Universidade Estadual de Campinas
FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO

Arthur de Jesus Simas

eZtunnel: An eBPF-based Traffic Acceleration Mechanism for Cloud-Native Infrastructures

eZtunnel: Um Mecanismo de Aceleração de Tráfego Baseado em eBPF para Infraestruturas Cloud-Native

Campinas
2025

Arthur de Jesus Simas

eZtunnel: An eBPF-based Traffic Acceleration Mechanism for Cloud-Native Infrastructures

eZtunnel: Um Mecanismo de Aceleração de Tráfego Baseado em eBPF para Infraestruturas Cloud-Native

Dissertation presented to the Faculty of Electrical and Computer Engineering of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Electrical Engineering, in the area of Computer Engineering.

Dissertação apresentada à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas como parte dos requisitos exigidos para a obtenção do título de Mestre em Engenharia Elétrica, na Área de Engenharia de Computação.

Supervisor: Prof. Dr. Christian Rodolfo Esteve Rothenberg

Este exemplar corresponde à versão final da tese defendida pelo aluno Arthur de Jesus Simas, e orientada pelo Prof. Dr. Christian Rodolfo Esteve Rothenberg

Campinas
2025

Ficha catalográfica
Universidade Estadual de Campinas (UNICAMP)
Biblioteca da Área de Engenharia e Arquitetura
Rose Meire da Silva - CRB 8/5974

Si42e

Simas, Arthur de Jesus, 2000-
eZtunnel: an eBPF-based traffic acceleration mechanism for cloud-native infrastructures / Arthur de Jesus Simas. – Campinas, SP : [s.n.], 2025.

Orientador: Christian Rodolfo Esteve Rothenberg.
Dissertação (mestrado) – Universidade Estadual de Campinas (UNICAMP), Faculdade de Engenharia Elétrica e de Computação.

1. Computação em nuvem. 2. Arquitetura de redes de computador. 3. Computation offloading. 4. Qualidade de Serviço (Redes de Computadores). I. Rothenberg, Christian Rodolfo Esteve, 1982-. II. Universidade Estadual de Campinas (UNICAMP). Faculdade de Engenharia Elétrica e de Computação. III. Título.

Informações complementares

Título em outro idioma: eZtunnel: um mecanismo de aceleração de tráfego baseado em eBPF para infraestruturas cloud-native
Palavras-chave em inglês:
Cloud computing
Network architecture
Computation offloading
Quality of service
Área de concentração: Engenharia de Computação
Titulação: Mestre em Engenharia Elétrica
Banca examinadora:
Christian Rodolfo Esteve Rothenberg [Orientador]
Fabio Luciano Verdi
Jéferson Campos Nobre
Data de defesa: 16-01-2025
Programa de Pós-Graduação: Engenharia Elétrica

Objetivos de Desenvolvimento Sustentável (ODS)
Não se aplica

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0001-7705-4929
- Currículo Lattes do autor: http://lattes.cnpq.br/7361619387683682

Prof. Dr. Christian Rodolfo Esteve Rothenberg (FEEC/UNICAMP, Presidente)

Prof. Dr. Fabio Luciano Verdi (UFSCar – *Campus* de Sorocaba)

Prof. Dr. Jéferson Campos Nobre (Instituto de Informática – Universidade Federal do Rio Grande do Sul)

A ata de defesa, com as respectivas assinaturas dos membros da Comissão Julgadora, encontra-se no SIGA (Sistema de Fluxo de Dissertação/Tese) e na Secretaria de Pós-Graduação da Faculdade de Engenharia Elétrica e de Computação.

Acknowledgments (Agradecimentos)

Na tentativa de utilizar palavras adequadas para expressar gratidão àqueles que, de alguma forma, contribuíram com o meu progresso...
Agradeço,

– aos membros da banca examinadora pelos valorosos comentários, sugestões e contribuições, que ajudaram a melhorar a qualidade deste manuscrito;
– ao Prof. Christian pela orientação, apoio e ajuda, generosamente oferecidos desde antes do início 'oficial' do mestrado;
– ao Prof. Tiago e ao Prof. Wesley por me ajudarem a engatinhar e dar os primeiros passos na pesquisa acadêmica;
– ao Prof. Fábio pelo apoio desde o princípio da minha jornada no mestrado;
– à Profª. Sahudy por me cobrar excelência e me aconselhar nas minhas decisões e indecisões acadêmicas, e na vida;
– aos meus gatos, Amora e Forrest, pelo afeto e companhia durante os prolongados períodos de estudo, e por *meow meow meow meow meow*;
– aos meus pais, Cristina e Vanderlei, à minha irmã, Maria Eduarda, e à minha vó, Adelina, pelo apoio, compreensão e encorajamento em tudo o que faço, pelo cuidado e educação, pela construção da minha bússola moral, por sempre estarem presentes, e por me proporcionarem um porto-seguro, mesmo que 'tudo' dê errado;
– à minha tia, Carmem, pelo incentivo e companhia nos estudos;
– ao meu tio, Jorge, por despertar a minha curiosidade na computação e eletrônica;
– aos meus amigos da graduação, Caio, Enzo, Gabriel (Viana), Guilherme, Luiz, Rafael e Tales, os 'IndianCucks', pelos anos de companheirismo, divertimento, e aturação (na faculdade e, agora, nos encontros semestrais);
– ao meu outro amigo da graduação, Gabriel (Givigi), pela extensão aos anteriores— apesar de não fazer parte do mesmo grupo—, e pelos sábios conselhos de vida;
– ao meu outro amigo, também da graduação, João, pela ajuda a trilhar a carreira acadêmica e conversas frutíferas, além da amizade, é claro;
– aos meus amigos do colégio, Bianca, Clara, Cintia, Sarah, Silvio e Tammy, pelas risadas e por aturarem os momentos 'desne';
– This study was financed, in part, by the *São Paulo Research Foundation* (FAPESP), Brasil. Process Number #2023/05222-3 and #2021/00199-8 (CPE SMARTNESS);
– This study was financed in part by the *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior* – Brasil (CAPES) – Finance Code 001.

We all need people who will give us feedback. That's how we improve.

– Bill Gates

Resumo

Aplicações nativas para a nuvem, caracterizadas por sua escalabilidade, resiliência e flexibilidade, adotam uma arquitetura de microsserviços para decompor aplicações em serviços menores e independentes. Essa arquitetura, embora ofereça benefícios significativos, introduz desafios na comunicação entre serviços, exigindo o uso de *frameworks* avançados de orquestração e comunicação, como Kubernetes e Istio, respectivamente. No entanto, a complexidade dessas tecnologias impõe uma sobrecarga substancial na infraestrutura, que introduzem um maior caminho no processamento dos pacotes.

Este trabalho identifica os gargalos de desempenho decorrentes do uso extensivo de *service meshes*, destacando o problema crítico de sobrecarga da CPU devido a tarefas relacionadas à rede. Para enfrentar esses desafios, propomos o eZtunnel, uma técnica de offloading transparente para permitir comunicações eficientes em *service meshes*. Esta proposta utiliza o filtro de pacotes estendido (do inglês, *extended Berkeley Packet Filter* – eBPF) como tecnologia habilitadora para abordar e mitigar o problema.

Os experimentos mostram que a solução proposta pode otimizar as métricas de rede de comunicação intra-nó, tal como redução de *Flow Completion Time* (FCT) em 41,2%, latência em 42,0%, e aumento de vazão em 68,5%. O consumo de memória foi baixo, alcançando até 60,5 MB. O uso de CPU foi variável entre +29,2 e -23,2%. Através dessa abordagem, a pesquisa busca desbloquear o potencial das aplicações nativas para a nuvem, garantindo que os avanços arquiteturais se traduzam em benefícios significativos.

Palavras-chave Cloud Computing, Service Mesh, Kubernetes, eBPF, Computation Offloading

Abstract

Cloud-native applications, characterized by their scalability, resilience, and flexibility, adopt a microservices architecture to decompose applications into smaller, independently manageable services. This architecture, while offering significant benefits, introduces challenges in service-to-service communication, requiring the use of advanced orchestration and communication frameworks such as Kubernetes and Istio, respectively. However, the complexity of these technologies impose substantial overhead on the underlying infrastructure, introducing longer packet processing paths.

This work identifies the performance bottlenecks arising from the extensive use of service meshes, highlighting the critical issue of CPU overload due to networking-related tasks. To address these challenges, we propose eZtunnel, a transparent offloading technique to allow efficient communications in service meshes. This proposal leverages *extended Berkeley Packet Filter* (eBPF) as the enabler technology to address and mitigate the problem.

Experiments show that the proposed solution can improve intra-node networking metrics, such as reduction in average Flow Completion Time (FCT) by 41.2%, latency by 42.0%, and increase in throughput by 68.5%. Memory footprint was small, reaching at most 60.5 MB. CPU usage was variable between +29.2% and -23.2%. Through this approach, the research aims to unlock the full potential of cloud-native applications, ensuring that the architectural advancements translate into relevant benefits.

Keywords Cloud Computing, Service Mesh, Kubernetes, eBPF, Computation Offloading

List of Figures

List of Tables

List of Listings

List of Acronyms

ASIC Application-Specific Integrated Circuit. 38–40, 44

cgroup control group. 46, 48, 53

CI/CD Continuous Integration and Continuous Delivery. 21, 24, 25

CNCF Cloud Native Computing Foundation. 23

CNI Container Network Interface. 25, 27, 32, 33, 38, 40, 41

DevOps Development Operations. 21

DPDK Data Plane Development Kit. 37, 38, 42, 44

DPU Data Processing Unit. 37–39, 42, 44

eBPF extended Berkeley Packet Filter. 18–20, 30, 32, 34–37, 40–49, 51–54, 56, 59, 62, 72–76

FCT Flow Completion Time. 9, 11, 18, 41, 56, 59, 62–65, 72, 74–76

FPGA Field Programmable Gate Array. 40

gRPC Google Remote Procedure Calls. 33, 34

IPDK Infrastructure Programmer Development Kit. 38, 42, 44

IPU Infrastructure Processing Unit. 39

mTLS mutual Transport Layer Security. 22, 29

NAT Network Address Translation. 26

NIC Network Interface Card. 40

OPS Operations per Second. 9, 11, 57, 66, 67, 73

OVS Open vSwitch. 33, 40

P4 Programming Protocol-independent Packet Processors. 37–39, 44

RPS Requests per Second. 9, 11, 57, 65, 66, 73

SPDK Storage Performance Development Kit. 38

ToR Switch Top-of-Rack Switch. 39

VM Virtual Machine. 58

XDP eXpress Data Path. 35, 36, 40, 42

Contents

Chapter 1

Introduction

In the rapidly evolving landscape of cloud computing, the adoption of cloud-native technologies represents a fundamental shift in how applications are developed, deployed, and managed [1], [2]. These applications are commonly based on microservices architectures to make the most of cloud environments' scalability, resilience, and flexibility.

To manage a large number of microservices, Kubernetes and a service mesh are often used to, respectively, (*i.*) orchestrate containerized applications across diverse environments [3] and (*ii.*) provide advanced inter-service communication features, such as service discovery, load balancing, encryption, authorization, and observability [4].

Service meshes often implement the sidecar deployment mode, which injects a proxy beside each service. This mode dramatically degrades performance, particularly when the Kubernetes cluster orchestrates many services [5], [6]. To overcome this problem, some service mesh proposals introduce the concept of a per-node shared agent [7], which instead of a pod per sidecar, it supports a node per agent (a 'sidecar' equivalent), and therefore, multiple pods per agent.

However, both architectures introduce performance penalties due to the poor usage of kernel capabilities and inefficiencies of the traditional kernel network stack, which limit their ability to fully optimize performance at scale. This overhead may result in lower throughput, higher latency, and ultimately a degradation in end users' quality of service [8], [9].

This leads us to the crux of the problem: while cloud-native technologies, microservices, container orchestration tools, and service meshes have revolutionized application development and deployment, they inadvertently introduce performance bottlenecks [5].

The root cause of this issue is the excessive traversals of the regular packet transmission path provided by the kernel network stack, resulting in inefficient resource utilization and diminished application performance [6].

Given that, it is clear that there is a need for an optimization strategy to alleviate the burden on the host CPU, particularly networking tasks, using more efficient mechanisms. Such an optimization strategy would optimize resource utilization, enhance application performance, and unlock the full potential of cloud-native technologies in microservices-based environments.

To this end, we propose eZtunnel[1], a transparent offloading technique to enable efficient communications in service meshes, independent of the proxy deployment mode (sidecar or per-node shared agent). Our proposal leverages the extended Berkeley Packet Filter (eBPF) as the key technology to address and mitigate the problem. This approach involves deploying eBPF programs to intercept and redirect packets at the kernel level, bypassing the kernel network stack.

Our experiments show that, among four different service mesh variations—no service mesh, `cilium`, `istio-ambient`, and `istio-sidecar`—the proposal can improve intra-node networking metrics, such as reduction in average Flow Completion Time (FCT) by 41.2%, latency by 42.0%, and increase in throughput by 68.5%. Memory footprint was small, reaching at most 60.5 MB, while CPU usage was variable between +29.2% and -23.2%.

By addressing this fundamental problem, we contribute to create more efficient, scalable, and performance-oriented cloud-native applications, ensuring that the architectural innovations brought about by microservices and service meshes translate into realistic benefits for organizations and their end-users.

1.1 Research Outline

Research Problem. Current cloud-native environments use a container orchestration software (*e.g.*, Kubernetes) and a service mesh (*e.g.*, Istio) to add more networking capabilities on top of the cluster. Service meshes can be deployed in two modes, sidecar and a per-node shared agent. Both deployment modes present increased overheads due to a longer communication path. Although the overhead imposed by the sidecar deployment mode have been explored and a solution is presented in Yang *et al.* [6], the per-node shared agent is still unsolved.

Objectives. The main objective of this work is to present an overview of how service meshes work, their inherent overhead problem, and the implementation of a transparent offloading mechanism to enable efficient networking in service meshes.

[1] **eZtunnel = eBPF-based + ztunnel**. Originally, *eZtunnel* aimed to improve *ztunnel*, an Istio Ambient component. Later, it was discovered that the solution could also improve other service mesh architectures, not only Istio Ambient. Then, the name no longer makes sense and is kept only for historical reasons.

Contributions. The contributions of this dissertation can be summarized as follows:

- Overview of cloud-native applications, including container orchestration and service meshes, presenting their respective problems and alternatives;
- Investigation of the state-of-the-art service mesh offloading to find open gaps and improvement opportunities;
- A transparent offloading method based on eBPF to improve microservice communication in diverse service mesh environments;
- Experimental evaluation of our implementation on a testbed deployed with Kubernetes and Istio to assess our approach's potential to alleviate service mesh overheads;
- Release of all source code and datasets to ensure reproducibility according to open research practices.

Moreover, the following publication results were achieved:

- **Arthur J Simas**, Christian Esteve Rothenberg, Gergely Pongrácz, "eSeMeshA: eBPF-based Service Mesh Acceleration for Cloud-Native Infrastructures," in *IEEE 11th International Conference on Network Softwarization* (IEEE NetSoft), 2025.
- **Arthur J Simas**, Fabricio Rodriguez Cesen, Christian Esteve Rothenberg, "eZtunnel: Leveraging eBPF to Transparently Offload Service Mesh Data Plane Networking," in *IEEE 13th International Conference on Cloud Networking* (IEEE CloudNet), 2024.
- **Arthur J Simas**, Christian E Rothenberg, "Network Performance: Evaluating Offloading Strategies in Modern Programmable Infrastructures," in *XV Encontro dos Alunos e Docentes do Departamento de Engenharia de Computação e Automação Industrial*, 2023.
- **Arthur Simas**, William Melo, Leonardo Guimarães, Christian Rothenberg, "PathsViewer: Uma Interface para Exploração de Dados Espaço-Temporais," in *Anais Estendidos do XLI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, 2023.

Methodology. To accomplish the objectives, we reviewed the literature to identify existing offloading techniques, both academic research and industry practices in this field. Then, we implemented a prototype using the selected technology, setup a benchmark testbed and assessed the prototype's effectiveness in reducing the overhead and improving the selected metrics [10]–[12].

1.2 Organization

The remainder of this manuscript is organized as follows:

Background on Cloud-Native Environments (§2). Introduces Kubernetes networking and service meshes, highlighting their limitations and the need for optimization strategies.

Offloading Technologies Review (§3). Explores existing offloading techniques, from software frameworks to hardware-based solutions, and positions the proposed eZtunnel approach.

eZtunnel: Design and Implementation (§4). Details the eBPF-based solution to improve intra-node networking, bypassing the Linux network stack to optimize network-related metrics and improve resource utilization.

Experimental Evaluation (§5). Presents benchmarks on diverse workloads, demonstrating significant performance gains across most metrics, with some areas for refinement.

Conclusions and Future Work (§6). Summarizes findings, emphasizes the impact of the optimization, and suggests directions for extending the work.

Chapter 2

Background on Cloud-Native Environments

As applications grow in scale and become more complex, traditional approaches to manage their lifecycle are often insufficient. To address these challenges, modern applications are designed using the cloud-native architecture to make the most of cloud environments, such as scalability, resilience, and flexibility [1], [13], [14].

Cloud-native architecture is composed of these core pillars: **Microservices**, represented in Figure 2.1, which decompose applications into smaller, loosely coupled services, allowing for easier updates and adaptability; **Containers and Orchestration**, where containers encapsulate those microservices for consistent deployment across environments, while orchestrators manage their lifecycle; **Development Operations (DevOps)**, which encourages collaboration between development and operations teams to automate infrastructure and software delivery, improving deployment speed and reliability; and **Continuous Integration and Continuous Delivery (CI/CD)**, which streamlines automated testing, building, and deployment to ensure frequent releases.

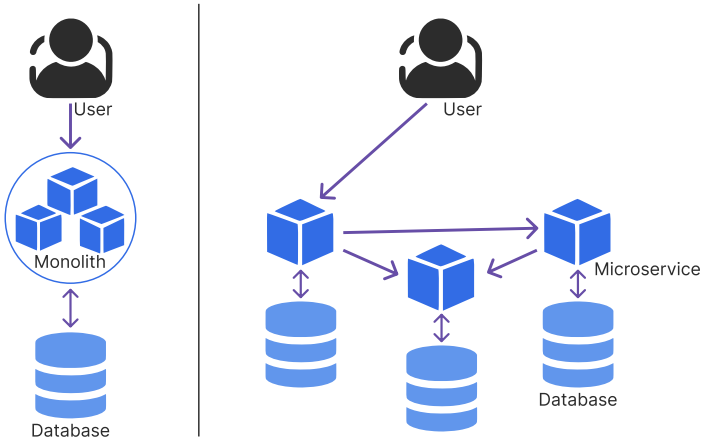


Figure 2.1: Difference between a monolithic application (left) and microservices architecture (right).

Among these pillars, orchestration is particularly special because it manages the containerized microservices, ensuring they scale efficiently, recover from failures, communicate reliably, and many other tasks. Effective orchestration addresses challenges such as resource allocation, load balancing, and service discovery, which are crucial for maintaining performance in distributed systems.

On top of that, cluster operators usually place an abstraction layer, the service mesh, to extend the capabilities of the orchestrator to provide advanced networking features. A service mesh abstracts the communication logic away from the application code, offering traffic management, observability, and security. The service mesh transparently handles tasks like load balancing, retries, circuit breaking, mutual Transport Layer Security (mTLS), and monitoring.

In this chapter, we will present the core concepts of container orchestration using Kubernetes, Kubernetes networking, service meshes, and optimization opportunities within this context.

2.1 Container Orchestration with Kubernetes

Containers package an application and its dependencies into a single unit that can run consistently across different computing environments. This abstraction ensures that the application behaves the same independently of the environment it is running, whether a developer's laptop, a test environment, or in production. Containers are lightweight, portable, and can be quickly created and destroyed, making them ideal for microservices-based architectures [15].

Manually managing these containers, especially as applications scale, becomes impractical over time. This is where container orchestration comes into play. Orchestration automates tasks required to deploy, manage, and maintain containers in a distributed environment, ensuring consistency, availability, and efficiency.

Container orchestration addresses the need for deployment automation, scaling, and operation of containerized applications. It allows developers and operators to manage the lifecycle of a large set of containers efficiently, ensuring that applications remain resilient and adaptable to varying demands.

Container orchestration platforms perform several tasks to manage the deployment and operation of containers. These tasks include[1]:

Service Discovery and Networking. In a microservices architecture, services need to communicate with each other reliably. Orchestration platforms provide mechanisms for service discovery, enabling containers to find and connect with other services automatically, wether containers are placed within the same node and across nodes.

[1] https://kubernetes.io/docs/concepts/overview/

Scaling and Load Balancing. Modern applications often experience fluctuating demand, requiring dynamic scaling of resources. Orchestrators handle horizontal scaling by automatically increasing or decreasing the number of container instances based on metrics such as CPU usage or request rates. With help of service discovery, load can be balanced across multiple containers to optimize resource utilization and prevent any single container from becoming a bottleneck.

Rolling Updates and Rollbacks. To minimize disruption during application updates, orchestrators support rolling updates, where new versions of a container are gradually deployed while the old versions are phased out. If issues are detected during the update, orchestrators can perform rollbacks to revert to a previous stable state.

Automated Scheduling. Orchestrators simplify the deployment of containers by automating the process of scheduling and launching containers on available nodes within a cluster. This ensures that containers are deployed to appropriate hosts based on resource availability and defined policies.

Resource Management. Efficient allocation of resources, such as CPU and memory, is essential to maintaining performance and stability in distributed systems. Orchestrators ensure that containers are assigned the necessary resources and that these resources are best used across nodes in the cluster.

Health Management and Self-Healing. To ensure high availability, orchestration platforms continuously monitor the health of containers. If a container fails, the orchestrator can automatically restart the container, reschedule it on a different node, or replace the container if necessary. This self-healing capability minimizes downtime and improves the resilience of the application.

Configuration and Secret Management. Managing configuration data and secrets (such as API keys or credentials) is simplified by orchestrators. They provide mechanisms to inject configuration data and secrets into containers securely, ensuring that sensitive information is not hard-coded into application images.

Several platforms are available for container orchestration, each offering a set of features to manage containerized workloads. The most widely adopted orchestration platform is Kubernetes, an open-source software initially developed by Google and now maintained by the Cloud Native Computing Foundation (CNCF). Kubernetes has become the *de facto* standard for container orchestration due to its robust feature set, large ecosystem, and strong community support [1], [16].

Kubernetes can be deployed in two ways: managed and unmanaged. In the **unmanaged Kubernetes** setup, operators are responsible for installing, configuring, and maintaining the Kubernetes control plane and worker nodes. This approach provides full control and customization but requires significant operational effort and expertise. In contrast, **managed Kubernetes** offerings simplify this process by outsourcing the management of the Kubernetes control plane to cloud providers. Examples of managed Kubernetes include Amazon Elastic Kubernetes Service (EKS), Azure Kubernetes Service (AKS), Google Kubernetes Engine (GKE), and Oracle Container Engine for Kubernetes (OKE).

Aside from Kubernetes, other orchestration platforms include:

- **Red Hat OpenShift** is a Kubernetes-based platform that provides additional enterprise features such as integrated CI/CD pipelines, developer tools, and enhanced security;
- **Docker Swarm** is an orchestration tool developed by Docker. It is easier to set up compared to Kubernetes, but offers fewer features and is better suited for small deployments;
- **Apache Mesos** is designed for managing large-scale distributed systems and complex scheduling needs. When used with Marathon—a container orchestration framework for Mesos—, it can manage container deployments effectively;
- Developed by HashiCorp, **Nomad** is a lightweight and flexible orchestrator that supports both containers and non-containerized workloads. It is designed for simplicity and can be deployed in a single binary;
- **Proprietary container orchestration services**, such as **Amazon Elastic Container Service** (ECS), and **Azure Container Instances** (ACI), offer managed environments for running containers without relying on Kubernetes. These services integrate tightly with their respective cloud ecosystems and simplify container management by abstracting the complexities of orchestration, almost similar to managed Kubernetes offerings.

In cloud-native environments, where applications are built to take advantage of the scalability and flexibility of the cloud, container orchestration simplifies the management and automation of these workloads. Orchestration ensures that containerized applications are deployed consistently across different cloud providers, on-premises data centers, and hybrid cloud environments. It provides the automation needed to manage the complexity of distributed systems, enabling applications to scale seamlessly, recover from failures, and adapt to changing traffic.

Furthermore, container orchestration platforms integrate with other cloud-native tools and practices, such as CI/CD pipelines, monitoring systems, and service meshes. This integration enables developers and operators to implement modern software delivery practices, such as continuous deployment, automated testing, and real-time observability, enhancing the overall efficiency and reliability of applications.

This piece of software is a foundational technology in cloud-native environments. It addresses the challenges of deploying and managing containerized applications at scale, ensuring that these applications remain resilient, scalable, and adaptable. As the adoption of cloud-native architectures continues to grow, effective orchestrators will remain essential for delivering reliable and efficient software systems.

2.2 Kubernetes Networking

Building on the concepts of container orchestration, Kubernetes networking is a set of components to facilitate the communication between containerized applications part of a Kubernetes cluster. It ensures that applications running in containers can interact within the cluster and with external services. The networking model in Kubernetes is designed to address the challenges of managing distributed applications by providing a flexible yet consistent communication mechanism.

Kubernetes supports different networking paths for handling communication within and outside the cluster[2]:

- **Container-to-container**: Direct communication between containers within the same pod, provided by the `localhost` networking;
- **Pod-to-pod**: Direct communication between pods within the cluster, facilitated by the Container Network Interface (CNI) plugin;
- **Pod-to-service**: Communication between pods and services, enabled by the Service abstraction and kube-proxy;
- **External-to-internal**: External users accessing services through NodePort, LoadBalancer, or Ingress.

Within a Kubernetes cluster, pod-to-pod communication can be classified into two types: inter-node and intra-node, each having unique implications for network design, routing, and pod placement. As illustrated in Figure 2.2, inter-node communication occurs when pods reside on different nodes. In this case, communication between Pod 1 on Node A and Pod 2 on Node B is an example of inter-node communication.

[2] https://kubernetes.io/docs/concepts/cluster-administration/networking/

This exchange involves network traffic that must traverse the underlying network infrastructure between the nodes. In contrast, intra-node communication takes place between pods on the same node. For example, the communication between Pod 2 and 3, which are both hosted on Node B, is an example of intra-node communication. This type of communication typically benefits from lower latency and higher throughput since the data exchange remains within the same physical or virtual machine.

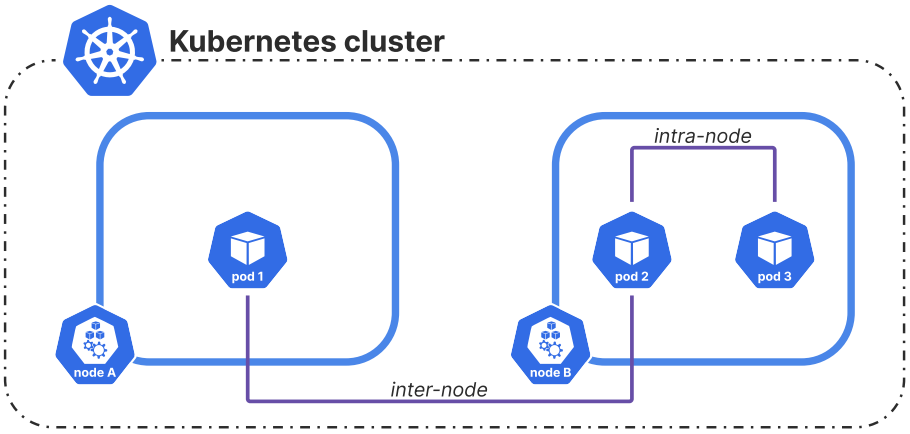


Figure 2.2: Diagram of a Kubernetes cluster consisting of two nodes. Node A contains Pod 1, while Node B hosts Pods 2 and 3. Communication between Pods 1 and 2 is classified as inter-node, while communication between Pods 2 and 3 is intra-node.

Additionally, the Kubernetes defines a networking model with several fundamental principles to simplify the communication between containers, pods, and services[3]:

- **Every pod has a unique IP address**: Each pod in a Kubernetes cluster is assigned a unique IP address. This allows for direct communication between pods without the need for Network Address Translation (NAT);
- **Flat network structure**: All pods within a cluster can communicate with each other directly, as if they are on a single, flat network. This simplifies service-to-service communication and reduces the complexity of network management;
- **Container-to-container communication**: Containers within the same pod share the same network namespace and IP address. They can communicate with each other through `localhost` using different port numbers;
- **Service abstraction**: Kubernetes uses services to provide stable endpoints for accessing applications. Services abstract the underlying pods and load balance traffic among them, allowing for seamless scaling and failover.

[3] https://kubernetes.io/docs/concepts/services-networking/

Kubernetes does not come with a built-in network implementation, instead, it relies on **Container Network Interface (CNI)**[4] plugins to provide networking capabilities. These plugins allow Kubernetes to make use of the networking model described above. Some popular CNI plugins include Calico, Flannel, and Cilium. The choice of CNI plugin depends on the specific requirements of the cluster, such as performance, security, and ease of use.

Aside from the networking model, Kubernetes networking uses some key components to manage communication within the cluster. These components include the CNI plugin, pods, services, endpoints, ingress, and network policies.

A **pod**[5] is the smallest deployable unit in Kubernetes. It encapsulates one or more containers. Each pod has its own IP address, which allows containers inside them to communicate directly with other pods in the cluster. Containers within the same pod share the pod's IP address and can communicate using `localhost`.

A **service**[6] in Kubernetes is an abstraction that defines a logical set of pods and a policy for accessing them. Services enable consistent and reliable communication by providing a stable IP address and DNS name, even when the set of pods behind the service changes (*e.g.*, in the case pods are scaled up, down or replaced). There are several types of services in Kubernetes:

- **ClusterIP**: The default service type, which provides an internal IP address for communication within the cluster;
- **NodePort**: Exposes the service on a specific port of each node, making it accessible externally;
- **LoadBalancer**: Integrates with cloud provider load balancers to expose the service to external traffic;
- **ExternalName**: Maps a service to an external DNS name, allowing pods to connect to services outside the cluster scope.

Endpoints track the IP addresses of the pods associated with a service. When a service is created, Kubernetes generates an endpoint object that maps to the pods selected by the service, allowing the service to direct traffic to the appropriate pods.

Kube-proxy is a network proxy that runs on each node in the cluster to implement services. It manages the network rules that enable communication between services and pods. Kube-proxy can operate in different modes, such as *iptables* or *IPVS*, to handle load balancing and routing efficiently. This component can be freely replaced by another implementation as desired.

[4] https://github.com/containernetworking/cni
[5] https://kubernetes.io/docs/concepts/workloads/pods/
[6] https://kubernetes.io/docs/concepts/services-networking/service/

The built-in **DNS service**[7] facilitates name resolution within the cluster. Each service in Kubernetes is assigned a DNS name, allowing pods to refer to services by their DNS names rather than IP addresses. The DNS service is typically provided by the cluster add-on **CoreDNS**. For example, a service named `my-service` in the `default` namespace can be accessed by `my-service.default.svc.cluster.local`.

An **Ingress**[8] is an API object that manages external access to services within a cluster. It provides HTTP and HTTPS routing, allowing the operator to define rules for routing traffic to different services based on hostnames or paths.

Network policies[9] define rules for controlling the communication between pods. They allow the operator to specify which pods are allowed to communicate with each other, providing a mechanism for securing the cluster's network traffic. For example, the operator can whitelist namespaces allowed to access an application.

The concepts of Kubernetes networking are important to manage containerized applications. It provides the mechanisms to enable communication between internal services and external APIs. The flexibility offered by the components of Kubernetes networking enables the platform to support a wide range of networking requirements in modern cloud-native environments.

2.3 Service Mesh

While Kubernetes networking ensures that pods and services can communicate, it does not natively provide all the needed features for large-scale, distributed applications. There is provided basic support for service discovery, load balancing, and networking features through its core components like Services, Endpoints, and Ingress. As the number of microservices increases, managing service communication becomes more challenging, such as[10]:

- **Fine-grained traffic control**: Routing traffic dynamically based on versioning, latency, or other factors;
- **Resilience mechanisms**: Implementing retries, timeouts, and circuit breakers to handle failures gracefully;
- **Security**: Ensuring encrypted communication and implementing authentication and authorization between services;
- **Observability**: Gaining visibility into service communication, including metrics, logs, and distributed tracing.

[7] https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/
[8] https://kubernetes.io/docs/concepts/services-networking/ingress/
[9] https://kubernetes.io/docs/concepts/services-networking/network-policies/
[10] https://glossary.cncf.io/service-mesh/

These requirements are difficult to achieve solely with raw Kubernetes networking features. In this scenario, service meshes extends Kubernetes capabilities to address these needs of managing microservices communication.

A service mesh is a dedicated infrastructure layer that manages communication between services in a microservices architecture, as exemplified in Figure 2.3 using the Online Boutique [17] application. It abstracts the network-related details from the application code and provides a consistent way to handle common networking tasks, such as load balancing, traffic routing, retries, circuit breaking, security (*e.g.*, mTLS), and observability (*e.g.*, metrics and tracing).

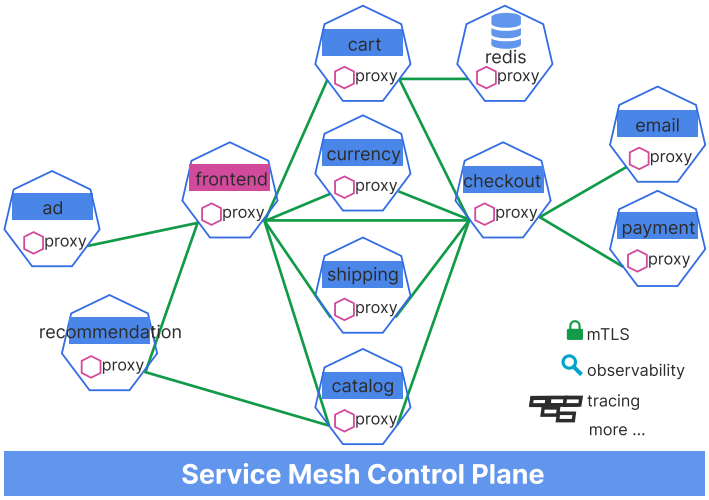


Figure 2.3: Online Boutique [17] application deployed in a service mesh environment.

Instead of embedding these tasks directly into the application, the service mesh shifts these responsibilities to the infrastructure layer. This separation of concerns allows developers to focus on writing application logic, while the service mesh handles the complexities of service-to-service communication. Service meshes offer a range of features that enhance the management of service-to-service communication:

- **Policy enforcement**: Defining and enforcing rules for network traffic, such as restricting which services can communicate with each other;
- **Traffic management**: Advanced routing capabilities, such as blue/green deployments, canary releases, and traffic splitting;
- **Resilience**: Automatic retries, timeouts, circuit breaking, and rate limiting to improve the reliability of services;
- **Security**: mTLS for encrypted communication, along with authentication and authorization mechanisms;
- **Observability**: Metrics, logs, and distributed tracing to provide deep insights into service performance and health.

At a high level, a service mesh works by intercepting the communication between services and applying policies and custom behaviors defined by operators. This interception is typically done using middleware proxies somewhere in between the networking path to handle the network traffic. The control plane of the service mesh manages these proxies, providing a central place to define and enforce networking policies and behaviors. In essence, a typical service mesh architecture consists of two main components:

- The **data plane** handles the actual traffic between services. The data plane consists of the middleware proxies;
- The **control plane** provides a centralized way to configure and manage the proxies in the data plane. It defines policies for traffic routing, security, and observability, and pushes these configurations to the data plane.

Service meshes can be categorized based on how they implement traffic interception and management. The two main types of service meshes are **sidecar-based** and **sidecar-less** architectures.

In a **sidecar-based service mesh**, each service instance has a dedicated proxy running as a sidecar container, as shown in Figure 2.4. This proxy intercepts and manages all incoming and outgoing network traffic for that service. The sidecar approach is popular because it is straightforward to implement in Kubernetes, where each pod can include the sidecar proxy as an additional container. Examples of sidecar-based service meshes include Istio, Linkerd, Consul.

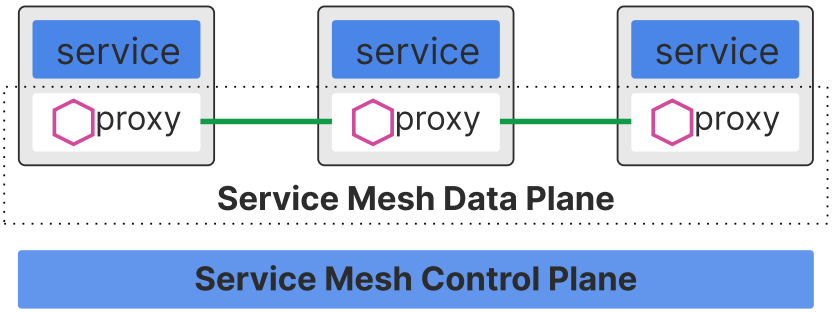


Figure 2.4: Sidecar-based service mesh architecture.

A **sidecar-less service mesh** takes a different approach by avoiding sidecar proxies. Instead, traffic interception and control are handled by agents running on the host node, exemplified in Figure 2.5, or by integrating directly with the kernel (*e.g.*, using eBPF). Examples of sidecar-less service meshes include Cilium, which uses eBPF to provide its features; and Istio's Ambient Mesh mode, which eliminates the need for sidecars by using a per-node shared proxy to intercept traffic, reducing resource consumption while maintaining service mesh capabilities.

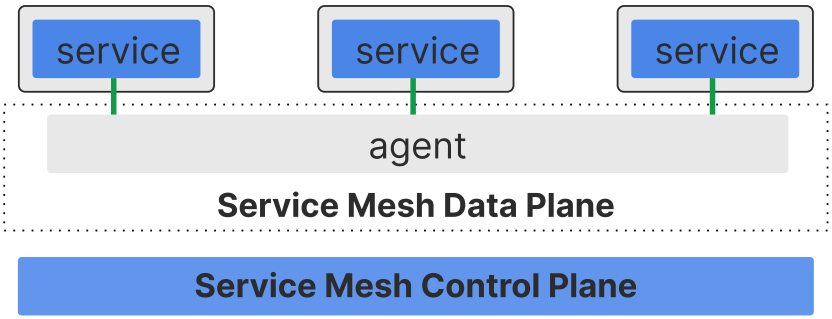


Figure 2.5: Per-node shared agent service mesh architecture.

The sidecar-less approach can simplify deployments and reduce resource overhead, since it eliminates the need to run a dedicated proxy for each service instance. Instead of multiple sidecar proxies consuming memory and CPU in each pod (1 pod $:$ 1 sidecar), a single agent or kernel-level integration can manage traffic for the entire node (1 node $:$ 1 sidecar $\therefore$ n pod $:$ 1 sidecar). This reduces the overall resource consumption and makes the deployment architecture cleaner and more efficient, especially in deployments with a high number of pods.

Service meshes extend Kubernetes networking by providing advanced capabilities for managing service-to-service communication in distributed systems. By abstracting the complexities of traffic management, security, and observability, service meshes enable developers and operators to build resilient, secure, and observable applications. Whether using a sidecar-based or sidecar-less architecture, service meshes improve the developer experience in the cloud-native ecosystem, providing the needed features of modern microservices deployments.

2.4 Performance Analysis

Performance is important in cloud-native environments, especially when dealing with container orchestration, Kubernetes networking, and service meshes. Each of these components introduces layers of abstraction that can impact the overall performance of applications. A thorough understanding of performance aspects is essential for advancing research in computer networks and cloud networking, enabling the development of more efficient, responsive, and scalable systems.

Kubernetes networking provides the basic mechanisms for communication between pods, services, and external endpoints. While Kubernetes offers flexibility and scalability, implementations of its networking model can introduce latency and resource overhead. The performance of Kubernetes networking is significantly impacted by the network plugin used.

The comprehensive benchmark study, Ducastel [18], evaluated several popular CNI plugins over a 40 Gbps network. The tested plugins included Antrea, Calico, Canal, Cilium, Flannel, Kube-OVN, and Kube-router, each with different configurations, such as default settings, eBPF integration, and encryption mechanisms like WireGuard and IPsec, totaling 21 different variations.

The benchmark utilized three Supermicro bare-metal servers connected via a 40 Gbps network switch (Figure 2.6), with jumbo frames enabled (MTU 9000). Kubernetes v1.26.12 was deployed using the RKE2[11] distribution on Ubuntu 22.04. The study aimed to reflect real-world conditions avoiding system tuning, aside from necessary adjustments like WireGuard installation and jumbo frame setup.

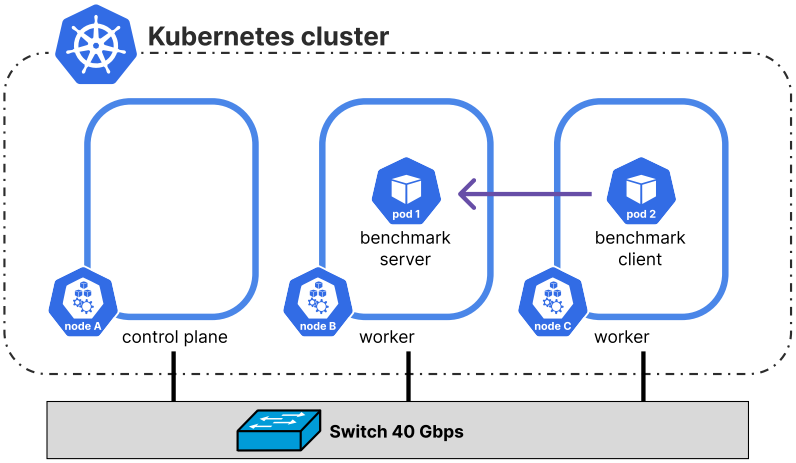


Figure 2.6: Benchmark architecture to evaluate Kubernetes network plugins performance [18].

The results revealed notable performance differences among the CNI plugins:

- **Antrea**: Performed well and is rapidly evolving, incorporating many appealing features. It is considered a strong alternative in the CNI landscape;
- **Calico**: Showed competitive performance, especially in its eBPF mode. However, certain advanced features were limited to its paid commercial product (Tigera), potentially affecting its suitability for some deployments;
- **Canal**: Offers a balance between Flannel and Calico. Provides network policies, but may not match the performance and feature set of more specialized CNIs;
- **Cilium**: Demonstrated strong performance across various configurations, particularly when utilizing eBPF for kube-proxy replacement. This plugin reduced latency and improved throughput, although increased memory usage due to aggressive eBPF maps pre-allocation;

[11] https://docs.rke2.io/

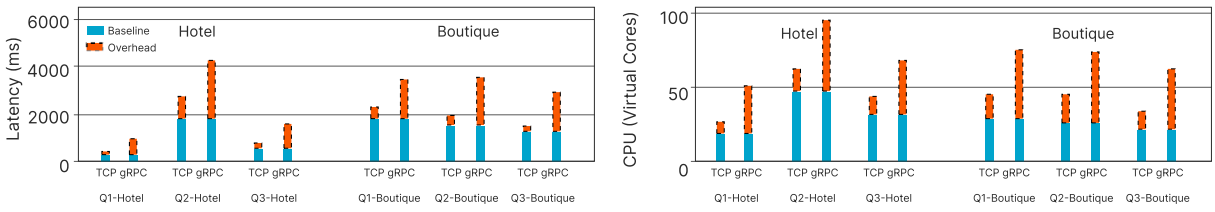
- **Flannel**: While emphasizing operational simplicity, Flannel's performance was generally lower compared to other CNIs and lacks some of the more advanced features like network policies;
- **Kube-OVN**: The plugin is memory and CPU-intensive. Offers advanced features thanks to Open vSwitch (OVS) but may require more resources compared to other CNIs;
- **Kube-router**: Exceptionally lightweight and efficient, performing well across all tested scenarios. Provides network policies out-of-the-box and supports a wide range of architectures, making it suitable for low-resource clusters like edge environments. The only concern is the currently small maintainers team.

In summary, the study concluded that for standard clusters, the recommended choices among the evaluated CNI plugins are Cilium, followed by Calico and Antrea. For low-resource environments, such as edge computing, the recommendation is primarily Kube-router, next to Flannel and Canal.

Besides of the network plugin, the **service mesh** is used to enhance Kubernetes networking by adding advanced features, however, it introduces performance penalties and increases resource consumption.

The study Zhu *et al.* [5] offers a comprehensive analysis of the performance costs associated with service mesh. The researchers developed a tool called *MeshInsight* to quantify these overheads in various deployment scenarios. The experiments were executed on Cloudlab machines and the software environment included Ubuntu 20.04 LTS (Linux kernel v5.4.0) with Kubernetes v1.12.5, Istio v1.13.0, and Envoy v1.21.0.

Their findings indicate that service meshes can substantially impact performance, with a 61% latency increase and a 92% CPU usage increase in benchmark applications operating in TCP mode, while using the Google Remote Procedure Calls (gRPC) protocol mode, the overhead is higher, with latency increases of up to 269% and CPU usage of up to 163% higher, as shown in Figure 2.7.



(a) Latency (b) CPU usage

Figure 2.7: Measurements of latency and CPU overheads resulting from the usage of service meshes in the Hotel Reservation [19] and Online Boutique [17] benchmark applications for three different queries, compared to a baseline scenario where applications run without a service mesh [5].

The higher gRPC overhead is primarily due to Envoy's packet manipulation processing that adds extra headers and increases the response packet size. Another contributing factor is protocol parsing, which is responsible for 63∼77% of the total overhead.

2.5 Summary

This chapter presented the core concepts related to container orchestration, Kubernetes networking, and service meshes in cloud-native environments. We explored the role of Kubernetes in managing containerized applications, its networking capabilities, and the limitations it faces when scaling microservices.

We also examined the importance of service meshes in addressing the advanced networking needs of microservices. Service meshes provide features like traffic management, security, and observability, but they come with performance penalties, such as increased latency and resource consumption. The addition of service mesh layers often leads to substantial overhead, particularly in sidecar-based architectures.

Performance analysis highlighted the impact of Kubernetes networking plugins and service meshes on Kubernetes environments. While some networking plugins show improved performance, such as Cilium, Calico, and Antrea, service meshes in general can lead to significant latency and CPU overhead. Despite these challenges, the service mesh approach remains an important part for complex microservices architectures.

In general, the chapter covered the necessary understanding of Kubernetes and the involved challenges in Kubernetes networking and service mesh implementations, providing a foundation for exploring optimization techniques, such as eBPF offloading, which can help mitigate these overheads.

Chapter 3

Offloading Technologies Review

In the context of network functions offloading, several technologies can be employed for optimization and acceleration. These technologies aim to improve network performance, reduce latency, and offload certain processing tasks to more efficient mechanisms within the kernel or user space. Some other techniques go further, offloading certain tasks from the host CPU to accelerators or specialized hardware. These technologies and their benefits are discussed below.

3.1 Frameworks and Programming Languages

The utilization of frameworks and specialized programming languages plays a relevant role in optimization and acceleration efforts. These software-based methods are focused on enhancing the resource usage of the underlying hardware, improving related metrics, and offloading specific processing tasks to more optimized routines.

3.1.1 eBPF and XDP

The extended Berkeley Packet Filter (eBPF) enables a safe and efficient execution of custom programs within the Linux kernel, extending its capabilities without requiring to recompile the kernel source code or load kernel modules [20]–[22].

eBPF programs are event-driven and can be attached to various pre-defined hooks within the kernel, including system calls, function entry or exit, kernel tracepoints, network events, and custom hook points. Figure 3.1 exemplifies some predefined hook points in the network path.

There are different eBPF program types available, each acting on different aspects of the Linux kernel. The eXpress Data Path (XDP) is the main type for networking-specific use cases. XDP leverages the eBPF framework to enable high-performance packet processing at the earliest possible stage of the Linux networking

stack. It allows direct packet processing at the network interface driver level before the kernel processes the packet. XDP programs can further improve network processing when hardware-offloaded to a networking device, such as a SmartNIC. By doing so, XDP can bypass the networking stack, achieving extremely low latency and high throughput [22]–[24].

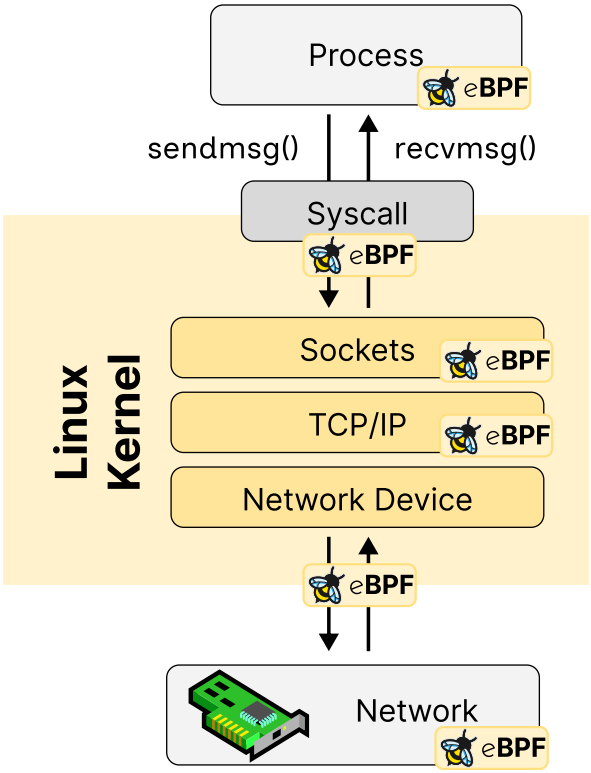


Figure 3.1: eBPF network hook points [21].

This flexibility enables diverse use cases, such as in networking. It can be used to perform packet filtering, monitoring, and manipulation of network traffic, allowing real-time, low-latency processing of network packets without having to pass them through user-space processes.

In a Kubernetes environment, eBPF can be used to implement advanced network security [25], load balancing [26], traffic shaping [27], and optimized packet transmission path [6], which can significantly improve network performance and reduce the waste of CPU resources to switch context between kernel and user space.

The study Yang *et al.* [6] proposes a novel approach using eBPF to address latency issues in service mesh architectures. The research intelligently reduces packet transmission times by avoiding the traditional kernel network stack. Their work shows that the non-intrusive solution can significantly reduce request latency by up to 21%, while also slightly reducing CPU and memory usage. This contribution not only adds to the ongoing discussion about network function optimization, but it also highlights the

versatility of eBPF in improving the efficiency of cloud-native systems. The findings from Yang *et al.* [6] shows eBPF's potential to change the way we offload network functions, enriching the spectrum of solutions aimed at overcoming the challenges posed by the increasing complexity of modern networking environments.

3.1.2 P4

With Programming Protocol-independent Packet Processors (P4), network engineers can customize the switch's packet processing logic to suit their requirements. This includes defining custom packet parsers, match-action tables, and actions, allowing the switch to efficiently process and forward network traffic according to the desired rules [28].

The flexibility of the language ensures portability across both hardware (*e.g.*, Tofino Switches, and Data Processing Units (DPUs) [29]) and software targets (*e.g.*, DPDK [30], [31]), releasing the potential of the infrastructure components.

The P4 programming language has gained attention as a tool for network packet processing. In the context of Kubernetes offloading, the work Jain *et al.* [32] discusses the utility of P4 in creating a Kubernetes load balancer and associated network functions tailored to meet the demands of scalability, security, and network performance. P4 demonstrates remarkable versatility across diverse deployment scenarios, from edge computing to data centers, including per-node load distributions.

3.1.3 DPDK

Developed by Intel, the Data Plane Development Kit (DPDK) is a set of libraries and drivers that allows applications to bypass the Linux kernel's networking stack and communicate directly with the underlying hardware. By doing so, DPDK significantly reduces the overhead added by the kernel, resulting in improved data plane performance and reduced latency [33].

In Kubernetes, DPDK can be used to accelerate networking functions and container networking, especially in scenarios where ultra-low latency and high throughput are required, such as in Network Function Virtualization (NFV) use cases [34].

In this sense, the field of programmable network devices is shifting towards customizable packet processing within the data plane. The work Alfredsson [35] investigates offloading strategies for multipath QUIC, evaluating frameworks, programming languages, and hardware devices for offloading cryptographic functions. It's a demonstration of the DPDK framework, in conjunction with P4, employed to design and implement packet processing offloading prototypes, showcasing their efficiency on a Nvidia BlueField-2.

3.1.4 IPDK

The Infrastructure Programmer Development Kit (IPDK) is an open-source and vendor-neutral framework of drivers and APIs designed for the purpose of infrastructure offload and management. At its core, IPDK provides a common platform for enhancing performance, resource optimization, and the security of the underlying infrastructure. IPDK uses a collection of well-established tools, namely Storage Performance Development Kit (SPDK), DPDK, and P4, that run on a range of hardware components, including CPUs, DPUs, and switches. These tools facilitate the realization of various functionalities such as network virtualization, storage virtualization, workload provisioning, root-of-trust establishment, and offload capabilities found in the given platform [36], [37].

A use case for IPDK in Kubernetes involves the transparent offloading of networking rules from the Calico CNI plugin to a P4-programmable target device [38].

While DPDK is a well-established framework for offloading, there remains a gap in exploring the capabilities and potential advantages of the IPDK framework, which offers infrastructure offload for a range of devices and services, including CPUs, DPUs, and switches. As of late 2024, it lacks community traction and published papers, warranting further investigation into its capabilities and potential contributions to the domain of network offloading [35].

3.2 Programmable Networking Hardware

Hardware-based methods for offloading involve the use of specialized hardware components to offload from the CPU certain networking tasks, such as encryption, decryption, compression, or packet processing. These hardware are designed to perform specific tasks faster and more efficiently than general-purpose CPUs.

3.2.1 Tofino Switch

Tofino is a family of programmable network switch Application-Specific Integrated Circuits (ASICs) developed by Barefoot Networks, acquired by Intel. These ASICs are switch protocol independent and allow network operators to implement custom forwarding behaviors using the P4 language, enabling highly efficient and flexible packet processing within the network switch hardware [39]–[41].

The usage of Tofino switches in Kubernetes clusters can lead to improved network performance [42], enhanced load balancing [43], advanced network management and observability [44], all of which contribute to a more efficient and optimized container orchestration environment.

The usage of P4-Programmable Tofino Switches is exemplified in Zha *et al.* [42], which presents a novel approach called EZPath, designed to enhance container networking traffic performance within data centers. This proposal leverages the programmable capabilities of Top-of-Rack Switches (ToR Switches), offering a seamless solution to expedite container traffic. By directly offloading traffic from containers to ToR Switches, it mitigates network bottlenecks, ensuring improved application performance without requiring changes to user applications, kernel modifications, or additional hardware support, while also demonstrating a substantial 35% throughput increase and a 42% reduction in tail latency, highlighting the effectiveness of using programmable switches for offloading in container environments.

In spite of these performance improvements, Intel has recently shifted its focus from the development of new Tofino Switch ASICs to concentrating on Infrastructure Processing Units (IPUs). Within the broader context of Tofino-based research, future usage of these programmable switches warrants careful consideration given the potential implications of this business decision change.

3.2.2 DPU or IPU

A Data Processing Unit (DPU) or Infrastructure Processing Unit (IPU)[1]—Intel's DPU—is a specialized hardware unit designed to accelerate infrastructure-related tasks in data centers. These tasks may include network functions, storage operations, security tasks, or other virtualized infrastructure workloads [45]–[48].

Incorporating DPUs in Kubernetes deployments can provide benefits in improved efficiency, enhanced security, faster network processing, and resource optimization [29].

Following the decision to stop the development of Tofino Switch ASICs, Intel is pushing the industry towards IPUs as a key component in network offloading strategies. A case in point is the collaborative effort between Intel and Google Cloud [49], [50]. This partnership has led to the introduction of the Intel IPU E2000, previously known as Mount Evans, which is a groundbreaking ASIC-based IPU designed to enhance high-performance computing and data-intensive workloads. As the industry seeks versatile and high-performance hardware solutions, IPUs such as the E2000 represent a significant development in the pursuit of efficient offloading capabilities for next-generation programmable infrastructures.

[1] Throughout this manuscript, IPU is used when referring specifically to Intel's DPU.

3.2.3 SmartNIC

A SmartNIC is a special Network Interface Card (NIC) capable of additional processing beyond traditional NICs. It is equipped with a programmable processor (*e.g.*, Field Programmable Gate Array (FPGA) or ASIC) and memory, allowing it to offload various network-related functions out of the host CPU. SmartNICs can handle tasks such as packet processing, security, and virtualization, reducing the load on the host CPU and improving network performance [51]–[54].

In Kubernetes, SmartNICs can improve container networking, security, and overall cluster performance by offloading tasks to specialized hardware [55].

In this realm, SmartNICs have also played an important role as valuable hardware components for offloading. The work Kato *et al.* [51] introduces a solution for optimizing Kubernetes microservice architecture. This approach leverages Nvidia BlueField SmartNICs to achieve full offloading of OVS, resulting in significantly reduced latency and alleviating the load on the host processor. The full offload, integrated into the Antrea OVS-based CNI plug-in, yields latency reductions of up to 55% for intra-node communications and up to 57% for inter-node communications. SmartNICs, with their offloading capabilities, demonstrate their potential for addressing latency and communication challenges within containerized environments.

Recent publications demonstrate some new approaches in the offloading landscape. The paper Brunella *et al.* [24] proposes the solution hXDP, which runs eBPF-based packet processing tasks on NetFPGA-SUME SmartNICs [56], specifically targeting Linux XDP. Remarkably, hXDP optimizes FPGA resources and matches the performance of high-end CPUs while significantly reducing packet forwarding latency. In this sense, Pacífico *et al.* [27] proposes eBPFlow, an optimized implementation of eBPF in FPGA, offering a throughput 2.59 Gbps higher. These researches underscores the potential of a hybrid approach, harnessing the strengths of FPGAs, eBPF, and XDP, to tailor offloading strategies to the unique requirements of each network processing scenario, thereby optimizing both performance and resource utilization.

3.3 Related Work

The landscape of network offloading is rapidly evolving, driven by the need to enhance the performance, security, and scalability of Kubernetes and other cloud environments. Within this context, various innovative approaches have been explored to optimize network packet processing. Related to our work, we found X-IO [57], SPRIGHT [58], Cilium [59], and Yang *et al.* [6].

X-IO [57], a high-performance I/O interface, aims to eliminate kernel networking overheads and contention of microservices using shared memory processing. Although it offers a 2.8∼4.1× latency improvement, it requires changes to the application code and is unable to run alongside a service mesh, lacking many benefits provided by it.

Another proposal, SPRIGHT [58], a serverless framework, makes use of shared memory and eBPF to improve the scalability of the data plane. It exhibits competitive performance results of 5× throughput improvement, 53× latency reduction and 27× CPU usage savings compared to Knative. Despite that, it also suffers from the lack of service mesh features.

As an alternative, Cilium [59], a sidecar-less service mesh, heavily uses eBPF to implement its features. The downside is that it requires the usage of its own CNI. In this sense, clusters deployed with other CNIs cannot use Cilium.

Finally, Yang *et al.* [6] presents a network optimization based on eBPF to bypass the kernel network processing. The work improves request latency by up to 21% for 90% of requests. Still, it only works for Istio in sidecar mode.

Compared to related work, eZtunnel, our proposal, offers a better approach to optimize service mesh data plane networking. It works with the newest service mesh proposals, like ambient mesh, offloads the network transparently, and does not require redeployment of the cluster. Table 3.1 summarizes the findings of related work.

Work	**Technique**	**Advantages**	**Limitations**
X-IO [57]	Shared memory I/O	Eliminates kernel overheads; achieves 2.8∼4.1× latency improvement	Requires application code modifications; cannot coexist with service meshes
SPRIGHT [58]	Serverless framework	Uses shared memory and eBPF; achieves 5× throughput, 53× latency reduction, and 27× CPU savings	Lacks features provided by service meshes
Cilium [59]	eBPF sidecar-less mesh	Implements features without sidecar proxies	Requires use of its own CNI, incompatible with other CNIs; requires eBPF support
Yang *et al.* [6]	eBPF-based optimization	Reduces request latency by up to 21%; reduces CPU/memory usage	Limited to Istio in sidecar mode; IPv4-only
eZtunnel (Our Work)	eBPF-based offloading	Transparent offloading; increases intra-node throughput by 68.5%, reduces latency by 42.0%, and FCT by 41.2%; integrates with ambient mesh; no cluster redeployment required; supports both IPv4 and IPv6	Requires eBPF support; not fully tested with other service meshes; memory usage increased by at most 60.5 MB; no support for Docker-based deployments (*e.g.*, kind, minikube)

Table 3.1: Comparison of the most important related works.

3.4 Summary

This chapter explored the diverse landscape of offloading techniques that aim to address the challenges of network optimization, including in modern cloud-native environments. These techniques span software-based frameworks, programmable hardware, and hybrid approaches, each with its advantages and trade-offs.

Software frameworks like eBPF and XDP showcase the potential for high-performance packet processing within the kernel, bypassing traditional networking stacks to deliver low latency and high throughput. Similarly, programming tools like P4 enable precise control over network behavior, particularly when combined with advanced hardware solutions like Tofino switches or DPUs. Additionally, frameworks such as DPDK and IPDK highlight the flexibility and power of user-space packet processing and infrastructure management for Kubernetes clusters.

Hardware-based approaches, including SmartNICs, programmable switches, and DPUs, offer unparalleled efficiency by offloading resource-intensive tasks directly to specialized components. These techniques significantly reduce latency and CPU usage while enhancing the scalability and security of network infrastructures.

The related work reviewed in this chapter illustrates the breadth of solutions available, from the latency reductions achieved by sidecar-free service meshes to the throughput gains enabled by hardware-based offloading. Each approach provides valuable insights into overcoming the complexities of Kubernetes networking and service mesh architectures, yet limitations persist in terms of compatibility, deployment flexibility, and hardware requirements.

In this context, our proposed solution, eZtunnel, emerges as a promising approach to address these challenges. By leveraging eBPF for transparent offloading in service mesh environments, eZtunnel reduces intra-node latency, minimizes jitter, and integrates seamlessly with modern service mesh designs like ambient mesh. Unlike other techniques, it requires no cluster redeployment, ensuring a smooth and efficient optimization process without significant overhead.

Chapter 4

eZtunnel: Design and Implementation

Current cloud-native environments face challenges with networking overhead due to container orchestration platforms like Kubernetes and networking enhancement layers such as service meshes like Istio. Both sidecar-based and sidecar-less service mesh deployment modes introduce a longer communication path and increased performance overhead. The performance of applications as a whole can be affected by the costly networking stack traversals required to handle microservices communication.

While solutions addressing the inefficiencies of sidecar deployment mode have been proposed, the performance drawbacks associated with the sidecar-less mode remain unresolved. Our proposal seeks to avoid long networking stack traversals by employing eBPF as a software optimization mechanism to bypass the Linux kernel network processing.

This chapter explores and describes eZtunnel, the proposed eBPF-based transparent traffic acceleration mechanism to reduce the associated overhead of intra-node networking in cloud-native infrastructures.

4.1 Guidelines

To develop an effective offloading strategy for next-generation programmable infrastructures, we aimed to come up with a solution that **ensures compatibility with Kubernetes and service mesh deployments**, allowing seamless integration without requiring environment modifications. It needed to **support both deployment modes**, sidecar-based and sidecar-less service mesh configurations, and be **transparent** to avoid requiring modifications to applications or infrastructure components.

Additionally, the solution had to address key performance aspects, such as **reducing intra-node networking overhead**, optimizing the communication path to alleviate costly traversals through the Linux network stack, and **reducing latency** by bypassing unnecessary kernel processing.

Moreover, the solution needed to be **resource-efficient**, minimizing computational and memory usage to support scalability while maintaining performance. It also aimed to be **cost-effective**, avoiding reliance on specialized hardware and ensuring feasibility within commodity cloud environments. By **using mature and evolving technologies and toolsets**, the strategy can provide robust functionality while benefiting from strong community support, leading to better maintainability.

4.2 Technology Selection for Offloading

Taking the discussed requirements and guidelines into consideration, we analyzed the offloading technologies explored in the literature review (§3) to identify a suitable key technology for building the project, ensuring alignment with outlined ideas.

P4 offers highly efficient programmable packet processing, but relies on specialized hardware, which limits its applicability in commodity cloud environments.

Although promising, IPDK's ecosystem and toolset remain immature, making it less suitable for immediate deployment. In contrast, although Tofino Switch ASICs are mature, relying on them poses risks due to Intel's decision to discontinue their development.

DPDK, despite its high performance, operates in user space and requires application modifications, which reduces transparency and increases intrusiveness. DPDK is primarily effective for optimizing inter-node traffic, such as DPUs and SmartNICs, not offering benefits for intra-node communication.

As opposed to the aforementioned technologies, eBPF stands out as a highly flexible and efficient solution that addresses many of these limitations. Unlike P4, eBPF does not require specialized hardware, making it suitable for deployment in commodity cloud environments where cost and accessibility are taken into consideration. Its integration with the Linux kernel ensures compatibility with existing infrastructures, eliminating the need for specialized or proprietary equipment.

While IPDK and Tofino Switch ASICs face challenges related to maturity and future development risks, eBPF benefits from an active and rapidly evolving ecosystem. This ensures continuous improvements, widespread community support, and a reduced risk of obsolescence. Additionally, eBPF's maturity as a technology enables it to be readily deployed without requiring infrastructure upgrade, as almost every modern Linux system since version 3.15 supports it [60].

Compared to DPDK, eBPF operates directly within the kernel, bypassing the overhead of context switching between user-space and kernel-space. This in-kernel operation not only reduces latency but also provides a more transparent and less intrusive mechanism for optimizing network functions.

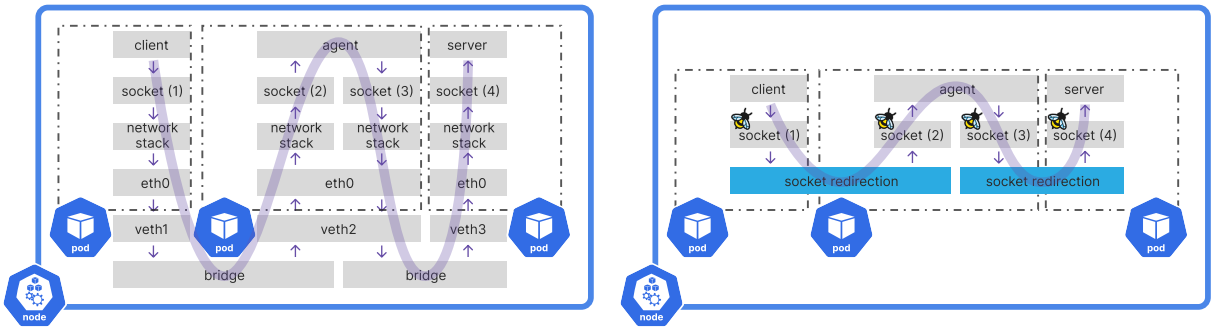
Due to these features, eBPF is the most reasonable choice for an offloading strategy in this scenario. The ability to operate with minimal performance overhead while providing broad observability and control over system behavior makes eBPF a standout choice for next-generation programmable infrastructures.

4.3 General Overview of the Solution

In the context of service meshes, intra-node networking can become a burden due to the high volume of traffic between containers. Efficient management of this traffic is critical to optimizing application performance and resource utilization at node-level.

Figure 4.1a depicts the packet's path from a client to a server process in a sidecar-less service mesh environment. Initially, a message is written to socket (1). It traverses the network stack down to the network interface, where a virtual bridge forwards the packet to the respective interface of the agent's pod. It then traverses again the network stack, and socket (2) delivers the message to the process. The same process is repeated from socket (3) to socket (4) to forward the message to the target server, and also to send the response back from the server to the client.

Our design to shorten the packet's path involves a socket redirection mechanism based on eBPF, as illustrated in Figure 4.1b. Instead of traversing the network stack, network interfaces, and virtual bridges, as the packet is written to the socket, it is directly redirected to the other socket's end and delivered to the process, entirely bypassing the intermediary kernel network processing. This mechanism works regardless of the service mesh deployment mode (*e.g.*, sidecar-based or sidecar-less).



(a) Default packet path

(b) Packet path of our proposed solution

Figure 4.1: Packet path of a sidecar-less service mesh with and without our proposed solution.

Considering that in a service mesh architecture packets always need to travel through the middleware proxy first, this improvement is useful even in cases where the packet destination is outside the node.

4.4 Design

To route messages between sockets and skip the Linux network stack, sockets must first be captured, stored, and monitored for messages.

A *SockOps* program is attached to a *control group (cgroup)*, which responds to events in sockets (*e.g.*, socket established) of processes in the attached *cgroup* hierarchy. It allows us to change socket parameters and opportunistically store them in eBPF Maps [61]. Linux provides a variety of eBPF Maps, including *SockHash*, which is used to store sockets in a hash table with a user-defined key, and *HashMap*, which both key and value can be used-defined.

Once sockets are captured and stored in *SockHash* and *HashMap* via the *SockOps* program, the next phase involves defining how messages traverse these sockets without invoking the kernel's default packet path.

A *SkMsg* program is attached to *SockHash* to handle messages sent through the stored sockets, *i.e.* when '`sendmsg`' and '`sendfile`' syscalls are executed on sockets that are part of the Map the *SkMsg* program is attached to.

As arriving messages are detected, the redirection mechanism involves the query of *HashMap* to obtain the respective destination of the message. Then, messages are redirected to the other end of the socket using the helper function `bpf_msg_redirect_hash`.

Figure 4.2 details the socket redirection workflow. (1) When the server socket is created, it is captured by *SockOps* program. The captured socket is (3) stored in *SockHash* and (4) the reverse mapping is calculated and stored in *HashMap*. (2) The client socket is also captured and (3) stored in the *SockHash* and (4) *HashMap*. (5) When a message is written to the socket, the *SkMsg* program is triggered, (6) *HashMap* is queried to obtain the mapping of the other side of the socket and (7) (8) redirects it to the obtained socket, (9) delivering it to the server process.

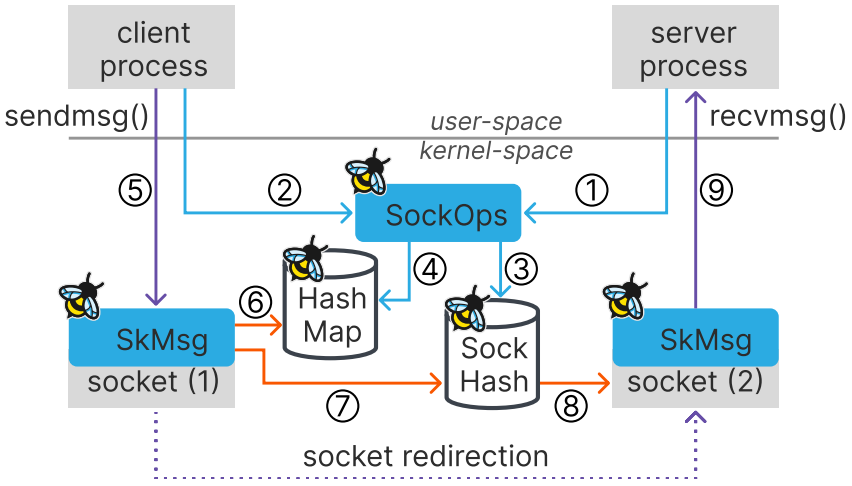


Figure 4.2: Socket redirection workflow using eBPF.

4.5 Implementation

The implementation is composed by the user-space code and eBPF programs, where the user-space code handles loading the eBPF programs into the kernel.

The implementation relies on a modern toolchain that simplifies the development, deployment, and management of eBPF programs. Central to this toolchain is the Aya library [62], a Rust library designed for eBPF programming. Aya provides a robust framework for writing, compiling, and loading eBPF programs while maintaining portability and minimizing dependencies on external tools.

This library was chosen for its features as an eBPF library fully implemented in Rust with bindings to eBPF helpers in C. Unlike other eBPF libraries that often depend on C-based frameworks or C code, Aya offers a Rust-native solution. This brings several advantages of the Rust language, including improved memory safety without garbage collection, type safety, better integration with Rust-based projects, and a simplified development workflow. Aya supports a "compile-once, run-everywhere" approach, which ensures that the eBPF program can operate across various Linux distributions and kernel versions without requiring per-environment recompilation.

The eBPF programs are written in Rust and compiled into eBPF bytecode using the Rust toolchain. The compilation process relies on `cargo`, Rust's package manager and build system, along with two crates[1]:

- `bpf-linker`: A crate that facilitates the process of statically link eBPF objects, ensuring compatibility with kernel requirements and generating optimized bytecode;
- `bindgen-cli`: This tool generates Rust bindings to C headers, allowing the eBPF programs to interface seamlessly with kernel structures and APIs.

Additionally, the Aya library provides utilities for managing the lifecycle of eBPF programs from user-space. When the user-space code is executed, it loads the precompiled eBPF bytecode into the kernel. This step is handled programmatically, without the need for external tools like `bpftool` or custom loaders.

4.5.1 User-space program

The eBPF programs are precompiled into bytecode and included in the compiled binary. The macro `include_bytes_aligned` ensures that the bytecode is included as a compile-time constant, avoiding runtime dependency on external files, and `Ebpf::load()` handles parsing and loading the eBPF program into memory. This procedure is shown in Listing 4.1.

[1] In Rust, a 'crate' is similar to 'library' or 'package' in other languages.

```
let mut bpf = Ebpf::load(include_bytes_aligned!(
    "path/to/compiled/ebpf/program"
))?;
```

Listing 4.1: Loading the eBPF program bytecode into memory.

The *SockOps* program monitors socket lifecycle events and must be attached to a cgroup. The directory of the cgroup can be provided via a command-line argument or a default value is used, as defined in Listing 4.2. Then, in Listing 4.3, the corresponding file descriptor of the given cgroup is obtained.

```
#[derive(Debug, Parser)]
struct Opt {
    #[clap(short, long, default_value = "/sys/fs/cgroup/")]
    cgroup: String,
}
```

Listing 4.2: Declaration of the userspace program options.

```
let opt = Opt::parse();
let cgroup_file = File::open(opt.cgroup)?;
let cgroup_fd = cgroup_file.as_fd();
```

Listing 4.3: Obtaining the file descriptor of the provided *cgroup* directory.

Once the file descriptor of the cgroup is available, the *SockOps* program can be attached to it. In Listing 4.4, the `intercept_active_sockets` program is retrieved (`.program_mut()`) from the loaded eBPF object, loaded into the kernel (`.load()`) and attached to the specified cgroup (`.attach()`).

```
let program_name = "intercept_active_sockets";
let intercept_active_sockets: &mut SockOps =
    bpf.program_mut(program_name).unwrap().try_into()?;
intercept_active_sockets.load()?;
intercept_active_sockets.attach(cgroup_fd, CgroupAttachMode::default())
    .context(format!("failed to attach SockOps program `{program_name}`"))?;
```

Listing 4.4: Attaching the *SockOps* program to the *cgroup* file descriptor.

When the *SockOps* program detects new sockets, they are captured and stored in a specialized eBPF Map called *SockHash*. This Map allows the *SockOps* program to store references to active sockets. In Listing 4.5, the Map is configured and the respective file descriptor is obtained.

```
let sockets: SockHash<_, [u8; mem::size_of::<SockId>()]> =
    bpf.map("SOCKETS").unwrap().try_into()?;
let sockets_fd = sockets.fd().try_clone()?;
```

Listing 4.5: Setting up the *SockHash* Map and obtain its file descriptor.

The *SkMsg* program is responsible for redirecting messages between sockets. It operates on sockets stored in the *SockHash* Map. Similarly to the previous eBPF program, in Listing 4.6, the `redirect_between_sockets` program is retrieved (`.program_mut()`) and loaded (`.load()`), then it is attached to the *SockHash* Map referenced by its file descriptor (`.attach()`). This ensures that the *SkMsg* program is triggered when data is sent through any socket stored in the Map.

```
1 let program_name = "redirect_between_sockets";
2 let redirect_between_sockets: &mut SkMsg =
3     bpf.program_mut(program_name).unwrap().try_into()?;
4 redirect_between_sockets.load()?;
5 redirect_between_sockets.attach(&sockets_fd)
6     .context(format!("failed to attach SkMsg program `{program_name}`"))?;
```

Listing 4.6: Attaching the *SkMsg* program to the *SockHash* Map file descriptor.

4.5.2 Kernel-space program

The user-space code does not directly interfere with the socket handling mechanism, on the other hand, the eBPF programs perform this role entirely. The first step is to obtain the sockets and manage Maps using the *SockOps* program. The `#[sock_ops]` attribute macro defines the program type of the function, which gets a parameter of type `SockOpsContext`, as seen in the function signature in Listing 4.7.

```
1 #[sock_ops]
2 fn intercept_active_sockets(ctx: SockOpsContext) -> u32 { /* ... */ }
```

Listing 4.7: Function signature of the `intercept_active_sockets` eBPF program.

The internals of this function follow the logic depicted in Figure 4.3. When a socket event is triggered, first the IP address is normalized, *i.e.* IPv4 addresses are mapped to IPv6. Then, it checks if the socket operation is active (*i.e.* the client socket initiating the connection) or passive (*i.e.* server receiving the connection). Finally, the socket is stored in the `SOCKETS<key: SockId, value: socket>` eBPF Map using a special hash key, and the reverse side of the socket is calculated and stored in `SOCKETS_REVERSED<key: SockPairTuple, value: SockId>` eBPF Map.

IPv4 addresses are mapped to the IPv6 address space using the prefix `::FFFF:`$x.y.z.w$, where $x.y.z.w$ represents the original IPv4 address, as specified by RFC 4291[2]. Address collision is mitigated as the RFC reserves this IPv6 prefix exclusively for IPv4-mapped IPv6 addresses. In this way, the program is able to handle both `AF_INET` and `AF_INET6` sockets. This process is executed in the same way to normalize the value of both `local_ip` and `remote_ip`, as exemplified in Listing 4.8.

[2] https://datatracker.ietf.org/doc/html/rfc4291#section-2.5.5.2

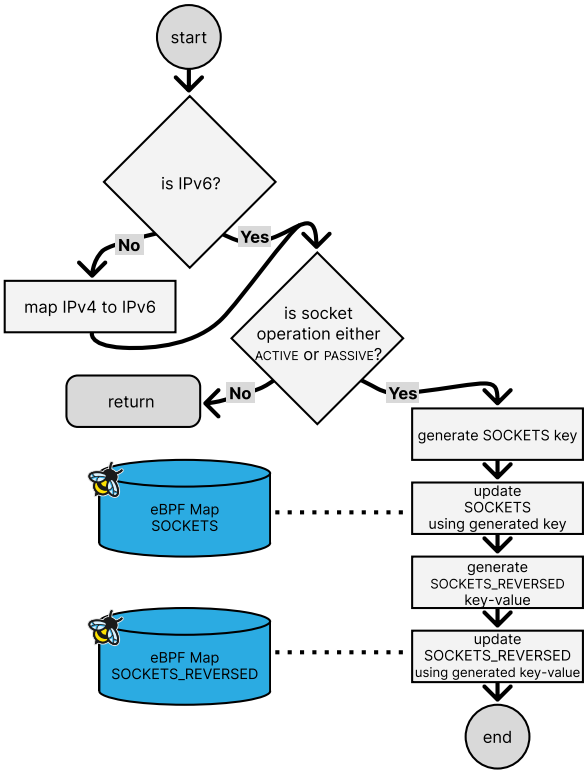


Figure 4.3: Flowchart of the `intercept_active_sockets` eBPF program.

```
let local_ip = if family == IpAddrKind::V4 as u8 {
    [0, 0, 0xffff, ops.local_ip4.swap_bytes()]
} else {
    ops.local_ip6
};
```

Listing 4.8: IPv4 to IPv6 mapping, according to RFC 4291.

The `SOCKETS<key: SockId, value: socket>` Map is indexed by `SockId` (Listing 4.9), derived from the socket's 4-tuple `{source IP-port, destination IP-port}`. If the socket operation is active (*i.e.* client-side), the local IP and port are used to identify the connection because the server IP and port are common to all connections. Else, the socket operation is passive (*i.e.* server-side), then the remote IP and port are used, which represent the client. This procedure is in Listing 4.10.

```
#[repr(C, packed)]
#[derive(Copy, Clone)]
pub struct SockId {
    pub side: SockSide,
    pub ip: [u32; 4usize],
    pub port: u16,
}
```

Listing 4.9: `SockId` struct definition.

```
let mut sock_id = if ops.op == BPF_SOCK_OPS_ACTIVE_ESTABLISHED_CB {
    SockId {
        side: SockSide::Client,
        ip: local_ip,
        port: local_port,
    }
} else {
    SockId {
        side: SockSide::Server,
        ip: remote_ip,
        port: remote_port,
    }
};
```

Listing 4.10: `SOCKETS` Map index key, defined by `sock_id`.

To calculate the reversed side of the socket identified by `SockPairTuple` (the Map key, Listing 4.11), `sock_id` is reused and the side is flipped (the respective Map value). The key-value is stored in `SOCKETS_REVERSED<key: SockPairTuple, value: SockId>` eBPF Map. This is shown in Listing 4.12.

```
#[repr(C, packed)]
#[derive(Copy, Clone)]
pub struct SockPairTuple {
    pub local_ip: [u32; 4usize],
    pub local_port: u16,

    pub remote_ip: [u32; 4usize],
    pub remote_port: u16,
}
```

Listing 4.11: `SockPairTuple` struct definition.

```
let sock_pair_tuple = SockPairTuple {
    local_ip,
    local_port,
    remote_ip,
    remote_port,
};

sock_id.side = if sock_id.side == SockSide::Client {
    SockSide::Server
} else {
    SockSide::Client
};
```

Listing 4.12: Calculation of the `SOCKETS_REVERSED` value.

Following the *SockOps* program, the second step is the *SkMsg* program to perform the actual redirection of messages between sockets. The `#[sk_msg]` attribute macro is used to define eBPF program type of the function, which gets a parameter of the type `SkMsgContext`, as defined in the function signature in Listing 4.13.

```
1 #[sk_msg]
2 fn redirect_between_sockets(ctx: SkMsgContext) -> u32 { /* ... */ }
```

Listing 4.13: Function signature of the `redirect_between_sockets` eBPF program.

The functioning of `redirect_between_sockets` is shown in Figure 4.4. Using the populated Maps by `intercept_active_sockets`, the remaining part is the redirection mechanism. When a stored socket in the `SOCKETS` Map triggers a message event, IPv4 addresses are once again time mapped to IPv6. Then, the key to obtain the reversed socket stored in `SOCKETS_REVERSED` is constructed, in the same way `sock_pair_tuple` is. The returned value from the Map is used as the key to get the corresponding socket, to which the message is redirected to.

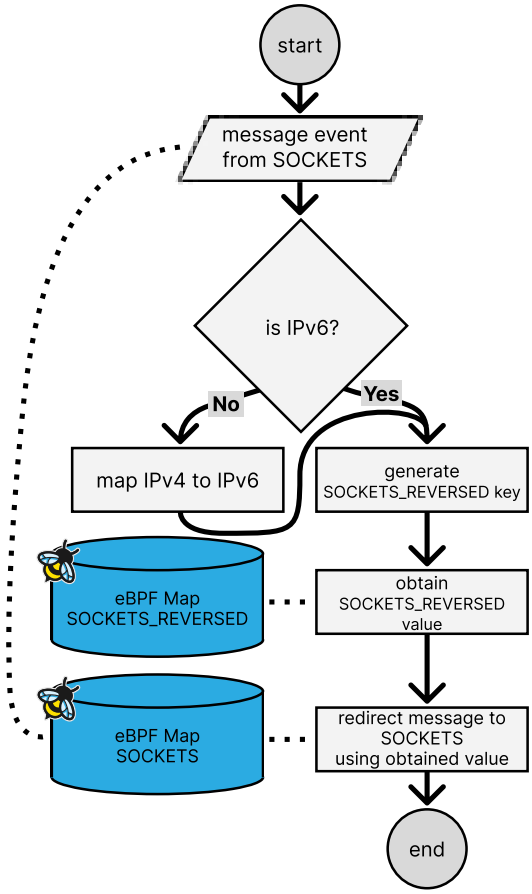


Figure 4.4: Flowchart of the `redirect_between_sockets` eBPF program.

4.6 Deployment Considerations

The eBPF programs and their support user-space component can be deployed in different ways. The simplest deployment method is to manually download the binary from the GitHub repository to the Kubernetes node (*e.g.*, using curl or wget) then execute it with root privileges. This is useful for a quick and easy test of the solution.

Another alternative method is to package it into a container image and deploy as a *DaemonSet*. Each node's *DaemonSet* should have access to the node filesystem then attach the *SockOps* and *SkMsg* programs to the top level *cgroup* of the node and initialize the necessary eBPF Maps. Using Kubernetes primitives for deployment guarantees that the prototype is active on all nodes within the cluster and ensures a more robust deployment.

To facilitate testing and debugging, it is always recommended to deploy the prototype in a spare Kubernetes cluster, not used for production workloads. After the initial testing, it can be gradually deployed to a production cluster, selectively picking nodes using Kubernetes' node selector. This allows the implementation to be evaluated in a controlled environment before being rolled out cluster-wide.

4.7 Summary

This chapter presented an eBPF-based offloading strategy to optimize intra-node networking in Kubernetes environments with service meshes. Given the identified performance overhead associated with traditional networking stacks, we adhered to a set of guidelines for the project and proposed an offloading solution based on eBPF to bypass kernel processing, aiming to improve network metrics and resource utilization. The solution was tested in various scenarios and the results are presented in the following chapter.

By using the eBPF programs *SockOps* and *SkMsg*, our approach captures socket events, stores socket references in *SockHash* and *HashMap* eBPF Maps, and redirects messages directly between sockets, avoiding unnecessary traversals of the Linux network stack. This solution is intended to work transparently in both sidecar-based and sidecar-less service mesh configurations. The implementation was built using Aya, a Rust library, which enabled efficient development and deployment of eBPF programs.

In conclusion, the proposed eBPF offloading strategy provides a flexible, cost-effective, and high-performance solution for reducing the overhead of intra-node networking in Kubernetes environments, making it suitable for modern cloud-native architectures.

Chapter 5

Experimental Evaluation

This chapter evaluates the proposed offloading strategy by analyzing its performance in practical scenarios. The focus is on measuring how the eBPF-based optimization enhances networking and system performance across selected workloads. Key network and system metrics such as latency, throughput, and resource utilization are assessed using benchmarks designed to represent both real-world use cases and commonly used scenarios. The chapter outlines the workloads, metrics, testbed setup, benchmark results, and the discussion of the outcomes.

5.1 Guidelines

The selection and evaluation of workloads in this experiment follow some guidelines to ensure that the results are reproducible, meaningful, and representative of real-world use cases in cloud-native environments. The guidelines outline the key principles for the design and evaluation of workloads.

Reproducibility. Workloads should be designed to be reproducible, with consistent configurations and clear metrics. This allows for reliable comparisons across different experimental setups and ensures that performance improvements can be attributed to the offloading strategy.

Diversity of Scenarios. A range of network scenarios should be covered, from low-latency, high-frequency interactions to high-concurrency, data-intensive operations. This ensures that the evaluation addresses various service mesh performance bottlenecks across different application requirements.

Representativeness. The chosen workloads should reflect the common network patterns found in cloud-native applications. These patterns include basic request-

response communication as well as more complex and data-intensive interactions typical of microservice-based architectures.

Following these guidelines, the workloads chosen for this evaluation provide an overview of the service mesh performance optimization assessment.

5.2 Workloads

The workloads selected for this evaluation aim to assess the impact of the offloading strategy on network performance and system usage, focusing on common networking tasks in cloud-native environments. These workloads are designed to test the effect of the optimization across different communication patterns and system loads.

file-transfer. It consists of a server transferring a large synthetic file (1,000 MiB) to a client over TCP using *ncat* utility[1]. The workload simulates a high-throughput scenario commonly encountered in data-intensive applications, such as file sharing, media streaming, or backup operations. It evaluates the system's ability to efficiently handle bulk data transfers under controlled conditions.

ping-echo. The *ping-echo* workload consists of a ping client and an echo server, both implemented in Python. The client sends a small packet (24 bytes) over TCP containing a timestamp to the server, which responds with the same packet. This request-response cycle pattern is typical of many microservices interactions; also, it was helpful to understand the interaction between eZtunnel and service mesh. This workload evaluates low-latency, high-frequency communication, which is common in microservices environments where services frequently exchange small amounts of data.

redis. This workload uses a real Redis[2] instance with the Memtier Benchmark[3] tool to generate high-concurrency, data-intensive operations. It offers a broad range of configuration parameters, such as total runs, amount of threads, clients per thread, requests per client, and others. The workload is designed to simulate the type of load encountered in caching or session management services.

[1] https://nmap.org/ncat/
[2] https://redis.io/docs/latest/get-started/
[3] https://github.com/RedisLabs/memtier_benchmark

5.3 Metrics

The following metrics were used to evaluate the performance of the eBPF-based offloading strategy in the selected workloads. These metrics provide insight into how the system's networking and resource usage are impacted by the offloading mechanism. Network-related metrics are collected by the client process itself, while resource usage-related metrics are monitored by *statexec*[4].

Latency. Latency, also called 'Round Trip Time latency' (RTT latency), refers to the time it takes for a packet to travel from the client to the server and back. This metric is capable of evaluating the responsiveness of microservices, particularly in real-time applications. Lower latency translates to faster communication between microservices.

Jitter. Jitter measures the variation in packet delay over time, indicating the consistency of latency. It is critical for applications requiring stable and predictable communication, such as real-time systems and streaming services. Reducing jitter ensures smoother interactions and a better end-user experience. This metric is derived from the network latency. It is defined by the average deviation from the network mean latency, as described in the formula:

$$\sigma = \sqrt{\frac{1}{N}\sum(x_i - \overline{x})^2} \tag{5.1}$$

Where:

- σ: population standard deviation
- x_i: each value from the population
- $\overline{x}$: population mean
- N: size of the population

Flow Completion Time (FCT). Unlike single-packet latency, FCT accounts for the complete transfer of multiple packets that constitute a flow, making it particularly relevant for applications involving bulk data transfers or file exchanges. Lower FCT indicates more efficient handling of end-to-end communication. FCT is essential for evaluating the performance of services that depend on large data transmissions, such as database replication, video streaming, and file transfer operations.

[4] https://github.com/blackswifthosting/statexec

Throughput. Throughput is the measurement of the rate at which data can be transmitted between services. It is used to understand how much data the system can handle under different loads. Throughput is typically measured in terms of requests (Requests per Second (RPS)), operations (Operations per Second (OPS)) or amount of data (*e.g.*, megabytes per second (MBps)). It can be derived from latency as following:

$$RPS = OPS = \frac{1}{l} \qquad bps = \frac{s}{l} \tag{5.2}$$

Where:

- l: latency (in seconds)
- s: size of each request

CPU Usage. This metric measures the percentage of CPU resources consumed by the whole system. High CPU usage can indicate inefficient processing, especially in network-related tasks. A reduction in CPU usage would demonstrate the effectiveness of the offloading strategy in optimizing resource consumption.

Memory Usage. It refers to the amount of system memory used during the execution of workloads. High memory usage may lead to system instability, including crashes or the invocation of out-of-memory (OOM) killers, disrupting service availability.

5.4 Testbed Setup

The testbed for evaluation was designed to ensure reproducibility and consistency, minimizing variability caused by external factors while still providing a realistic environment for testing the proposed optimization.

Server Specifications. A single physical machine was used for the experiments, featuring the following configuration:

- **CPU**: Intel Xeon E5-2630 v4, base clock 2.2 GHz, boost clock up to 3.1 GHz, 10 cores, 20 threads;
- **RAM**: 2x 16 GB, DDR4, 3200 MHz;
- **Swap Memory**: 8 GB;
- **Storage**: SSD NVMe 1 TB;
- **Operational System**: Fedora Server 40;
- **Kernel**: Linux v6.8.5, cgroup2 enabled;
- **Network Interface**: Integrated Gigabit Ethernet.

Virtualization. To emulate a Kubernetes environment, LXD[5] was used to create lightweight Virtual Machines (VMs). LXD offers near-native performance and resource isolation, making it suitable for testing networking and workload scenarios. The setup of the instance is carried out by *cloud-init*[6] to allow consistent and reproducible installations. Each instance was configured as type '*vm*' with dedicated resources:

- **Number of VMs**: 1;
- **Allocated Resources per VM**: 4 vCPUs, 8 GB memory;
- **Image**: `ubuntu:24.04`;
- **Kernel**: Linux v6.8.0-49-generic, cgroup2 enabled.

Instances can be accessed with the command `lxc shell <instance id>`.

Kubernetes Cluster. A single-node Kubernetes cluster was deployed using the RKE2 distribution in the VM provisioned by LXD. The single node will handle the control plane and worker pods. The cluster, as presented in Figure 5.1, was configured as follows:

- **Container Orchestrator**: Kubernetes v1.31.4-rke2r1;
- **Container Runtime**: containerd v1.7.23-k3s2;
- **Network Plugin**: Cilium v1.16.4 for Cilium Service Mesh, else Calico v3.29.1.

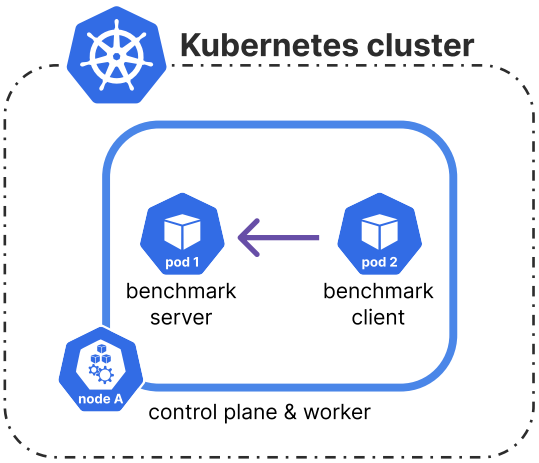


Figure 5.1: Benchmark architecture of the experimental evaluation.

[5] https://canonical.com/lxd
[6] https://documentation.ubuntu.com/lxd/en/latest/cloud-init/

Service Mesh. The experimental environment included four distinct service mesh setups for evaluation:

- *none*: no service mesh;
- `cilium`: Cilium v1.16.4;
- `istio-ambient`: Istio v1.24.2 in Ambient mode;
- `istio-sidecar`: Istio v1.24.2, in the default sidecar mode.

Optimization Setup. To deploy the proposed eBPF-based optimization strategy, precompiled release binaries were built to simplify installation and execution. The setup process is outlined in chapter Artifacts.

Workload Execution. To deploy workloads in the Kubernetes cluster, preconfigured Kubernetes manifests were utilized. These manifests contain the deployment specification, service, and configuration details required for each workload. The procedure for each workload is presented in chapter Artifacts.

5.5 Benchmark Results

This section presents the results of the experimental evaluation, emphasizing the impact of the proposed eBPF-based offloading strategy across various workloads and configurations. The benchmarks were designed to evaluate the system's networking performance and resource utilization under diverse conditions.

The evaluation involves three workloads—*file-transfer*, *ping-echo*, and *redis*—executed under four distinct service mesh configurations: no service mesh, `cilium`, `istio-ambient`, and `istio-sidecar`.

For each configuration, it was tested both with the eZtunnel optimization disabled (baseline) and enabled. This setup results in a total of 24 unique benchmark scenarios. Each workload was executed five times per scenario, totaling 120 executions.

Data collection includes six metrics (latency, jitter, FCT, throughput, CPU usage, and memory usage). To facilitate further exploration, the complete dataset—including raw measurements, data processing routines, and all visualizations—is openly available in the repository found in chapter Artifacts.

5.5.1 Latency and Jitter

The latency data from the *ping-echo* workload demonstrates significant improvements when using the eZtunnel offloading strategy across different service meshes, as observed in Figure 5.2 and Table 5.1. In the absence of a service

mesh (*none*), eZtunnel reduces latency by 38.0% at the 25th percentile (p25), 39.4% at the median (p50), 34.8% on average, and 38.2% at the 75th percentile (p75). Similar reductions are observed in other service mesh configurations, with eZtunnel achieving latency improvements of 26.6% to 41.9% at p25, 27.6% to 42.0% at p50, 22.9% to 41.9% on average, and 27.5% to 41.6% at p75 for the service mesh configurations `cilium`, `istio-ambient`, and `istio-sidecar`. Notably, `istio-sidecar` exhibits the highest baseline latencies, but eZtunnel still delivers substantial reductions of approximately 42% across all percentiles.

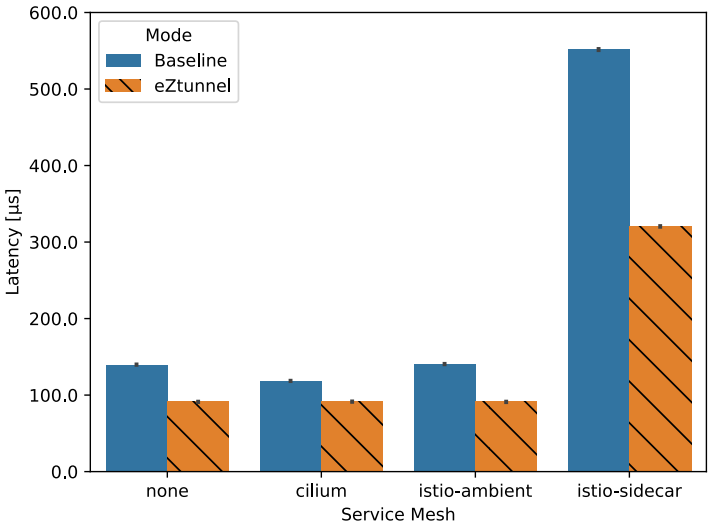


Figure 5.2: Latency of *ping-echo* workload.

Service Mesh	Mode	Latency (µs) p25	p50	average	p75
none	Baseline	97.7	121.4	139.8	157.8
	eZtunnel	60.6 (**-38.0%**)	73.5 (**-39.4%**)	91.2 (**-34.8%**)	97.5 (**-38.2%**)
`cilium`	Baseline	83.0	102.0	118.5	134.3
	eZtunnel	60.9 (**-26.6%**)	73.8 (**-27.6%**)	91.4 (**-22.9%**)	97.3 (**-27.5%**)
`istio-ambient`	Baseline	98.8	122.0	140.5	158.6
	eZtunnel	59.8 (**-39.5%**)	72.4 (**-40.6%**)	91.1 (**-35.2%**)	96.3 (**-39.3%**)
`istio-sidecar`	Baseline	406.5	495.6	551.5	615.3
	eZtunnel	236.1 (**-41.9%**)	287.4 (**-42.0%**)	320.4 (**-41.9%**)	359.5 (**-41.6%**)

Table 5.1: Latency of *ping-echo* workload.

The jitter results for the *ping-echo* workload, highlighted in Figure 5.3 and Table 5.2, presents varying effects of the eZtunnel offloading strategy across service mesh configurations. In the *none* service mesh configuration, eZtunnel achieves modest reductions in jitter, with decreases of 3.1% at p25 and 1.4% on average, while p50 remains nearly unchanged (0.2%) and p75 sees a slight increase of 0.7%. However, in `cilium`, eZtunnel leads to higher jitter, increasing p50 to 10.8%. Similarly,

in `istio-ambient`, p50 jitter rises by 1.5%. The most significant improvements occur in `istio-sidecar`, where eZtunnel reduces jitter by 35.6% at p50, addressing the higher baseline jitter in this configuration.

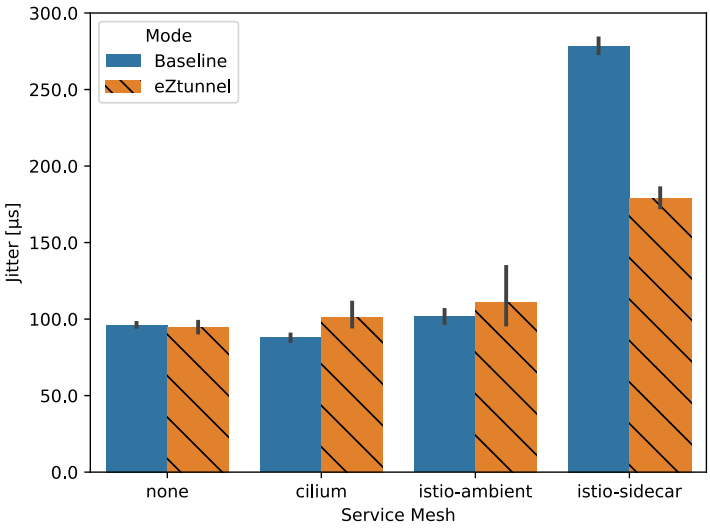


Figure 5.3: Jitter of *ping-echo* workload.

Service Mesh	Mode	Jitter (µs) p25	p50	average	p75
none	Baseline	95.2	96.0	96.1	96.6
	eZtunnel	92.3 (**-3.1%**)	95.8 (**-0.2%**)	94.7 (**-1.4%**)	97.3 (**0.7%**)
`cilium`	Baseline	87.6	87.9	87.9	89.2
	eZtunnel	96.0 (**9.6%**)	97.4 (**10.8%**)	101.3 (**15.2%**)	99.4 (**11.5%**)
`istio-ambient`	Baseline	99.8	102.3	101.6	103.3
	eZtunnel	94.7 (**-5.1%**)	103.9 (**1.5%**)	110.9 (**9.1%**)	104.5 (**1.1%**)
`istio-sidecar`	Baseline	276.7	278.3	278.5	279.3
	eZtunnel	171.8 (**-37.9%**)	179.1 (**-35.6%**)	178.8 (**-35.8%**)	180.0 (**-35.6%**)

Table 5.2: Jitter of *ping-echo* workload.

The impact of eZtunnel in latency for the *redis* workload are illustrated in Figure 5.4 and Table 5.3. Without a service mesh (*none*), eZtunnel significantly reduces latency, with improvements of 31.7% at p25, 32.2% at p50, 31.5% on average, and 31.2% at p75. For the service mesh `cilium`, the baseline latencies are already low, and eZtunnel maintains similar performance, with minor reductions of 1.4% across all percentiles to 1.6% on average. In `istio-ambient`, eZtunnel also achieves notable latency reductions of approximately 27.8% to 28.5% across p25, p50, average, and p75. However, in `istio-sidecar`, eZtunnel has minimal impact, with slight increase in latency ranging from 0.9% at p25 to 2.1% at p75.

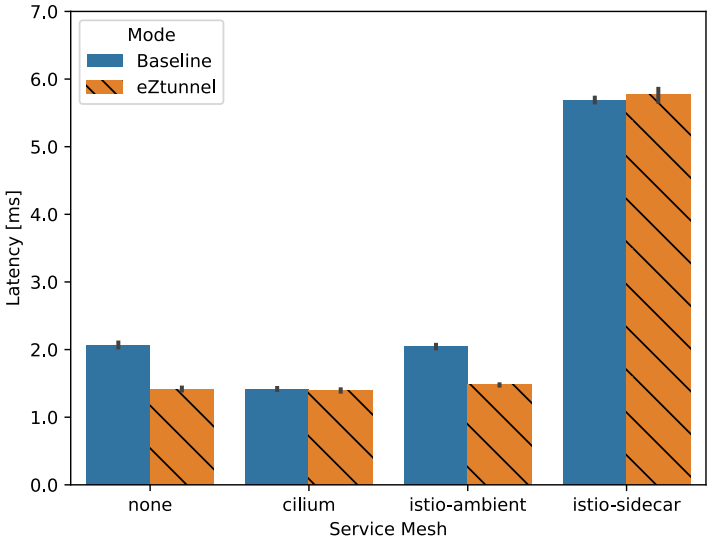


Figure 5.4: Latency of *redis* workload.

Service Mesh	Mode	Latency (ms) p25	p50	average	p75
none	Baseline	2.0	2.1	2.1	2.1
	eZtunnel	1.4 (**-31.7%**)	1.4 (**-32.2%**)	1.4 (**-31.5%**)	1.4 (**-31.2%**)
`cilium`	Baseline	1.4	1.4	1.4	1.4
	eZtunnel	1.4 (**-1.4%**)	1.4 (**-1.4%**)	1.4 (**-1.6%**)	1.4 (**-1.4%**)
`istio-ambient`	Baseline	2.0	2.1	2.0	2.1
	eZtunnel	1.5 (**-27.8%**)	1.5 (**-28.5%**)	1.5 (**-27.9%**)	1.5 (**-28.5%**)
`istio-sidecar`	Baseline	5.7	5.7	5.7	5.7
	eZtunnel	5.7 (**0.9%**)	5.8 (**1.7%**)	5.8 (**1.4%**)	5.8 (**2.1%**)

Table 5.3: Latency of *redis* workload.

Jitter is not available for the *redis* workload, and both latency and jitter data are unavailable for the *file-transfer* workload. This limitation arises because network-related metrics are collected by the client process itself, as outlined in the metrics description (section 5.3).

5.5.2 Flow Completion Time

FCT results for the *file-transfer* workload, depicted in Figure 5.5 and Table 5.4, reveal the impact of the eBPF-based offloading strategy across different service mesh configurations. In the service mesh configuration *none*, eZtunnel reduces FCT by 21.2% at p25, 21.0% at p50 and on average, and 21.1% at p75. Similarly, in the `istio-ambient` configuration, FCT improvements range from 19.3% at p25 to 20.4% at p75. For the `cilium` configuration, eZtunnel achieves reductions of 10.8% at p25, 11.7% at p50, 11.3% on average, and 12.2% at p75. However, in the `istio-sidecar` configuration, eZtunnel increases FCT by 20.8% at p25, 20.0% at p50, 21.2% on

average, and 23.5% at p75. Although it shown a negative impact in the `istio-sidecar` setup, where baseline values are already lower, FCT values with eZtunnel are within the range of 8.2s to 8.5s for all service mesh setups.

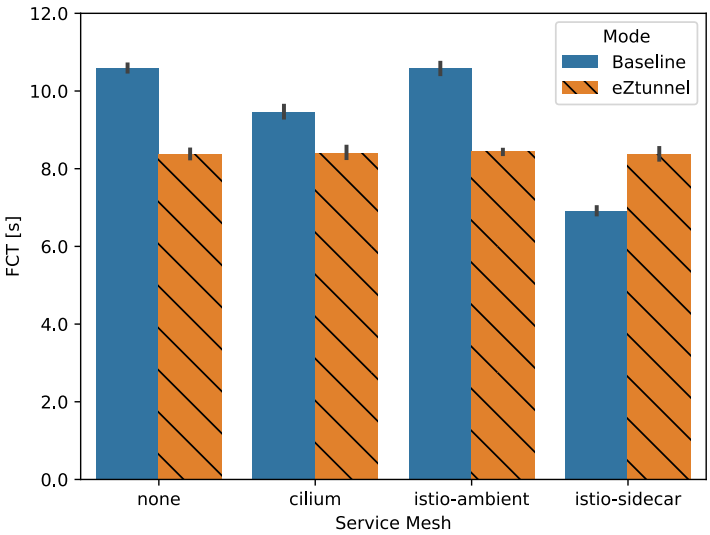


Figure 5.5: Flow Completion Time (FCT) of *file-transfer* workload.

Service Mesh	Mode	FCT (s) p25	p50	average	p75
none	Baseline	10.5	10.5	10.6	10.7
	eZtunnel	8.3 (**-21.2%**)	8.3 (**-21.0%**)	8.4 (**-21.0%**)	8.4 (**-21.1%**)
`cilium`	Baseline	9.3	9.5	9.5	9.6
	eZtunnel	8.3 (**-10.8%**)	8.4 (**-11.7%**)	8.4 (**-11.3%**)	8.4 (**-12.2%**)
`istio-ambient`	Baseline	10.4	10.6	10.6	10.7
	eZtunnel	8.4 (**-19.3%**)	8.5 (**-20.1%**)	8.4 (**-20.2%**)	8.5 (**-20.4%**)
`istio-sidecar`	Baseline	6.8	6.9	6.9	6.9
	eZtunnel	8.2 (**20.8%**)	8.3 (**20.0%**)	8.4 (**21.2%**)	8.5 (**23.5%**)

Table 5.4: Flow Completion Time (FCT) of *file-transfer* workload.

The FCT results for the *ping-echo* workload, illustrated in Figure 5.6 and Table 5.5, followed a consistent down trend with the usage of the optimization. In the *none* configuration, eZtunnel reduces FCT by 34.2% at p25, 34.7% at p50, 34.4% on average, and 34.7% at p75. Similarly, in the `istio-ambient` configuration, FCT improvements range from 34.7% at p75 to 35.1% at p25. For the `cilium` configuration, it achieves reductions of 22.0% at p25, 22.7% at p50 and on average, and 23.6% at p75. The most significant improvements are observed in the `istio-sidecar` configuration, where eZtunnel reduces FCT by 41.2% at p50. For the configurations *none*, `cilium`, and `istio-ambient`, FCT values kept stable around 9.5s and 9.7s across all the percentiles.

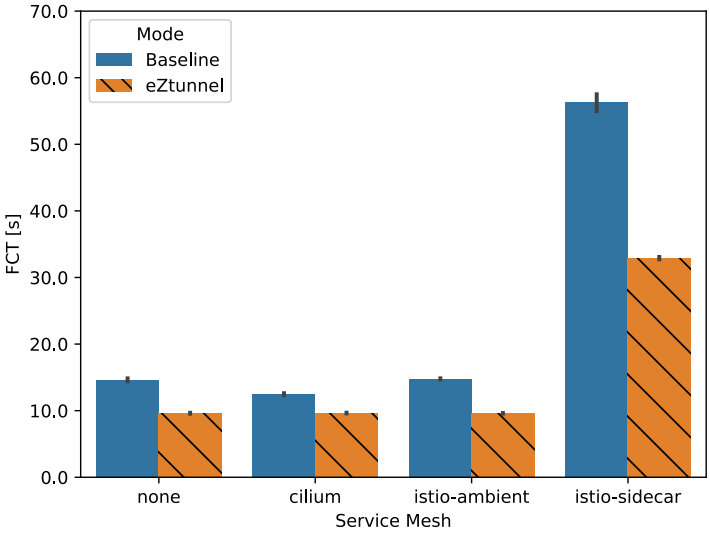


Figure 5.6: Flow Completion Time (FCT) of *ping-echo* workload.

Service Mesh	Mode	FCT (s) p25	p50	average	p75
none	Baseline	14.5	14.8	14.6	14.8
	eZtunnel	9.5 (**-34.2%**)	9.7 (**-34.7%**)	9.6 (**-34.4%**)	9.7 (**-34.7%**)
`cilium`	Baseline	12.3	12.4	12.5	12.6
	eZtunnel	9.6 (**-22.0%**)	9.6 (**-22.7%**)	9.6 (**-22.7%**)	9.7 (**-23.6%**)
`istio-ambient`	Baseline	14.7	14.7	14.8	14.8
	eZtunnel	9.5 (**-35.1%**)	9.6 (**-34.9%**)	9.6 (**-34.9%**)	9.6 (**-34.7%**)
`istio-sidecar`	Baseline	55.2	56.1	56.3	57.8
	eZtunnel	32.8 (**-40.7%**)	33.0 (**-41.2%**)	32.9 (**-41.5%**)	33.1 (**-42.9%**)

Table 5.5: Flow Completion Time (FCT) of *ping-echo* workload.

The FCT values for the *redis* workload, presented in Figure 5.7 and Table 5.6, show varying impacts offloading strategy across service mesh setups. In the *none* configuration, eZtunnel reduces FCT by 30% at p25, p50, and p75, and by 31.7% on average. Similarly, in the `istio-ambient` configuration, FCT improvements are consistent, with reductions of 30% at p25, p50, and p75, and 29.3% on average. For the `cilium` configuration, Baseline values are already lower than the other setups. However, eZtunnel still achieves modest reductions of 7.1% at p25 and 1.4% on average, while p50 and p75 remain unchanged. In contrast, the `istio-sidecar` configuration sees a slight increase in FCT, with eZtunnel adding 1.3% at p25 and on average, and 1.8% at p50 and p75. Similar to the *ping-echo* workload, the FCT values of the *redis* workload for configurations *none*, `cilium`, and `istio-ambient`, varied between 13s and 14s across the percentiles.

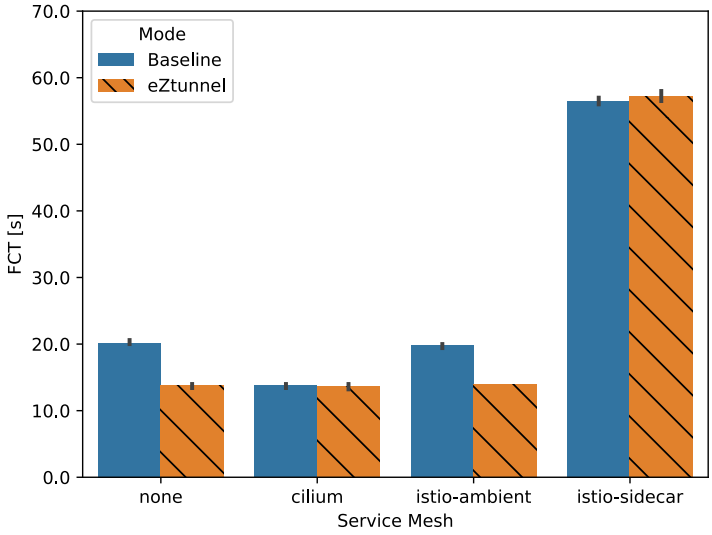


Figure 5.7: Flow Completion Time (FCT) of *redis* workload.

Service Mesh	Mode	FCT (s) p25	p50	average	p75
none	Baseline	20	20	20.2	20
	eZtunnel	14 (**-30%**)	14 (**-30%**)	13.8 (**-31.7%**)	14 (**-30%**)
`cilium`	Baseline	14	14	13.8	14
	eZtunnel	13 (**-7.1%**)	14 (**0%**)	13.6 (**-1.4%**)	14 (**0%**)
`istio-ambient`	Baseline	20	20	19.8	20
	eZtunnel	14 (**-30%**)	14 (**-30%**)	14 (**-29.3%**)	14 (**-30%**)
`istio-sidecar`	Baseline	56	56.5	56.5	57
	eZtunnel	56.8 (**1.3%**)	57.5 (**1.8%**)	57.2 (**1.3%**)	58 (**1.8%**)

Table 5.6: Flow Completion Time (FCT) of *redis* workload.

5.5.3 Throughput

We observed an upward trend with optimization enabled for the throughput performance. Values (in RPS) for the *ping-echo* workload, shown in Figure 5.8 and Table 5.7, demonstrate substantial performance improvements. In the *none* configuration, the optimization increases RPS by 65.1% at p50, and 60.1% on average. Likewise, in the `istio-ambient` configuration, RPS improvements are in the order of 68.5% at p50 and 62.4% on average. For the `cilium` configuration, eZtunnel achieves increases of 38.1% at p50 and 34.8% on average. The most dramatic improvements occur in the `istio-sidecar` configuration, where the offloading optimization boosts RPS by 72.4% at p50 and 70.9% on average. The optimized scenarios consistently outperformed their baseline counterparts.

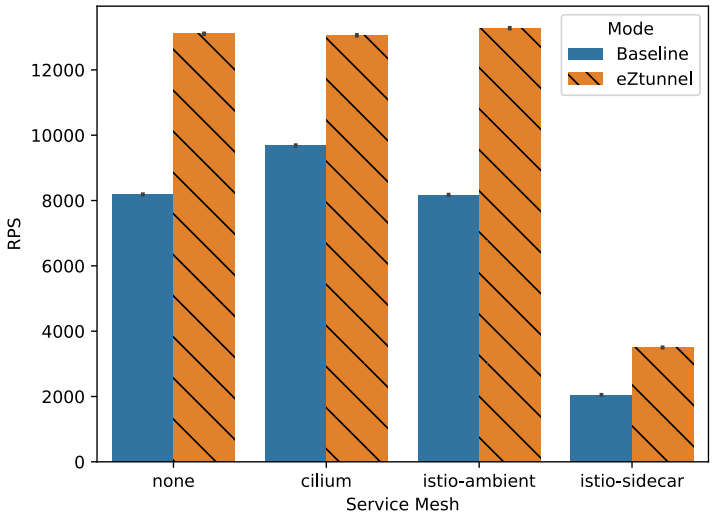


Figure 5.8: Requests per Second (RPS) of *ping-echo* workload.

Service Mesh	Mode	RPS p25	RPS p50	RPS average	RPS p75
none	Baseline	6338.0	8238.7	8186.1	10240.6
	eZtunnel	10260.2 (**61.9%**)	13600.8 (**65.1%**)	13104.6 (**60.1%**)	16514.2 (**61.3%**)
`cilium`	Baseline	7447.3	9803.7	9685.2	12055.0
	eZtunnel	10275.5 (**38.0%**)	13541.5 (**38.1%**)	13058.0 (**34.8%**)	16422.5 (**36.2%**)
`istio-ambient`	Baseline	6303.4	8196.4	8172.3	10124.0
	eZtunnel	10385.2 (**64.8%**)	13808.1 (**68.5%**)	13274.4 (**62.4%**)	16728.0 (**65.2%**)
`istio-sidecar`	Baseline	1625.3	2017.9	2049.1	2459.8
	eZtunnel	2781.4 (**71.1%**)	3479.5 (**72.4%**)	3502.1 (**70.9%**)	4235.2 (**72.2%**)

Table 5.7: Requests per Second (RPS) of *ping-echo* workload.

Throughput (measured in OPS) for the *redis* workload, depicted in Figure 5.9 and Table 5.8, reveal varying impacts of the offloading strategy. In the *none* configuration, eZtunnel increases OPS by 45.3% at p25, 47.4% at p50, 45.8% on average, and 46.3% at p75. OPS was also improved in the `istio-ambient` configuration, ranging from 38.5% at p75 to 39.3% at p25 and p50. For the `cilium` configuration, it achieves

Service Mesh	Mode	OPS p25	OPS p50	OPS average	OPS p75
none	Baseline	47902	48005	48248.8	48594
	eZtunnel	69583 (**45.3%**)	70765 (**47.4%**)	70342.8 (**45.8%**)	71096 (**46.3%**)
`cilium`	Baseline	69571	70428	70332.6	70701
	eZtunnel	70866 (**1.9%**)	71332 (**1.3%**)	71566.4 (**1.8%**)	71756 (**1.5%**)
`istio-ambient`	Baseline	48279	48314	48737.2	48781
	eZtunnel	67252 (**39.3%**)	67290 (**39.3%**)	67553.6 (**38.6%**)	67577 (**38.5%**)
`istio-sidecar`	Baseline	17450	17528.5	17542.5	17621
	eZtunnel	17072.8 (**-2.2%**)	17232 (**-1.7%**)	17305.2 (**-1.4%**)	17464.5 (**-0.9%**)

Table 5.8: Operations per Second (OPS) of *redis* workload.

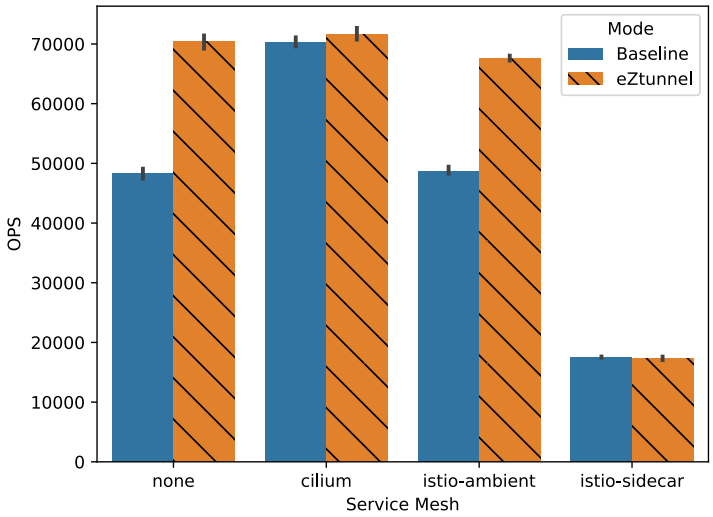


Figure 5.9: Operations per Second (OPS) of *redis* workload.

small gains of 1.3% at p50 and 1.8% on average. In contrast, in the `istio-sidecar` configuration, eZtunnel results in slight decreases in OPS, with reductions of 2.2% at p25, 1.7% at p50, 1.4% on average, and 0.9% at p75.

Throughput values are not available for the *file-transfer* workload, as the *ncat* client does not measure it.

5.5.4 CPU Usage

The CPU usage results reveal varying impacts of the offloading strategy. The first result, the *file-transfer* workload, as shown in Figure 5.10 and Table 5.9, presents mostly decreased values for CPU usage. For the `istio-ambient` configuration, eZtunnel achieves a more significant reduction of 12.5%. In the *none* configuration, eZtunnel reduces CPU usage by 5.8%, while in the `cilium` configuration, the reduction is minimal at 0.7%. However, in the `istio-sidecar` configuration, CPU usage increases by 13.7% with eZtunnel.

Service Mesh	Mode	CPU Usage (s)
none	Baseline	86.4
	eZtunnel	81.4 (**-5.8%**)
`cilium`	Baseline	83.4
	eZtunnel	82.8 (**-0.7%**)
`istio-ambient`	Baseline	103.0
	eZtunnel	90.1 (**-12.5%**)
`istio-sidecar`	Baseline	100.7
	eZtunnel	114.5 (**13.7%**)

Table 5.9: CPU usage of *file-transfer* workload.

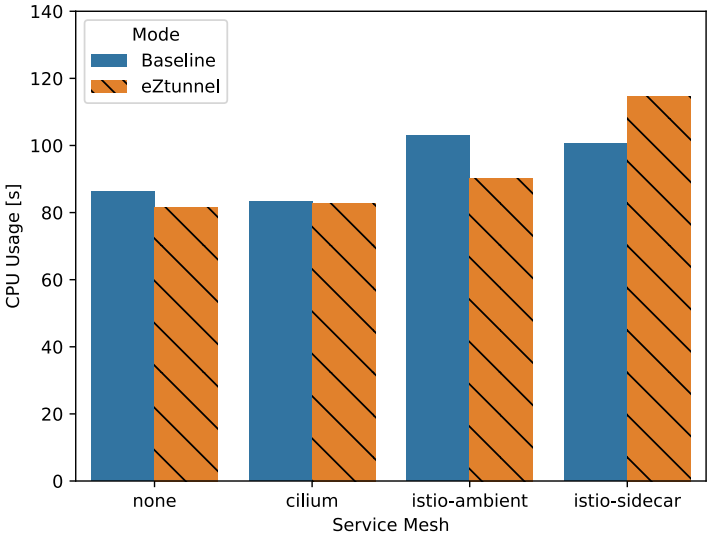


Figure 5.10: CPU usage of *file-transfer* workload.

For the *ping-echo* workload, illustrated in Figure 5.11 and Table 5.10, eZtunnel consistently reduces CPU usage across all service mesh configurations. The `istio-sidecar` configuration sees a substantial reduction of 23.2%, and the `istio-ambient` configuration achieves a 21.9% decrease in CPU usage. In the *none* configuration, CPU usage decreases by 16.2%. The smallest reduction, but still significative, is in the `cilium` configuration, with a 12.1% decrease.

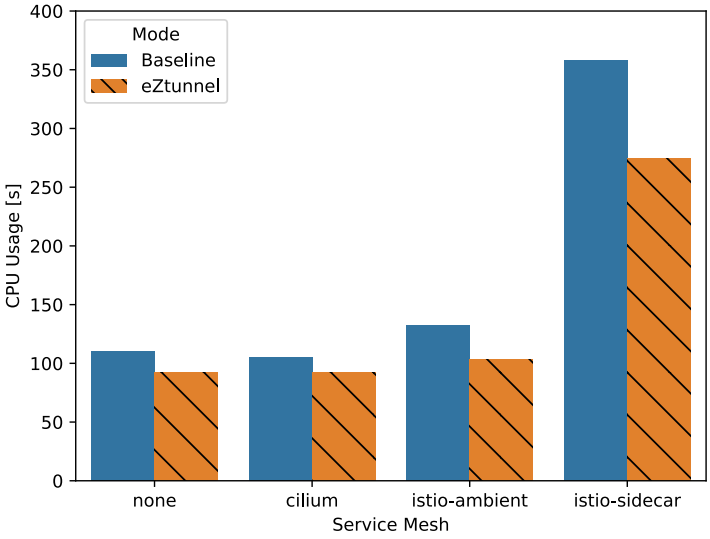


Figure 5.11: CPU usage of *ping-echo* workload.

Service Mesh	Mode	CPU Usage (s)
none	Baseline	110.5
	eZtunnel	92.6 (**-16.2%**)
`cilium`	Baseline	104.9
	eZtunnel	92.3 (**-12.1%**)
`istio-ambient`	Baseline	132.2
	eZtunnel	103.2 (**-21.9%**)
`istio-sidecar`	Baseline	357.8
	eZtunnel	274.7 (**-23.2%**)

Table 5.10: CPU usage of *ping-echo* workload.

The CPU usage results for the *redis* workload, depicted in Figure 5.12 and Table 5.11, show consistently increased usage. In the *none* configuration, eZtunnel increases CPU usage by 8.1%, The increase is more pronounced in the `cilium` configuration, with 29.2% increase, followed by 21.7% for the `istio-sidecar`. The smallest increase in CPU usage is the `istio-ambient` configuration, with CPU usage rise of 3.0%.

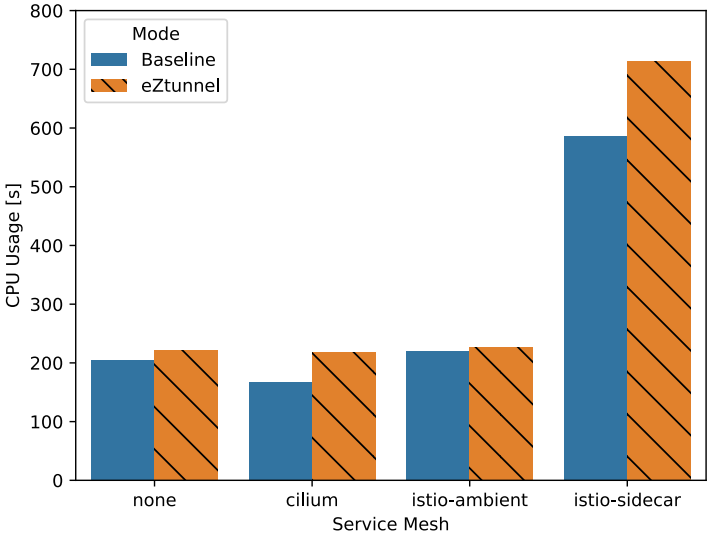


Figure 5.12: CPU usage of *redis* workload.

Service Mesh	Mode	CPU Usage (s)
none	Baseline	204.4
	eZtunnel	220.9 (**8.1%**)
`cilium`	Baseline	167.9
	eZtunnel	216.9 (**29.2%**)
`istio-ambient`	Baseline	219.7
	eZtunnel	226.4 (**3.0%**)
`istio-sidecar`	Baseline	586.4
	eZtunnel	713.9 (**21.7%**)

Table 5.11: CPU usage of *redis* workload.

5.5.5 Memory Usage

Memory usage of eZtunnel was light across all workloads and service mesh variations. The results for the *file-transfer* workload, presented in Figure 5.13 and Table 5.12, in the `istio-ambient` configuration, memory usage increases by 11.6 MB at p50—the smallest impact among the configurations. In the *none* configuration, memory usage rises by 24.8 MB at p50. For `istio-sidecar`, increases are more pronounced, with memory usage rising by 33.7 MB at p50. The `cilium` configuration sees the highest increase, with memory usage growing by 41.4 MB at p50.

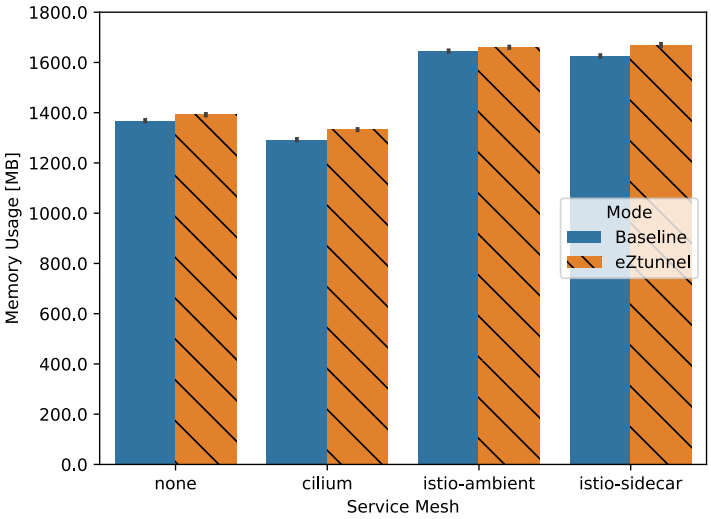


Figure 5.13: Memory usage of *file-transfer* workload.

Service Mesh	Mode	Memory Usage (MB) p25	p50	average	p75
none	Baseline	1354.9	1368.1	1368.0	1379.7
	eZtunnel	1381.6 (**+26.7**)	1392.9 (**+24.8**)	1392.0 (**+24.1**)	1401.6 (**+21.9**)
`cilium`	Baseline	1283.2	1291.1	1292.7	1302.4
	eZtunnel	1327.3 (**+44.1**)	1332.5 (**+41.4**)	1332.5 (**+39.8**)	1337.0 (**+34.6**)
`istio-ambient`	Baseline	1632.2	1647.5	1644.8	1656.8
	eZtunnel	1650.8 (**+18.5**)	1659.0 (**+11.6**)	1660.2 (**+15.4**)	1669.3 (**+12.5**)
`istio-sidecar`	Baseline	1614.2	1626.5	1625.4	1637.5
	eZtunnel	1648.9 (**+34.6**)	1660.2 (**+33.7**)	1668.6 (**+43.2**)	1689.6 (**+52.2**)

Table 5.12: Memory usage of *file-transfer* workload.

For the *ping-echo* workload, as illustrated in Figure 5.14 and Table 5.13, the `istio-ambient` configuration exhibits a smaller increase in memory usage of 16.8 MB at p50. Similarly, the `istio-sidecar` configuration experiences increases of 20.0 MB at p50. The `cilium` configuration exhibits 25.8 MB of additional memory usage at p50. In the *none* configuration, memory usage increases by 31.8 MB at p50—the biggest increase.

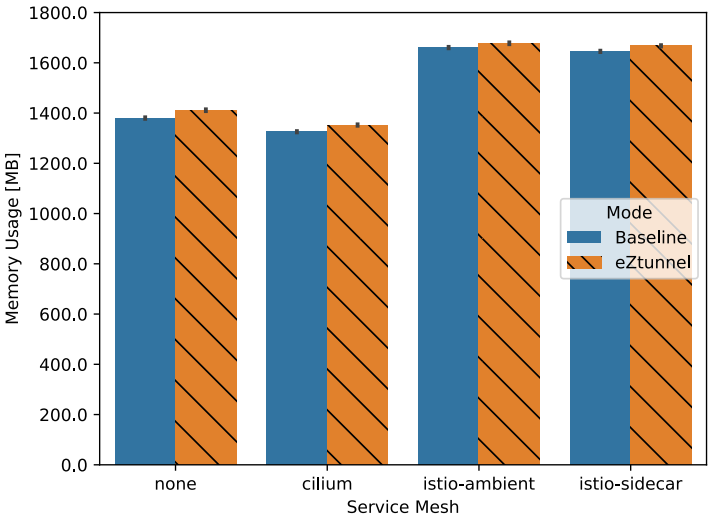


Figure 5.14: Memory usage of *ping-echo* workload.

Service Mesh	Mode	Memory Usage (MB) p25	p50	average	p75
none	Baseline	1373.6	1382.6	1380.2	1387.6
	eZtunnel	1404.5 (**+30.9**)	1414.5 (**+31.8**)	1411.4 (**+31.2**)	1422.1 (**+34.5**)
`cilium`	Baseline	1316.8	1325.1	1325.7	1334.6
	eZtunnel	1345.5 (**+28.7**)	1350.9 (**+25.8**)	1352.4 (**+26.7**)	1360.6 (**+26.0**)
`istio-ambient`	Baseline	1652.2	1662.5	1660.7	1673.6
	eZtunnel	1670.0 (**+17.8**)	1679.3 (**+16.8**)	1677.7 (**+17.0**)	1689.7 (**+16.1**)
`istio-sidecar`	Baseline	1636.2	1650.7	1645.7	1661.6
	eZtunnel	1663.0 (**+26.7**)	1670.7 (**+20.0**)	1667.6 (**+21.8**)	1678.0 (**+16.4**)

Table 5.13: Memory usage of *ping-echo* workload.

In the *redis* workload, shown in Figure 5.15 and Table 5.14, the *none* configuration sees the largest increase, with memory usage growing by 61.3 MB at p50, next to `istio-ambient` with 60.5 MB of increase at p50. For the `cilium` configuration, memory usage increases by 48.6 MB at p50. In the `istio-sidecar` configuration, memory usage rises by 47.7 MB at p50.

Service Mesh	Mode	Memory Usage (MB) p25	p50	average	p75
none	Baseline	1393.1	1464.2	1442.0	1479.4
	eZtunnel	1466.1 (**+73.0**)	1525.5 (**+61.3**)	1503.4 (**+61.4**)	1537.9 (**+58.5**)
`cilium`	Baseline	1333.3	1428.9	1392.9	1434.9
	eZtunnel	1414.9 (**+81.6**)	1477.6 (**+48.6**)	1452.4 (**+59.5**)	1485.8 (**+50.9**)
`istio-ambient`	Baseline	1637.8	1742.0	1705.2	1759.7
	eZtunnel	1738.9 (**+101.1**)	1802.5 (**+60.5**)	1773.6 (**+68.5**)	1813.3 (**+53.6**)
`istio-sidecar`	Baseline	1765.3	1788.3	1769.7	1813.9
	eZtunnel	1813.4 (**+48.1**)	1836.1 (**+47.7**)	1816.5 (**+46.8**)	1860.8 (**+46.9**)

Table 5.14: Memory usage of *redis* workload.

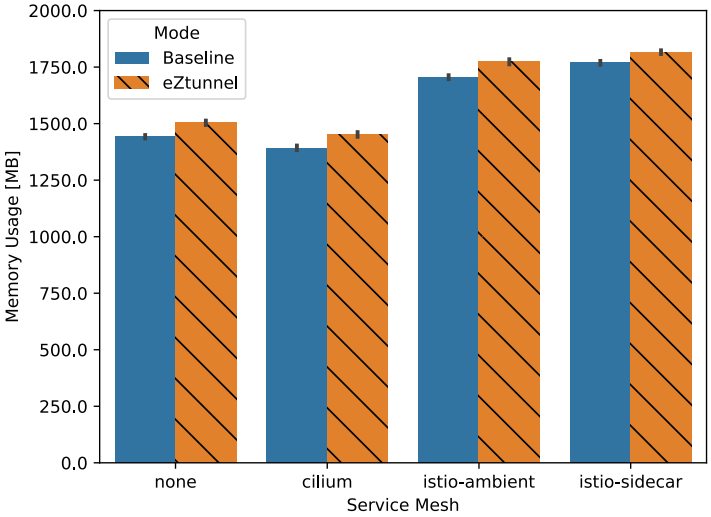


Figure 5.15: Memory usage of *redis* workload.

5.6 Discussion of Results

The benchmark results highlight the diverse impacts of optimization across various workloads and metrics, demonstrating significant performance improvements alongside some limitations. Key observations include:

Latency and Jitter. The eBPF-based optimization strategy significantly reduces latency across all workloads, with the most notable improvements in high-frequency, low-latency workloads like *ping-echo* and *redis*. In the *ping-echo* workload, latency reductions ranged from 27.6% to 42% at p50 across service mesh configurations, with the largest gains in `istio-sidecar`. Jitter improvements were more variable, with significant reductions in `istio-sidecar` (35.6% at p50) but slight increases in `cilium` and `istio-ambient`. For `redis`, latency improvements were substantial in *none* and `istio-ambient` (up to 32%) but minimal in `istio-sidecar`. These results highlight eZtunnel's effectiveness in reducing latency in general.

FCT. eZtunnel consistently reduces FCT for low-latency and bulk data transfer workloads. In the *file-transfer* workload, FCT improvements ranged from 11.7% to 21% at p50 in *none*, `cilium`, and `istio-ambient`, but increased by 20% in `istio-sidecar`. For the *ping-echo* workload, FCT reductions were consistent across configurations, with the largest improvement in `istio-sidecar` (41.2% at p50). In the *redis* workload, FCT improved by up to 30% in *none* and `istio-ambient` but saw slight increases in `istio-sidecar`. These findings suggest that eZtunnel is highly effective in reducing FCT, except in sidecar-based service meshes.

Throughput. eZtunnel significantly improves throughput for high-frequency workloads. In the *ping-echo* workload, RPS increased by up to 72.4% in `istio-sidecar`, with consistent improvements across all configurations. For the *redis* workload, OPS improved by up to 47.4% in *none* and `istio-ambient`, but saw slight decreases in `istio-sidecar`. These results demonstrate eZtunnel's ability to enhance request transferring rates, particularly in environments with high-frequency communication.

CPU Usage. The impact of eZtunnel on CPU usage varies by workload and service mesh configuration. For the *ping-echo* workload, CPU usage decreased by up to 23.2% in `istio-sidecar`, indicating reduced overhead for high-frequency workloads. However, in the *redis* workload, CPU usage increased by up to 29.2% in `cilium`, suggesting additional overhead for data-intensive operations. In the *file-transfer* workload, CPU usage decreased in *none* and `istio-ambient` but increased in `istio-sidecar`. These results highlight a trade-off between performance improvements and CPU resource utilization.

Memory Usage. eZtunnel introduces modest memory footprint across all workloads and configurations. The largest increases were observed in the *redis* workload (up to 61.3 MB in *none*) and the *file-transfer* workload (up to 41.4 MB in `cilium`). For the *ping-echo* workload, memory usage increased by up to 31.8 MB in *none*. While these increases are relatively small, they represent a consistent trade-off for the performance gains achieved by eZtunnel.

5.7 Summary

Throughout this chapter, we examined the benchmarking scenario, focusing on the impact of the proposed eBPF-based optimization on network performance and resource usage in a controlled environment. The primary aim was to assess the effectiveness of the optimization in enhancing intra-node communication performance in cloud-native applications.

The experimental setup adhered to well-defined guidelines to ensure that the results were meaningful, reproducible, and representative of real-world use cases. The selected workloads were designed to reflect common network patterns in cloud-native environments, covering diverse scenarios and enabling reliable comparisons. The use of free and open-source software reinforces the reproducibility of the findings.

Three distinct workloads were evaluated to capture diverse networking behaviors. The *file-transfer* workload simulates high-throughput scenarios typical of data-intensive applications, such as file sharing and media streaming; *ping-echo* represents

low-latency, high-frequency request-response patterns common in microservice interactions; and *redis* emulates high-concurrency, data-intensive operations often encountered in caching or session management services. These workloads offered a comprehensive view of service mesh performance under varied conditions, showcasing both the strengths and limitations of the optimization.

The evaluation utilized metrics that reflect critical aspects of networking performance and resource efficiency, including latency, jitter, FCT, throughput, CPU and memory usage. These metrics provided a complete overview of the system's behavior under optimized and unoptimized scenarios.

The optimization delivered substantial improvements across most scenarios. Significant reductions were observed for FCT. *File-transfer* achieved FCT improvements of 21% at most; *ping-echo* 41.2%; and *redis* 30%.

The optimization reduced latency in nearly all configurations, especially in the *ping-echo* workload, where *istio-sidecar* saw dramatic improvements, a reduction of 42.0%. *Redis* workload was also improved by 32.2%. Jitter improvement of *ping-echo* workload was of 35.6%.

Gains in throughput were evident in all workloads, with the *ping-echo* workload achieving up to 68.5% in the `istio-ambient` configuration and *redis* improving by up to 47.4% in the no service mesh configuration.

CPU usage was variable across workloads and service mesh setups. In the *file-transfer* workload, the CPU usage of eZtunnel ranged between +13.7% and -12.5%. In contrast, CPU usage in the *ping-echo* workload was consistently lowered by at most 23.2%, while in *redis* it was negatively increased by at most 29.2%.

While memory usage increased between 11.6 MB$\sim$60.5 MB due to the eBPF programs leveraging eBPF maps, the improvements in networking metrics outweighed this overhead in most scenarios.

The results validate the efficacy of the eBPF-based optimization in enhancing network performance in microservice architectures. The diversity of workloads demonstrated the optimization's broad applicability and robustness, while also highlighting areas for refinement, such as isolated scenarios where improvements were limited. By achieving consistent performance gains across various metrics, this work presents a practical and scalable approach to optimizing modern distributed systems.

Chapter 6

Conclusions and Future Work

This study presents a comprehensive exploration of the potential for eBPF-based optimizations to enhance networking performance in cloud-native environments. By addressing the inherent challenges of Kubernetes networking and service mesh overheads, the presented approach leverages eBPF's capabilities to improve intra-node communication latency, jitter, FCT, and throughput while maintaining compatibility with modern service mesh designs.

Through the analysis of background concepts, we highlighted the significant overhead introduced by traditional service meshes, particularly in sidecar-based architectures. These findings set the stage for the development of our proposal, eZtunnel, an offloading strategy that integrates seamlessly into both sidecar-based and sidecar-less service mesh configurations without requiring cluster redeployment. By employing eBPF programs such as *SockOps* and *SkMsg*, eZtunnel bypasses the traditional kernel networking stack, leading to measurable performance gains across a range of metrics.

Our experimental evaluation demonstrated the effectiveness of this optimization in real-world scenarios. The proposed optimization strategy consistently delivered improvements across diverse workloads—including *file-transfer*, *ping-echo*, and *redis*—and service mesh configurations—no service mesh, `cilium`, `istio-ambient`, and `istio-sidecar`. While these results underscore the robustness of our approach, we also identified specific cases where the optimization's impact was limited, pointing to opportunities for further refinement.

Despite the increased memory usage inherent to eBPF Maps, the benefits in terms of networking performance far outweigh the trade-offs, positioning eZtunnel as a cost-effective and scalable solution for Kubernetes-based environments. This work provides a foundation for the adoption of eBPF as a tool for optimizing microservices architectures.

While the results of this study are promising, they also open paths for further exploration and enhancement:

Addressing Optimization Limitations. Certain scenarios, such as FCT improvements in specific service mesh configurations (*e.g.*, `istio-sidecar`), revealed areas where the optimization's impact was constrained. Future work can focus on refining eBPF program logic or exploring complementary techniques to address these limitations.

Comprehensive Resource Management. While memory overheads were deemed acceptable in this study, future efforts could aim to further optimize resource usage, potentially exploring alternative eBPF Map configurations or dynamic memory allocation strategies.

Scaling to Multi-Node Clusters. The current implementation focuses on intra-node optimization. Extending this approach to optimize inter-node communication in multi-node Kubernetes clusters could provide even greater performance benefits.

Hardware Offloading. Exploring hardware offloading solutions, such as using SmartNICs, represents a promising direction. By delegating inter-node communication tasks to specialized hardware, it may be possible to achieve even lower latency and higher throughput while reducing CPU utilization. This approach could complement eBPF-based optimizations, leveraging the advanced capabilities of SmartNICs to further enhance performance in high-demand cloud-native environments.

Expanded Benchmarking. Incorporating additional workloads and scenarios—such as machine learning pipelines, real-time analytics, IoT applications, or 5G Core [63]—would provide a more comprehensive understanding of the optimization's applicability across diverse use cases.

Security Enhancements. Investigating the security implications of eBPF-based optimizations and developing mechanisms to safeguard against potential vulnerabilities would enhance the robustness of the proposed approach.

By building on the foundations presented in this study, future research can further explore the potential of eBPF and related offloading technologies to address the complex demands of modern distributed systems. This work not only demonstrates the feasibility of eBPF-based optimization but also paves the way for its broader adoption in cloud-native infrastructures.

Artifacts

- **Git Repository**: https://github.com/arthursimas1/mesh-fastpath
- **Research Data Repository**: https://doi.org/10.25824/redu/JVJ0BP

Bibliography

[1] D. Gannon, R. Barga, and N. Sundaresan, "Cloud-native applications," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 16–21, 2017. DOI: 10.1109/MCC.2017.4250939.

[2] R. Vaño, I. Lacalle, P. Sowiński, R. S-Julián, and C. E. Palau, "Cloud-native workload orchestration at the edge: A deployment review and future directions," *Sensors*, vol. 23, no. 4, 2023, ISSN: 1424-8220. DOI: 10.3390/s23042215.

[3] "Kubernetes website," Kubernetes. (), [Online]. Available: https://kubernetes.io.

[4] "Istio," Istio Authors. (), [Online]. Available: https://istio.io/.

[5] X. Zhu, G. She, B. Xue, Y. Zhang, Y. Zhang, X. K. Zou, X. Duan, P. He, A. Krishnamurthy, M. Lentz, D. Zhuo, and R. Mahajan, "Dissecting overheads of service mesh sidecars," in *Proceedings of the 2023 ACM Symposium on Cloud Computing*, ser. SoCC '23, Santa Cruz, CA, USA: Association for Computing Machinery, 2023, pp. 142–157, ISBN: 9798400703874. DOI: 10.1145/3620678.3624652.

[6] W. Yang, P. Chen, G. Yu, H. Zhang, and H. Zhang, "Network shortcut in data plane of service mesh with ebpf," *Journal of Network and Computer Applications*, vol. 222, Feb. 2024, ISSN: 1084-8045. DOI: 10.1016/j.jnca.2023.103805.

[7] J. Howard, E. J. Jackson, Y. Kohavi, I. Levine, J. Pettit, and L. Sun. "Introducing ambient mesh: A new dataplane mode for istio without sidecars," Istio Authors. (Sep. 2022), [Online]. Available: https://istio.io/latest/blog/2022/introducing-ambient-mesh/.

[8] A. Sriraman, A. Dhanotia, and T. F. Wenisch, "SoftSKU: Optimizing Server Architectures for Microservice Diversity @Scale," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19, Phoenix, Arizona: Association for Computing Machinery, 2019, pp. 513–526, ISBN: 9781450366694. DOI: 10.1145/3307650.3322227.

[9] A. Sriraman and A. Dhanotia, "Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20, Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 733–750, ISBN: 9781450371025. DOI: 10.1145/3373376.3378450.

[10] G. Aceto, A. Botta, W. de Donato, and A. Pescapè, "Cloud monitoring: A survey," *Computer Networks*, vol. 57, no. 9, pp. 2093–2115, 2013, ISSN: 1389-1286. DOI: 10.1016/j.comnet.2013.04.001.

[11] B. Chen and Z. M. Jiang, "A Survey of Software Log Instrumentation," *ACM Comput. Surv.*, vol. 54, no. 4, May 2021, ISSN: 0360-0300. DOI: 10.1145/3448976.

[12] K. Rodrigues, G. Bruno, K. Cardoso, S. Corrêa, and C. Both, "Uma Investigação Empírica sobre Observabilidade em Sistemas 5G Nativos de Nuvem," in *Anais do XL Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, Fortaleza: SBC, 2022, pp. 252–265. DOI: 10.5753/sbrc.2022.222304.

[13] "What is cloud native?" Oracle. (), [Online]. Available: https://www.oracle.com/cloud/cloud-native/what-is-cloud-native/.

[14] "What is cloud native?" Google. (), [Online]. Available: https://cloud.google.com/learn/what-is-cloud-native.

[15] "What is kubernetes?" IBM. (), [Online]. Available: https://www.ibm.com/topics/kubernetes.

[16] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *ACM Queue*, vol. 14, pp. 70–93, 2016. [Online]. Available: http://queue.acm.org/detail.cfm?id=2898444.

[17] "Microservices-demo: Sample cloud-first application with 10 microservices showcasing kubernetes, istio, and grpc," Google. (), [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo.

[18] A. Ducastel. "Benchmark results of kubernetes network plugins (cni) over 40gbit/s network [2024]." (2024), [Online]. Available: https://itnext.io/benchmark-results-of-kubernetes-network-plugins-cni-over-40gbit-s-network-2024-156f085a5e4e.

[19] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19, Providence, RI, USA: Association for Computing Machinery, 2019, pp. 3–18, ISBN: 9781450362405. DOI: 10.1145/3297858.3304013.

[20] *BPF Documentation*, Kernel Developers. [Online]. Available: https://docs.kernel.org/bpf/.

[21] "eBPF," eBPF Foundation. (), [Online]. Available: https://ebpf.io/.

[22] M. A. M. Vieira, M. S. Castanho, R. D. G. Pacífico, E. R. S. Santos, E. P. M. C. Júnior, and L. F. M. Vieira, "Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications," *ACM Comput. Surv.*, vol. 53, no. 1, Feb. 2020, ISSN: 0360-0300. DOI: 10.1145/3371038.

[23] *Program Types*, Cilium. [Online]. Available: https://docs.cilium.io/en/stable/bpf/progtypes/.

[24] M. S. Brunella, G. Belocchi, M. Bonola, S. Pontarelli, G. Siracusano, G. Bianchi, A. Cammarano, A. Palumbo, L. Petrucci, and R. Bifulco, "hXDP: Efficient Software Packet Processing on FPGA NICs," *Commun. ACM*, vol. 65, no. 8, pp. 92–100, Jul. 2022, ISSN: 0001-0782. DOI: 10.1145/3543668.

[25] A. Sadiq, H. J. Syed, A. A. Ansari, A. O. Ibrahim, M. Alohaly, and M. Elsadig, "Detection of denial of service attack in cloud based kubernetes using eBPF," *Appl. Sci. (Basel)*, vol. 13, no. 8, p. 4700, Apr. 2023. DOI: 10.3390/app13084700.

[26] J.-B. Lee, T.-H. Yoo, E.-H. Lee, B.-H. Hwang, S.-W. Ahn, and C.-H. Cho, "High-performance software load balancer for cloud-native architecture," *IEEE Access*, vol. 9, pp. 123 704–123 716, 2021. DOI: 10.1109/ACCESS.2021.3108801.

[27] R. D. G. Pacífico, L. F. S. Duarte, L. F. M. Vieira, B. Raghavan, J. A. M. Nacif, and M. A. M. Vieira, "eBPFlow: A Hardware/Software Platform to Seamlessly Offload Network Functions Leveraging eBPF," *IEEE/ACM Transactions on Networking*, pp. 1–14, 2023. DOI: 10.1109/TNET.2023.3318251.

[28] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014, ISSN: 0146-4833. DOI: 10.1145/2656877.2656890.

[29] R. Stoyanov, W. Armour, and N. Zilberman, "Network-accelerated cluster scheduler," in *Proceedings of the SIGCOMM '22 Poster and Demo Sessions*, ser. SIGCOMM '22, Amsterdam, Netherlands: Association for Computing Machinery, 2022, pp. 16–18, ISBN: 9781450394345. DOI: 10.1145/3546037.3546050.

[30] P. Vörös, D. Horpácsi, R. Kitlei, D. Leskó, M. Tejfel, and S. Laki, "T4P4S: A Target-independent Compiler for Protocol-independent Packet Processors," in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, 2018, pp. 1–8. DOI: 10.1109/HPSR.2018.8850752.

[31] "p4-dpdk-target: P4 driver SW for P4 DPDK target," Open Networking Foundation (ONF). (), [Online]. Available: https://github.com/p4lang/p4-dpdk-target.

[32] N. Jain, V. K. C. Mohan, A. Singhai, D. Chatterjee, and D. Daly, "Kubernetes Load-Balancing and Related Network Functions Using P4," in *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '21, Layfette, IN, USA: Association for Computing Machinery, 2022, pp. 133–135, ISBN: 9781450391689. DOI: 10.1145/3493425.3502768.

[33] A. Belkhiri, M. Pepin, M. Bly, and M. Dagenais, "Performance analysis of dpdk-based applications through tracing," *Journal of Parallel and Distributed Computing*, vol. 173, pp. 1–19, 2023, ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2022.10.012.

[34] M. Chiosi, D. Clarke, P. Willis, A. Reid, J. Feger, M. Bugenhagen, W. Khan, M. Fargano, C. Cui, H. Deng, J. Benitez, U. Michel, H. Damker, K. Ogaki, T. Matsuzaki, M. Fukui, K. Shimano, D. Delisle, Q. Loudier, C. Kolias, I. Guardini, E. Demaria, A. M. Roberto Minerva, D. López, F. J. R. Salguero, F. Ruhl, and P. Sen, "Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges & Call for Action," Darmstadt, Germany, Tech. Rep., Oct. 2012. [Online]. Available: https://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[35] R. Alfredsson, "Multipath Transport Protocol Offloading," M.S. thesis, Karlstad University, Department of Mathematics and Computer Science, 2022, p. 61.

[36] T. Geier and S. Rieger, "Improving the deployment of multi-tenant containerized network function acceleration," in *2022 5th International Conference on Advanced Communication Technologies and Networking (CommNet)*, 2022, pp. 1–7. DOI: 10.1109/CommNet56067.2022.9993962.

[37] "Infrastructure Programmer Development Kit (IPDK)," Intel. (), [Online]. Available: https://ipdk.io/.

[38] "K8s-infra-offload: Kubernetes infrastructure offload recipe," Intel. (), [Online]. Available: https://github.com/ipdk-io/k8s-infra-offload.

[39] "Intel© Intelligent Fabric Processors," Intel. (), [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html.

[40] A. AlSabeh, E. Kfoury, J. Crichigno, and E. Bou-Harb, "Leveraging SONiC Functionalities in Disaggregated Network Switches," in *2020 43rd International Conference on Telecommunications and Signal Processing (TSP)*, 2020, pp. 457–460. DOI: 10.1109/TSP49548.2020.9163508.

[41] D. Kim, J. Nelson, D. R. K. Ports, V. Sekar, and S. Seshan, "Redplane: Enabling fault-tolerant stateful in-switch applications," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ser. SIGCOMM '21, Virtual Event, USA: Association for Computing Machinery, 2021, pp. 223–244, ISBN: 9781450383837. DOI: 10.1145/3452296.3472905.

[42] Z. Zha, A. Wang, Y. Guo, Q. Li, K. Sun, and S. Chen, "EZPath: Expediting Container Network Traffic via Programmable Switches," in *2022 IFIP Networking Conference (IFIP Networking)*, 2022, pp. 1–8. DOI: 10.23919/IFIPNetworking55013.2022.9829818.

[43] M. M. Iordache-Sica, T. Kraiser, and O. Komolafe, “Seamless hardware-accelerated kubernetes networking,” in *Proceedings of the 2nd ACM SIGCOMM Workshop on Future of Internet Routing & Addressing*, ser. FIRA ’23, New York, NY, USA: Association for Computing Machinery, 2023, pp. 23–28, ISBN: 9798400702761. DOI: 10.1145/3607504.3609292.

[44] D. Scano, A. Giorgetti, F. Paolucci, A. Sgambelluri, J. Chammanara, J. Rothman, M. Al-Bado, E. Marx, S. Ahearne, and F. Cugini, “Enabling p4 network telemetry in edge micro data centers with kubernetes orchestration,” *IEEE Access*, vol. 11, pp. 22 637–22 653, 2023. DOI: 10.1109/ACCESS.2023.3249105.

[45] “Intel© Infrastructure Processing Unit (Intel© IPU),” Intel. (), [Online]. Available: https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html.

[46] A. Moore and J. Henrys, “IPU Based Cloud Infrastructure: The Fulcrum for Digital Business,” Tech. Rep., 2023. [Online]. Available: https://www.intel.com/content/www/us/en/products/docs/programmable/ipu-based-cloud-infrastructure-white-paper.html.

[47] R. Lal, J. B. Anderson, and A. Jackson, “Data processing unit’s entry into confidential computing,” in *Proceedings of the 12th International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP ’23, New York, NY, USA: Association for Computing Machinery, 2023, pp. 56–63, ISBN: 9798400716232. DOI: 10.1145/3623652.3623670.

[48] T. Groves, D. Hazen, G. Lockwood, and N. J. Wright, “Use it or lose it: Cheap compute everywhere,” in *Driving Scientific and Engineering Discoveries Through the Integration of Experiment, Big Data, and Modeling and Simulation*, J. Nichols, A. ‘B. Maccabe, J. Nutaro, S. Pophale, P. Devineni, T. Ahearn, and B. Verastegui, Eds., Cham: Springer International Publishing, 2022, pp. 280–298, ISBN: 978-3-030-96498-6.

[49] P. Kummrow. “Intel© IPU E2000: A collaborative achievement with Google Cloud,” Intel. (Jan. 2023), [Online]. Available: https://medium.com/intel-tech/intel-ipu-e2000-a-collaborative-achievement-with-google-cloud-eb1dda8c0177.

[50] N. Mehta. “The next wave of Google Cloud infrastructure innovation: New C3 VM and Hyperdisk,” Google. (Oct. 2022), [Online]. Available: https://cloud.google.com/blog/products/compute/introducing-c3-machines-with-googles-custom-intel-ipu.

[51] J. Kato, M. Sonoda, O. Shiraki, S. Gokita, and M. Hamaminato, “OVS Full Offload on Kubernetes for Low Latency,” Fujitsu Limited, Tech. Rep. 3, Jul. 2021.

[52] Y. Qiu, Q. Kang, M. Liu, and A. Chen, “Clara: Performance clarity for smartnic offloading,” in *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, ser. HotNets ’20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 16–22, ISBN: 9781450381451. DOI: 10.1145/3422604.3425929.

[53] B. Sukhwani, M. Kapur, A. Ohmacht, L. Schour, M. Ohmacht, C. Ward, C. Haymes, and S. Asaad, “Janus: An experimental reconfigurable smartnic with p4 programmability and sdn isolation,” in *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA ’23, New York, NY, USA: Association for Computing Machinery, 2023, p. 230, ISBN: 9781450394178. DOI: 10.1145/3543622.3573158.

[54] L. Linguaglossa, S. Lange, S. Pontarelli, G. Rétvári, D. Rossi, T. Zinner, R. Bifulco, M. Jarschel, and G. Bianchi, “Survey of performance acceleration techniques for network function virtualization,” *Proceedings of the IEEE*, vol. 107, no. 4, pp. 746–764, 2019. DOI: 10.1109/JPROC.2019.2896848.

[55] T. Nagendra and R. Hemavathy, “Unlocking kubernetes networking efficiency: Exploring data processing units for offloading and enhancing container network interfaces,” in *2023 4th IEEE Global Conference for Advancement in Technology (GCAT)*, 2023, pp. 1–7. DOI: 10.1109/GCAT59970.2023.10353542.

[56] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, “NetFPGA SUME: Toward 100 Gbps as Research Commodity,” *IEEE Micro*, vol. 34, no. 5, pp. 32–41, 2014. DOI: 10.1109/MM.2014.61.

[57] S. Qi, H.-S. Tsai, Y.-S. Liu, K. K. Ramakrishnan, and J.-C. Chen, “X-io: A high-performance unified i/o interface using lock-free shared memory processing,” in *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*, 2023, pp. 107–115. DOI: 10.1109/NetSoft57336.2023.10175428.

[58] S. Qi, L. Monis, Z. Zeng, I.-c. Wang, and K. K. Ramakrishnan, “Spright: Extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing,” in *Proceedings of the ACM SIGCOMM 2022 Conference*, ser. SIGCOMM ’22, Amsterdam, Netherlands: Association for Computing Machinery, 2022, pp. 780–794, ISBN: 9781450394208. DOI: 10.1145/3544216.3544259.

[59] T. Graf. “Cilium service mesh – everything you need to know,” Isovalent. (Jul. 2022), [Online]. Available: https://isovalent.com/blog/post/cilium-service-mesh/.

[60] *BPF Features by Linux Kernel Version*, BCC-Docs Authors. [Online]. Available: https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md.

[61] *eBPF-Docs*, eBPF-Docs Authors. [Online]. Available: https://ebpf-docs.dylanreimerink.nl/linux.

[62] *Aya – eBPF library for the Rust programming language*, The Aya Contributors. [Online]. Available: https://aya-rs.dev/.

[63] A. Khichane, I. Fajjari, N. Aitsaadi, and M. Gueroui, “Cloud Native 5G: an Efficient Orchestration of Cloud Native 5G System,” in *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, 2022, pp. 1–9. DOI: 10.1109/NOMS54207.2022.9789856.