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, comparation 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].

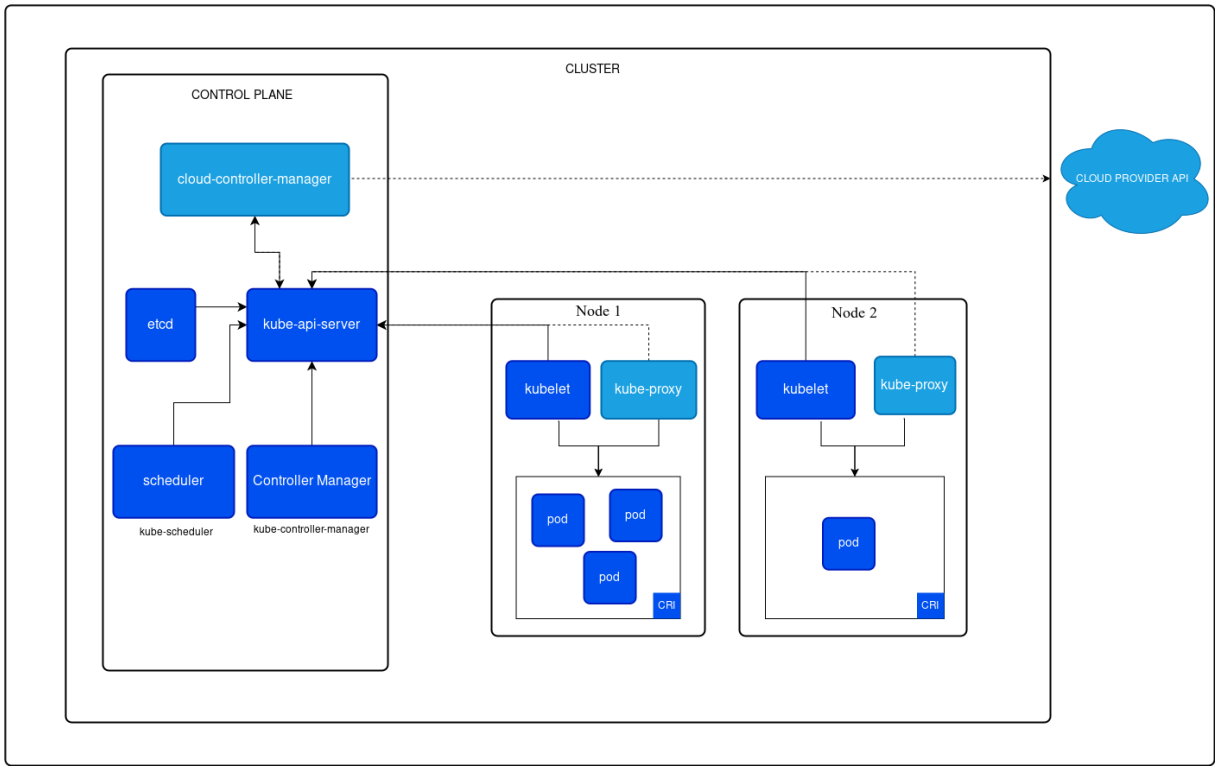

Figure 2.1: Kubernetes Cluster Architecture [4]

On figure 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 Gatewa API resource model is role-oriented [21].

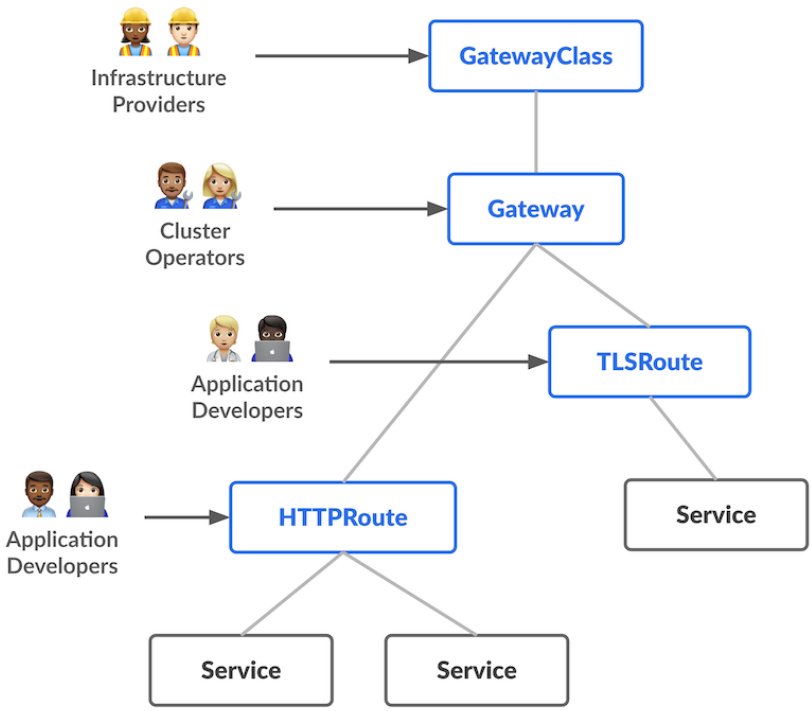

Figure 2.2: Gateway API roles-oriented resource model [21]

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].

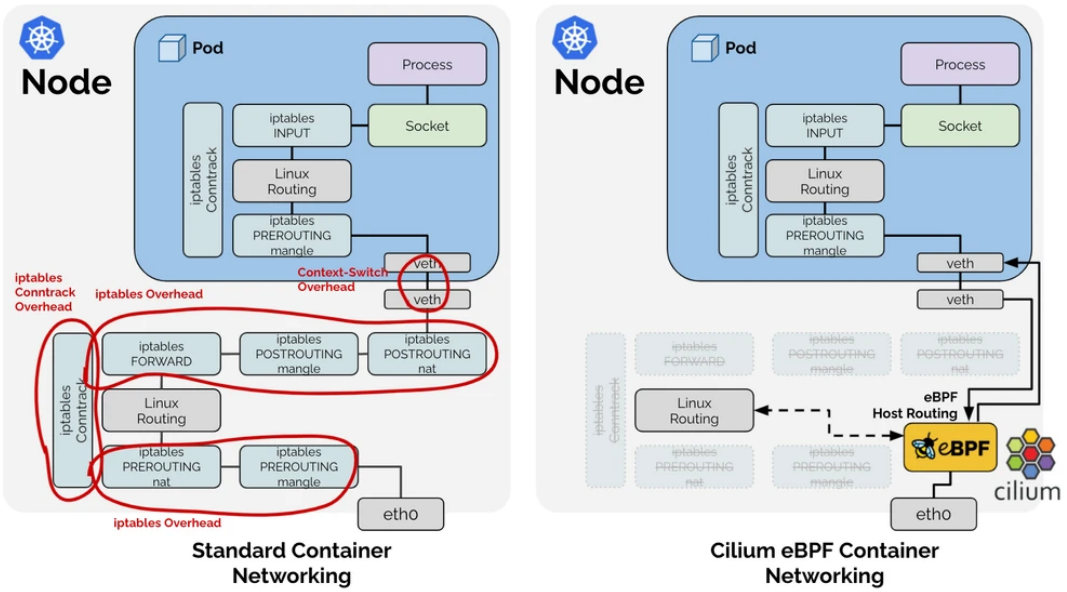

Figure 2.3: Cilium eBPF host-routing [37]

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].

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.