () {
9       const timestamp = new Date().toISOString();
10
11      const res = http.get('http://{{ gateway_api_ip.stdout }}/echo');
12
13      const hostnameMatch = res.body.match(/Hostname:\s*(\S+)/);
14      const hostname = hostnameMatch ? hostnameMatch[1] : 'Hostname not found';
15
16      console.log(`[${timestamp}] Hostname: ${hostname}`);
17  }
```

Listing 10: Grafana k6 script used in the infress scenario [60].

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 ingres scenario to create loadbalancing serbice for Gateway API [61].

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 enpoind [62].

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 colects data by pulling from specified enpoints in configuration [63].

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 also SaaS features. Configuration files are written in a declarative language called HashiCorp Configuration Language (HCL) shown on listing 11 [64]. The script 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 preffered [65].

The Terraform workflow is made up of three stages [64]:

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 applys changes defined in write stage to the infrastructure

```
1   resource "azurerm_resource_group" "rg" {
2     location = var.resource_group_location
3     name     = "rg${var.common_infix}"
4   }
5
6   resource "azurerm_kubernetes_cluster" "k8s" {
7     location            = azurerm_resource_group.rg.location
8     name                = "cluster${var.common_infix}"
9     resource_group_name = azurerm_resource_group.rg.name
10    dns_prefix          = "dns${var.common_infix}"
11
12    identity {
13      type = "SystemAssigned"
14    }
15
16    default_node_pool {
17      name       = "agentpool"
18      vm_size    = var.vm_type
19      node_count = var.node_count
20    }
21
22    linux_profile {
23      admin_username = var.username
24
25      ssh_key {
26        key_data = azapi_resource_action.ssh_public_key_gen.output.publicKey
27      }
28    }
29
30    network_profile {
31      network_plugin    = "none"
32      load_balancer_sku = "standard"
33    }
34  }
```

Listing 11: Terraform Azure Kubernetes Service creation script [65].

4.2. Egress scenario implementation

The engress scenario compares Antrea and Cilium egress gateway implementation performance. The overall resource utilization will be evaluated by comparing the results of running the same test with and without redirecting traffic through the egress gateway. 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.

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 12:

```
1   - name: Create antrea egress scenario with egress gateway
2     hosts: "{{ target | default('localhost') }}"
3     vars_files:
4       - ./vars/antrea.yml
5       - ./vars/common.yml
6       - ./vars/egress_gateway.yml
7       - ./vars/local.yml
8
9     roles:
10      - create_kind_cluster
11      - install_antrea
12      - wait_until_antrea_installed
13      - get_ip_for_egress_node
14      - deploy_antrea_egress_gateway
15      - monitoring
16      - terraform_run_egress_iperf
17      - scrap_prometheus_data
```

Listing 12: Kind config used in both scenarios [58].

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 – informations about local infrastructure, like node names, job name for prometheus and env type

The actual playbook from listing 12 consist of eight steps to perform automate provisioning infrastructure, running test and store output:

1. create_kind_cluster – creates kind cluster using config presented on listing 9

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 intil 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 12 produces following infrastructure:
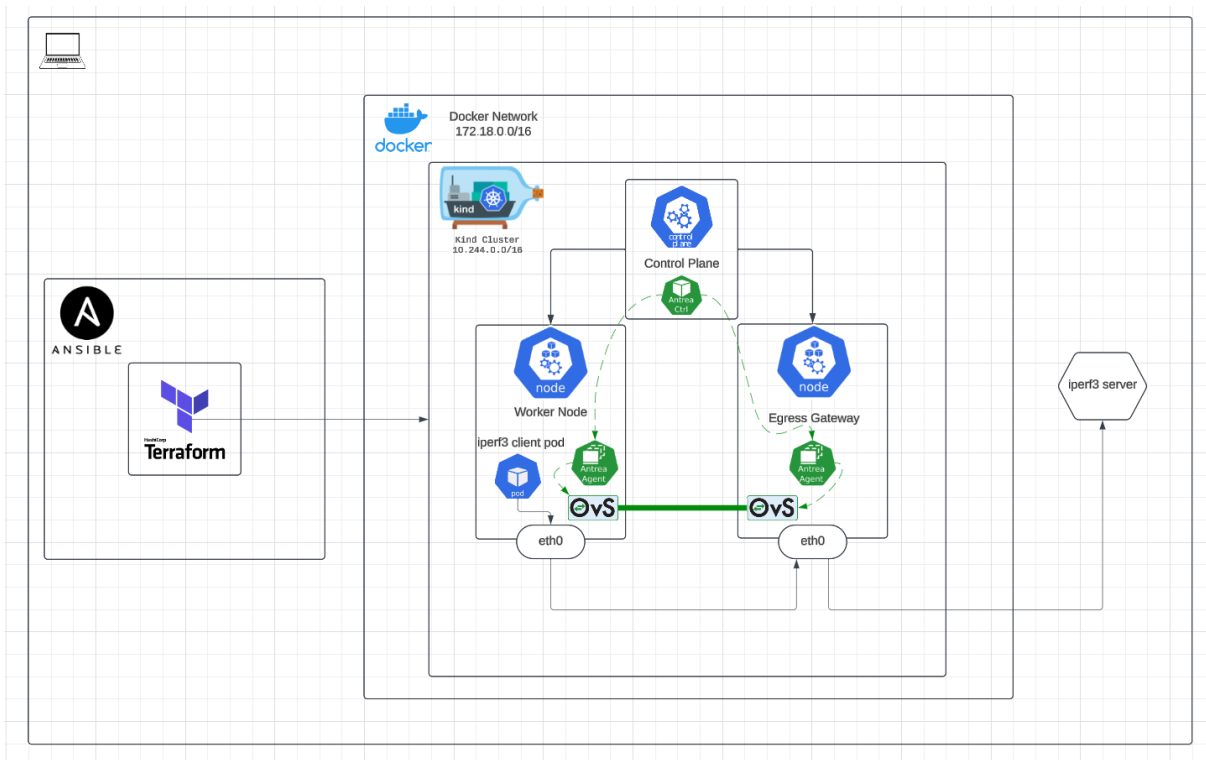

Figure 4.1: Antrea Egress Scenario infrastructure.

Kubernetes cluster visible in figure 4.1 is provisioned on personal computer using Kind, Ansible and Terraform. The cluster consists of three nodes, a control plane, a worker node and an egress gateway, which are belonging to docker network. When roles terraform_run_egress_iperf

begins and created iperf3 Pod is ready the test begins, client using TCP protocol is sending data to server, which is routed through egress gateway. Iperf3 server discovers that traffic is incomming from egress gateway node (it sees IP address of Egress Gateway Node), because traffic from the Pod is SNATed. After the measurement is over, unformatted data is downloaded from prometheus and saved to csv file

4.2.2. Cilium

4.3. Inress scenario implementation

4.3.1. Cloud deployment

4.3.2. Local deployment

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] Jon Dugan, Seth Elliott, Bruce A. Mah, Jeff Poskanzer, and Kaustubh Prabhu. What is iPerf / iPerf3 ? `https://iperf.fr/`. Accessed, 17-Dec-2024.

[56] Jon Dugan, Seth Elliott, Bruce A. Mah, Jeff Poskanzer, and Kaustubh Prabhu. iPerf 3 user documentation. `https://iperf.fr/iperf-doc.php`. Accessed, 17-Dec-2024.

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

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

[59] Inc. Red Hat. Templating (Jinja2). `https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_templating.html`. Accessed, 18-Dec-2024.

[60] Grafana Labs. k6/http. `https://grafana.com/docs/k6/latest/javascript-api/k6-http/`. Accessed, 18-Dec-2024.

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

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

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

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

[65] 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.