UNIVERSIDADE FEDERAL DO PAMPA

Diogo Mainart Monteiro

SmartINCO: A SmartNIC-based framework for In-Network Container Orchestration

Alegrete
2025

Diogo Mainart Monteiro

SmartINCO: A SmartNIC-based framework for In-Network Container Orchestration

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Marcelo Caggiani Luizelli

Coorientador: Prof. B.Sc. Victor Hugo Schneider Lopes

Alegrete
2025

Diogo Mainart Monteiro

SMARTINCO: A SMARTNIC-BASED FRAMEWORK FOR IN-NETWORK CONTAINER ORCHESTRATION

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação

Monografia defendida e aprovada em: 12 de dezembro de 2025.

Banca examinadora:

_____

Marcelo Caggiani Luizelli

Orientador

Unipampa

_____

Victor Hugo Schneider Lopes

Coorientador

Unipampa

_____

Francisco Vogt

Unicamp

Ariel Goes de Castro

Unicamp

Assinado eletronicamente por **MARCELO CAGGIANI LUIZELLI**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 16/12/2025, às 22:16, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **VICTOR HUGO SCHNEIDER LOPES**, **Aluno**, em 16/12/2025, às 23:02, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **Francisco Germano Vogt**, **Usuário Externo**, em 17/12/2025, às 04:31, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **Ariel Góes de Castro**, **Usuário Externo**, em 17/12/2025, às 09:51, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

A autenticidade deste documento pode ser conferida no site https://sei.unipampa.edu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **1919463** e o código CRC **7B6AF814**.

This work is dedicated to my grandfather,
Diogo Rolf Mainart.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my family, especially my parents, for their constant daily support and efforts to make me happy – something for which I am truly grateful. I am also deeply thankful to Dr. Marcelo Luizelli, who has provided me invaluable academic opportunities and served as a great mentor, inspiring me to become a better researcher and person. In extension, I would like to thank Dr. Fabio Rossi, who inspires me since my early years in the technical course. I am especially grateful to my partner, Gabriela, for her kindness and unwavering support throughout this period. Finally, I thank my research colleagues and friends – Victor Hugo, Pedro Henrique, Pedro, Luis Fernando, Gabriel, Thiago and Ariel – for their collaboration and friendship.

Nobody starts anything on Friday.
- Luizelli, Marcelo (2025)

RESUMO

A orquestração de contêineres tradicional, normalmente executada em um controlador centralizado, costuma depender de plataformas como Kubernetes para gerenciar recursos por meio da alocação de requisições e da escalabilidade de aplicações conteinerizadas. Essas ferramentas fornecem a disponibilidade e a segurança esperadas em ambientes de nuvem e são convenientes devido aos ciclos rápidos de desenvolvimento e manutenção. Contudo, orquestradores convencionais possuem limitações de desempenho sob cargas de rede elevadas: maior uso de recursos, monitoramento de baixa granularidade e tempos de reação lentos no loop de controle. Por outro lado, com o aumento das demandas computacionais, SmartNICs surgem como um elemento decisivo para otimização de desempenho. Dessa forma, operações como criptografia, compressão, e inspeção de pacotes podem ser aceleradas por hardware, permitindo o processamento de milhões de pacotes por segundo. Além disso, possuem Data Processing Units (DPUs) dedicadas que oferecem uma solução economicamente eficiente por meio de arquiteturas baseadas em Acorn RISC Machine (ARM) ou Application-Specific Integrated Circuit (ASIC), adequadas para descarregar tarefas dos Central Processing Units (CPUs) de servidores. Essa tecnologia viabiliza modelos híbridos de execução, nos quais parte do processamento é alocada para SmartNIC visando maior eficiência energética. Para superar as limitações da orquestração tradicional e gerenciar sistemas conteinerizados de forma eficiente, este trabalho apresenta uma Prova-de-Conceito que desloca a lógica de orquestração para dispositivos de rede especializados. O SmartINCO é o primeiro framework completo – com agendamento e escalonamento parametrizados – de Orquestração de Contêineres *In-Network* baseado em SmartNICs. Ele realiza o *offloading* das decisões de *scheduling* e *autoscaling* para a plataforma Mellanox BlueField-2. Demonstramos que o SmartINCO opera a 25 Million packets per second (Mpps) utilizando oito instâncias conteinerizadas de uma aplicação simples de encaminhamento baseada em Data Plane Development Kit (DPDK), superando soluções existentes de Orquestração In-Network. Também avaliamos o comportamento de três algoritmos tradicionais de escalonamento (Round-Robin, Best-Fit e Worst-Fit) nesse contexto, mostrando que ambientes centrados na rede exigem estratégias e políticas de alocação de recursos distintas.

Palavras-chave: Orquestração de Contêineres em Rede, Problemas de Agendamento, SmartNIC.

ABSTRACT

Traditional container orchestration, typically deployed on a centralized controller, often relies on stacks such as Kubernetes to manage resources by allocating requests and scaling containerized applications. These tools provide the expected availability and security for cloud environments and are convenient due to rapid development and maintenance cycles. However, conventional orchestrators face performance limitations under high network-load scenarios: increased resource usage, coarse-grained monitoring, and slow control-loop reactions. As computational demands increase, SmartNICs have emerged as a game-changer for performance optimization. That way, packet operations such as cryptography, compression, and inspection can be hardware-accelerated, allowing the processing of millions of packets per second. Also, the dedicated Data Processing Units (DPUs) offer a cost-efficient solution through Acorn RISC Machine (ARM) or Application-Specific Integrated Circuit (ASIC)-based architectures, making them suitable for offloading tasks from host Central Processing Units (CPUs). This technology has enabled hybrid execution models, where processing is partially offloaded to Data Processing Units (DPUs) to improve power efficiency. To address the limitations of traditional orchestration and efficiently manage containerized systems, this work presents a Proof-of-Concept that offloads orchestration logic to specialized network devices. SmartINCO is the first SmartNIC-based In-Network Container Orchestration framework that provides both scheduling and autoscaling parametrization. It offloads orchestration decision-making to the Mellanox BlueField-2 platform. We demonstrate that SmartINCO operates at 25 Million packets per second (Mpps) using eight container instances of a simple forwarding application using Data Plane Development Kit (DPDK), surpassing existing attempts at In-Network Orchestration. We also assess the behavior of three traditional scheduling algorithms (Round-Robin, Best-Fit, and Worst-Fit) in this context, showing that network-centric environments require distinct strategies and resource allocation policies.

Key-words: In-Network Container Orchestration, Scheduling Problem, SmartNIC.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

5G Fifth Generation of Mobile Network.

6G Sixth Generation of Mobile Network.

API Application Programming Interface.

ARM Acorn RISC Machine.

ASIC Application-Specific Integrated Circuit.

B Bytes.

BF Best-Fit.

BF2 BlueField-2.

CNA Cloud-Native Application.

CPU Central Processing Unit.

DNS Domain Name System.

DOCA Data Center-Optimized Architecture.

DPDK Data Plane Development Kit.

DPU Data Processing Unit.

DUT Device Under Test.

eSwitch Embedded Switch.

FPGA Field Programmable Gate Array.

Gbps Gigabits per second.

HPA Horizontal Pod Autoscaler.

Mpps Million packets per second.

NFV Network Functions Virtualization.

NIC Network Interface Card.

OS Operating System.

P4 Programming Protocol-independent Packet Processors.

PDP Programmable Data Plane.

QoS Quality of Service.

RR Round-Robin.

RSS Receive Side Scaling.

SDN Software-Defined Networking.

SF Scalable Function.

SmartNIC Smart Network Interface Card.

SoC System-on-Chip.

SON Self-Organizing Network.

TCP Transmission Control Protocol.

VF Virtual Function.

VNF Virtual Network Function.

VPA Vertical Pod Autoscaler.

WF Worst-Fit.

CONTENTS

1 INTRODUCTION

This chapter discusses the foundations of In-Network Container Orchestration, highlighting the limitations of container orchestrators on granularity and resource consumption. Furthermore, we briefly introduce the problem and the purpose of the current work.

1.1 Context and Motivation

With the deployment of Fifth Generation of Mobile Network (5G) and the upcoming Sixth Generation of Mobile Network (6G), modern applications such as Large Language Models require higher network performance than earlier generations. Consequently, they must handle high data traffic and intensive processing within short periods of time. An Ericsson report (Ericsson, 2024b) shows a continuous increase in the use of 5G, and current projections indicate that it will represent nearly 80% of global mobile data traffic by the end of the decade. This evolution requires a fundamental shift in how software and hardware are designed and monitored. The 5G white paper (NGMN Alliance, 2015) already demanded a modular solution that integrates Network Functions Virtualization (NFV) and Software-Defined Networking (SDN). The upcoming next generation of wireless telecommunications, 6G, proposes, through emerging hardware, to provide energy efficiency and sustainability solutions without compromising on performance (Ericsson, 2024a).

However, realizing the full potential of 5G – and preparing for 6G – demands processing capabilities that traditional architectures cannot always provide. To bridge this gap, research efforts have focused on process offloading to SDN devices known as Smart Network Interface Cards (SmartNICs). These devices are the evolution of ordinary network interface cards that commonly just perform simple processing and forwarding tasks. They provide a dedicated architecture that operates on hardware-time efficiency. Modern System-on-Chip (SoC)-based SmartNICs, such as the Mellanox Bluefield 3, support network speed up to 400 Gigabits per second (Gbps) not exceeding 150 watts (NVIDIA, 2025). SoC devices alongside with Data Plane Development Kit (DPDK) provide high efficiency and lower development complexity compared to other architectures. Other SmartNIC designs offer higher performance. However, they lack additional development and deployment capabilities. SoC-based architectures are normally operated using a high-level programming language which simplifies the development process while delivering a high energy efficiency (TIBBETTS; IBTISUM; PURI, 2025).

Large-scale network applications (e.g. Virtual Reality, Time-Sensitive Applications, Cloud Gaming) require high throughput in a minimal time window (ELBAMBY et al., 2018). They also operate in the microsecond-scale, which traditional monolithic designs cannot adequately handle in terms of scalability and flexibility (HOLMA; TOSKALA; NAKAMURA, 2020). The dynamic of these networks, where traffic patterns can

change within milliseconds, necessitates software architectures that can adapt in real-time to varying network conditions. Furthermore, the distributed and ultra-low latency requirements of edge computing applications demand architectural patterns that can efficiently utilize dispersed resources while maintaining Quality of Service (QoS). Even robust infrastructures, such as a cloud environment, which claim to provide computational power and storage at almost full availability (e.g. 99.999%), deal with a lack of flexibility and security when built over centralized designs (PASUPULATI; SHROPSHIRE, 2016).

This technological evolution creates a compelling need for software designs that can match the agility and performance characteristics of next-generation networks. To handle these emerging demands, the industry started to adopt new cloud-native architectures that provide more flexible, modular, and faster solutions to cloud environments. These software architectures are founded on the partitioning of processing into multiple task-specific components (i.e., microservices), each responsible for delivering responses within a short time frame. Therefore, instead of following a straight pipeline, Cloud-Native Applications (CNAs) distribute the workload among several lightweight processing boxes (i.e. containers). When the demand for a specific task increases, a controller allocates additional containers only for that specific task, instead of the entire workflow. This scaling behavior maintains Service Level Agreement compliance while minimizing unnecessary infrastructure costs, as demonstrated in (MAO; HUMPHREY, 2011). Driven by similar motivations, several network applications utilize this design: Open5GS, the core of 5G technology (Open5GS, 2017); OpenRAN (O-RAN Alliance, 2018); and NFVs applications (e.g., Vector Packet Processor) are among the many existing examples. CNAs are built in their majority to support the following design patterns: microservice, containerization and container orchestration.

Container orchestrators, such as Kubernetes, are generally composed of multiple distributed systems responsible for container management. These stacks provide a powerful environment for the development and maintenance of applications, commonly reducing development and maintenance costs. Kubernetes' robust availability, fault tolerance, and management features provide the persistence required to support these applications. Additionally, they provide a flexible environment through parametrization and extensions, which offer customized solutions for distinct purposes. However, besides their great flexibility and convenience, these orchestrators provide limited operation granularity which in some cases – most specially on time-sensitive applications – can decrease system efficiency. A study found that a network application experiences 4× more overhead when run on Kubernetes compared to Docker Swarm (BELTRE et al., 2019). They suggest this is due to Kubernetes full network stack virtualization. Furthermore, the limited metric granularity of standard approaches, forces the Kubernetes ecosystem to make scaling decisions in the scale of seconds. Usually, Kubernetes metrics collectors operate at second-level granularity (commonly around 15 seconds by default). However, this interval is configurable, and

alternative monitoring tools (e.g., Prometheus) can collect metrics at higher frequencies. Still, even with reduced intervals, the standard Kubernetes metrics pipeline is not designed for microsecond-level network measurements, which require specialized low-latency observability tools. Additionally, with 6G on the horizon, traditional CNA orchestration is becoming insufficient to perform time-sensitive use cases. Thus, even with an optimal and efficient scheduling solution, Kubernetes operates at the software-level, which leads to an increased resource usage, affecting the entire management lifecycle. With that in mind, this work proposes an initial version of the SmartINCO framework which is a first step towards In-Network Container Orchestration – explored in detail in Section 2.5.

1.2 Problem

Several studies have focused on offloading control of scheduling decisions to SmartNICs in various distributed system contexts. Maestro (PEREIRA; RAMOS; PEDROSA, 2024) builds hash maps that associate flow characteristics with specific Central Processing Unit (CPU) cores. RingLeader (LIN et al., 2023) introduces a more sophisticated approach by using a load balancer that assigns flows to cores based on scheduling behavior. Horus (YASSINI et al., 2024), in contrast, leverages Application-Specific Integrated Circuit (ASIC) architecture to map specific flows to racks. However, none of these approaches evaluate their effectiveness in a containerized environment.

In-ReAl (VOGT et al., 2024) is the only work that focuses on container orchestration offloading. It introduces a framework that offloads scaling decisions to SmartNICs using network metrics. This approach allows for more immediate detection of increased demand, thereby enabling faster scaling decisions. However, this solution is restricted to a Virtual Network Function (VNF) environment and lacks scheduler strategies. Therefore, there is a need to explore and evaluate existing scheduling algorithms in the context of container orchestration offloading to SmartNICs. Additionally, there is also a need to compare the performance of SmartNIC-based and conventional approaches.

1.3 Goals and Contributions

To address these limitations, this work proposes the framework **SmartINCO: a SmartNIC-based In-Network Container Orchestrator** for network applications. The solution exploits the effectiveness of BlueField-2 (BF2) hardware features and the energy-efficient Data Processing Unit (DPU) by offloading both scheduler and autoscaling decision-making to SmartNICs. The decisions, based on the VNF workload allocation, are performed in fractions of seconds. The solution, built on Docker Engine and Docker API, deploys fewer abstraction layers than traditional container orchestrators, which can boost the performance. Furthermore, we evaluate well-known scheduling heuristics – Round-Robin, Best-Fit, and Worst-Fit – in distinct network scenarios.

The contributions of this work can be summarized as follows:

- **Fully offloading container orchestration to SmartNIC;**
- **Provide SmartINCO, a low-overhead framework for SmartNIC-based container orchestration;**
- **Compare and evaluate the effectiveness of existing scheduling algorithms for container orchestration.**

1.4 Outline

The remainder of this work is organized as follows. Chapter 2 reviews related work in the context of In-Network Container Orchestration, with a focus on CNAs and the SmartNICs, specially SoC-based and their advantages. Chapter 3 presents the SmartINCO design at the host and SmartNIC level. Chapter 4 discusses the evaluation methodology and results obtained. Finally, in Chapter 5, we make final remarks and propose future work.

2 BACKGROUND AND RELATED WORK

This chapter presents the foundations and recent studies concerning container orchestration and SmartNIC. We begin by introducing their architectural and operational aspects, and then emphasize the gap in the literature regarding in-network container orchestration.

2.1 Cloud-Native Applications

In the beginning of cloud computing, cloud providers first offered Infrastructure as a Service using Virtual Machines. This model served simple applications, such as web applications or basic Application Programming Interfaces (APIs), quite well. However, the infrastructure was rigid and lacked supporting cloud services and a toolkit, which made maintenance difficult. To address these challenges, cloud platforms evolved over the years by adding new services and operational features.

As the systems have grown, the lack of agility, scalability, and resilience forced the industry to adopt new architectural paradigms that could better exploit the flexibility of cloud infrastructure. These paradigms laid the foundation for what is now known as cloud-native computing which center around microservices, containerization, and orchestration (GANNON; BARGA; SUNDARESAN, 2017). Below are definitions of these core concepts:

- **Microservice:** A software design architecture that fragments monolithic applications into multiple task-specialized services. The decoupled design of this architecture provides a convenient development and maintenance environment. Each microservice performs a specific task (commonly a short task) rapidly. Commonly, they intercommunicate through an internal network, indicating their current state or requesting external processing.

- **Containerization:** A lightweight virtualization technique that isolates applications and their dependencies (e.g., libraries, kernel resources) within containers by sharing the host operating system kernel. In contrast to traditional virtualization, containerization does not require a hypervisor layer, resulting in reduced overhead and faster startup times.

- **Container orchestration:** The automated management of containerized applications, including their deployment, scaling, networking, and lifecycle operations. Orchestration platforms such as Kubernetes coordinate multiple containers across clusters, ensuring high availability, fault tolerance, and efficient resource utilization.

The modularity of microservices and the scalability of containers provide elastic resource utilization that adjusts according to demand. Therefore, when the demand

for a specific task increases, the orchestrator allocates containers only for the specific microservice instead of the entire workflow. This allows for maintaining the QoS without incurring unnecessary costs of infrastructure.

2.2 Container orchestration

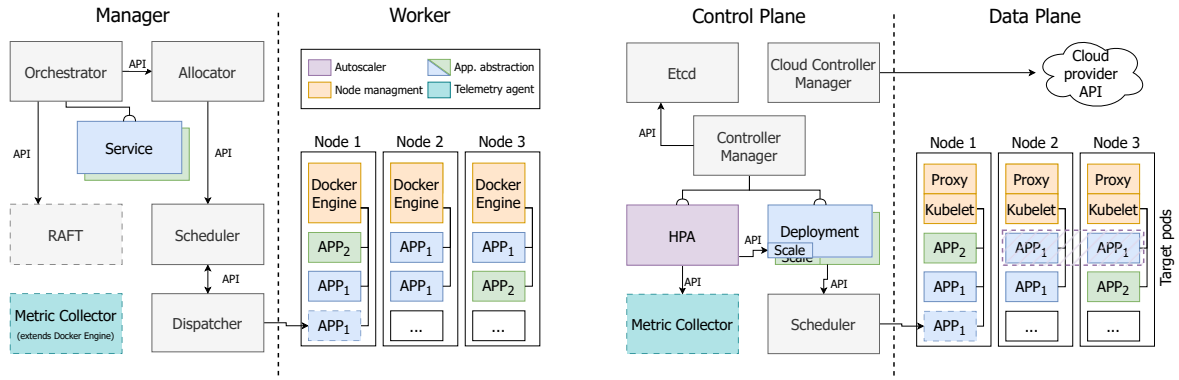
This section discusses the architectural and operational aspects of container orchestration platforms, focusing on Kubernetes and Docker Swarm as representative examples. Both platforms have gained significant traction in the industry due to their robust features and community support. We also discuss other solutions, such as Apache Mesos[1], and HashiCorp Nomad[2]. This section is based on official documentation and publicly available information provided by the Kubernetes and Docker Swarm projects, as published on their official websites (Kubernetes, 2024) (Docker, 2025).

2.2.1 Components of Orchestration

The first step in this process is the container engine, which provides the necessary tools to create, run, and manage containers on a single host. Popular container engines include Docker Engine and Podman. This layer offers immediate benefits such as packaging consistency, resource isolation, and rapid deployment. However, running containers directly on a single host presents significant limitations when transitioning to production environments, as it lacks native solutions for failure recovery, load distribution, and automated scaling across multiple servers. To overcome these challenges, container orchestration emerged as a critical management layer that provides the necessary intelligence to automate deployment, scaling, networking, and high availability for containerized workloads. While all container orchestration platforms share this common mission, they are not uniform; they differ fundamentally in their underlying architectural approaches and design. To illustrate these architectural differences, Figure 1 presents the control and data plane organization of Kubernetes and Docker Swarm, highlighting how each orchestrator manages container scheduling and cluster coordination.

In Docker Swarm, the orchestration and management of containers are handled by manager nodes and worker nodes, and their interaction is shown in Figure 1a. The manager nodes are responsible for maintaining the cluster state, orchestrating services, and handling user commands through the Swarm API. Managers use the Raft consensus algorithm to replicate and synchronize the cluster internal key-value store, which tracks configurations, services, and task assignments. The scheduler within the manager nodes decides where tasks should run, considering resource constraints and placement preferences. Managers also handle service scaling, rolling updates, and leader election to ensure

[1] `https://mesos.apache.org/`
[2] `https://hashicorp.com/en/products/nomad`



(a) Docker Swarm architecture (b) Kubernetes architecture

Figure 1 – Container orchestrators architecture

Source: Adapted from (Kubernetes, 2024) and (DOCKER, 2024)

fault tolerance and continuity. The worker nodes execute the tasks assigned by managers. Each worker runs a lightweight agent that communicates with the current leader manager, reporting node status and receiving task instructions. Workers host and manage the containers that implement the services defined in the Swarm configuration. The workload distribution across worker nodes can adapt dynamically according to resource availability and defined scaling policies. Despite its well-designed architecture, there are some cases where Docker Swarm may not be the best fit. For instance, it lacks some advanced features found in other orchestrators, such as Kubernetes, including extensive support for custom resource definitions and a more mature ecosystem for monitoring and logging. Additionally, Docker Swarm networking model is less flexible compared to Kubernetes extensive networking options, which can be a limitation for complex applications requiring sophisticated network configurations.

Kubernetes represents its functioning by separating into a Control Plane and Data Plane (as shown in Figure 1b). The Control Plane is responsible for the operational side of Kubernetes. The Data Plane, on the other hand, is where user applications run. The Control Plane handles pod orchestration and maintenance; it contains a set of task-specific and distributed systems that manage pod behavior. The API Server exposes the Control Plane services, creating a bridge between services and components. The Etcd stores API Server data in a key-value format including metrics, current and desired state. The Scheduler identifies and assigns unbound pods to nodes. The Controller Manager is a daemon process responsible for coordinating control loops – components that regulate the state of the system. Finally, the Cloud Controller Manager acts as a bridge between cloud providers' APIs, allowing private, public, and hybrid cloud processing. The Data Plane, on the other hand, is the user side, responsible for pods and deployments. This component contains the workers that process the requests (applications). The disposition can vary according to the demands and available resources. The Data Plane is com-

monly distributed among private and cloud infrastructures, while the Control Plane is centralized.

Other container orchestrators work similarly to Kubernetes and Docker Swarm, but they have their own architectural particularities. For instance, Apache Mesos is a framework-based cluster manager that handles distributed systems coordination, which fundamentally involves abstracting resources as a single pool for application provisioning. In this design, the master node is responsible for managing the cluster state and resource allocation. The master node communicates with agent nodes, which are responsible for running tasks and reporting resource availability. Frameworks like Marathon[3] and Chronos[4] operate on top of Mesos to provide container orchestration functionalities, including scheduling, scaling, and service discovery.

HashiCorp Nomad is a flexible and lightweight orchestrator that supports containerized and non-containerized workloads. It employs a client-server architecture where server nodes manage the cluster state and scheduling decisions, while client nodes execute the assigned tasks. Nomad's scheduler uses a bin-packing algorithm to optimize resource utilization across the cluster. It supports various workload types, including Docker containers, virtual machines, and standalone applications, making it versatile for different deployment scenarios.

Regardless of the architectural differences, all container orchestrators share common functionalities such as scheduling, autoscaling, service discovery, and load balancing. These features are essential for managing containerized applications in dynamic and distributed environments.

2.2.2 Autoscaling

Autoscaling is the feature of adjusting the number of instances of an application based on demand. This action aims primarily to increase energy efficiency by optimizing compute resources, thereby reducing infrastructure costs. The standard Kubernetes autoscaling approach consists of using HPA. Every certain period of time[a] HPA obtains resource usage metrics (i.e., CPU and memory usage) and computes the desired number of pods (*desiredReplicas*). The *desiredReplicas*, as shown in Equation 2.1, evaluates the ratio between the current metric value and the desired metric value of the pod, multiplied by the number of instances. Therefore, *desiredReplicas* indicates scaling every time that the current metric value is different from the desired value. For instance, if the desired metric is 100m and the current value is 200m, the number of pods must be doubled. Commonly, metrics involve the average value of computational resources (i.e., CPU and memory) usage from the targeted pods and are commonly obtained from aggregated APIs (*metrics.k8s.io*, *custom.metrics.k8s.io*, or *external.metrics.k8s.io*).

[3] `https://github.com/d2iq-archive/marathon`
[4] `https://github.com/mesos/chronos`

$$desiredReplicas = ceil\left[currentReplicas \cdot \frac{currentMetricValue}{desiredMetricValue}\right] \tag{2.1}$$

Table 1 – Horizontal Pod Autoscaler flags.

Flag	Description	Default
[a]horizontal-pod-autoscaler-sync-period	Time between evaluations	15s
[b]horizontal-pod-autoscaler-initial-readiness-delay	Delay for pods to become ready for HPA	30s
[c]horizontal-pod-autoscaler-cpu-initialization-period	Wait before CPU considered for new pod	5min
[d]horizontal-pod-autoscaler-downscale-stabilization	Look-back for downscale decisions	5m
[e]horizontal-pod-autoscaler-tolerance	Metric divergence before HPA scales	0.1

Source: Kubernetes (2025)

HPA, same as Kubernetes, is a very complex tool, with robust parameters and many extensions. It allows setting a waiting time before metrics are considered in scaling decisions, both before[b] and after[c] the pod is set as *Ready* status. Additionally, it waits a certain time window to scale down[d] to avoid fluctuations. Also, when the recommendation changes remain within a certain tolerance[e], the number of pods is maintained. It also supports multiple metrics, which is achieved by evaluating the desired number of pods for each metric and taking the most conservative value. This technique generally involves exporting new metrics to a Prometheus[5] server. Table 1 presents the default values of Kubernetes configuration flags. The interval between metric checks and `desiredReplicas` recommendations aligns with the metrics collection period, which is 15 seconds by default.

There are some scenarios where increasing the number of instances is not the best approach. For instance, when resource demand is irregular or bursty, creating new instances may lead to underutilization and increased costs. In such cases, scaling the resources of existing containers can be more efficient. This is where Vertical Pod Autoscaler (VPA) comes into play. This operation consists of changing the capacity of container resources instead of increasing the number of instances. Therefore, when a pod reaches the limit of resource usage, instead of creating a new instance, the orchestrator increases the resource limit of the container. Kubernetes, Nomad and Mesos support VPA as an alternative or complementary approach to HPA.

Unlike Kubernetes, Docker Swarm does not support automatic scaling. Other container orchestrators use a similar approach to Kubernetes, but with simpler parameters or external plugins. Nomad uses a plugin-based architecture that allows the definition of custom autoscaling strategies. It supports multiple metrics sources, including Prometheus and Datadog[6]. Apache Mesos, on the other hand, relies on external tools like Chronos and Marathon to implement autoscaling functionalities. These tools monitor resource usage

[5] `https://prometheus.io/`
[6] `https://datadoghq.com/`

and adjust the number of instances based on predefined policies. Some cloud providers such as Google GKE[7] and Amazon EKS[8], managed Kubernetes services, offer built-in autoscaling features similar to standard Kubernetes but with additional integrations to their respective cloud monitoring services (i.e. Google Cloud Monitoring[9] and Amazon CloudWatch[10]).

2.2.3 Scheduling

Initially we described scheduling as the allocation of instances to nodes. However, in the remainder of this work, we focus on request-level resource management. Therefore, from this point forward, the term *scheduling* will refer to the allocation of incoming requests to target containers (similar to proxying), reflecting the context of the subsequent sections.

The concept of scheduling dates back to early computer science, where it referred to the allocation of tasks to processing units in operating systems. In that context, scheduling algorithms were developed to optimize CPU utilization, minimize response time, and ensure fairness among processes. Traditional scheduling algorithms include Round-Robin (RR), Best-Fit (BF) and Worst-Fit (WF). RR consists of distributing requests equally among instances in a cyclic order. The idea is to avoid overloading a single instance while others remain idle. BF aims to allocate requests to the instance with the lowest resource usage, optimizing resource utilization. WF, on the other hand, allocates requests to the instance with the most available resources, aiming to balance the load across all instances.

In modern distributed systems, scheduling has evolved to manage resources across multiple nodes and services, aiming efficient workload distribution and optimal resource utilization. On the context of container orchestration, scheduling plays a crucial role in managing the flow of requests to container instances, balancing load, and maintaining service quality. The core function of a scheduler is to determine which container instance should handle each incoming request based on various criteria such as resource availability, current load, and predefined policies.

Kubernetes and Docker Swarm operate by default using the RR algorithm, distributing user requests equally through the instances. Docker Swarm integrates HAProxy[11], which allows the use of other traditional algorithms such as *leastconn* and *first*. The Least connections (i.e., WF) approach relies on attributing the request to the instance with the fewest active connections. This approach aims to distribute the effort to reduce overhead. While *first* (i.e. BF) selects the first instance available, aiming to disable unused instances. Alternative scheduling approaches include: *source*, which uses a hash

[7] `https://cloud.google.com/kubernetes-engine`
[8] `https://aws.amazon.com/eks/`
[9] `https://cloud.google.com/monitoring`
[10] `https://aws.amazon.com/cloudwatch/`
[11] `https://haproxy.org/`

mechanism with the source IP address; *URI*, which is another hash-based approach that uses a resource identifier; and *hdr*, based on header fields.

Mesos and Nomad also use similar scheduling strategies. Mesos employs a two-level scheduling approach where the master node allocates resources to frameworks, which then schedule tasks on agent nodes. This allows for flexible and efficient resource management across different workloads. Nomad uses a bin-packing algorithm to optimize resource utilization by placing tasks on nodes based on their resource requirements and availability.

2.2.4 Networking

Container orchestrators provide built-in networking solutions to facilitate communication between containers, services, and external clients. Kubernetes uses a flat networking model, where each pod receives its own IP address, allowing direct communication between pods across nodes. It employs network plugins (e.g., Calico[12] and Flannel[13]) to implement various networking features such as overlay networks, network policies, and service discovery. Some plugins also support advanced networking features like Container Network Interface and Container Network Model standards.

Docker Swarm, on the other hand, uses an overlay network that allows containers to communicate across different hosts. It provides built-in load balancing and service discovery through Domain Name System (DNS)-based mechanisms. Both orchestrators support ingress controllers to manage external access to services, enabling features like Secure Sockets Layer termination and path-based routing.

Nomad and Mesos networking models are generally simpler compared to Kubernetes and Docker Swarm, focusing more on task-level networking rather than service-level abstractions. However, they can be extended with additional tools and plugins to achieve similar functionalities. Mesos uses a modular networking approach, allowing users to choose from various networking solutions based on their requirements. Mesos' DNS, for instance, provides service discovery capabilities by mapping service names to IP addresses. Nomad, meanwhile, relies on Consul for service discovery and supports various networking modes, including host networking and bridge networks. Those orchestrators can also integrate with third-party networking solutions to enhance their capabilities. However, they generally lack the extensive built-in networking features found in Kubernetes and Docker Swarm.

Kubernetes provides extensive support for integrating with specialized hardware through device plugins. These plugins allow Kubernetes to recognize and manage resources such as Graphics Processing Units, Field Programmable Gate Arrays (FPGAs), and SmartNICs. By leveraging device plugins, users can schedule workloads that require specific hardware capabilities, ensuring optimal performance for applications that benefit

[12] `https://github.com/projectcalico/calico`
[13] `https://github.com/flannel-io/flannel`

from hardware acceleration. This integration is particularly relevant for high-performance computing and machine learning workloads, where specialized hardware can significantly enhance processing efficiency.

2.3 Data Plane Programmability

In SDN infrastructures, the network is segmented into two entities: the Control Plane and the Data Plane. The Control Plane is responsible for processing and network management, while the Data Plane focuses on forwarding tasks. This separation emerged to overcome the limitations of earlier network designs, where forwarding and control logic were tightly integrated, and each device required manual configuration upon addition to the network (KREUTZ et al., 2014). Although this architecture improves network reconfigurability, it also introduces a significant communication overhead between the control and data planes, which can ultimately degrade network performance.

To overcome these constraints, the Data Plane Programmability paradigm emerged, enabling forwarding devices to make decisions without consulting the Control Plane. Programmable Data Plane (PDP) devices vary in architecture and capabilities, often trading off operational flexibility and performance. ASICs are among the fastest processors for network-specific tasks such as checksumming and cryptography. Modern devices (e.g., Intel Tofino (Intel, 2021)) use Programming Protocol-independent Packet Processors (P4), a language that supports custom header formats and processing logic independently of existing protocols (BOSSHART et al., 2014). However, their operations are limited in scope, lacking floating-point operations and having limited register actions. FPGAs, in contrast, provide higher flexibility than ASICs while being only about 3.2× slower (KUON; ROSE, 2006). FPGA functionality is defined using Hardware Description Language, which enables maximum resource utilization. Alternatively, SoC-based devices strike a balance between flexibility and hardware efficiency (LIU et al., 2021). An SoC is a self-contained microchip that integrates the essential components of a processing unit. In PDP architectures, SoCs typically include high-speed RAM and DPU or Infrastructure Processing Unit cores, thus delivering superior energy efficiency compared to traditional servers at a reduced cost (BURSTEIN, 2021).

2.4 SmartNICs

The first occurrence of the term SmartNIC dates to the 1990s, when a fundamental work evaluated the impact of increasing Network Interface Card (NIC) functionalities (PONOMAREV; GHOSE, 1998). At that time, the NICs were focused on simple packet reception and transmission. Over the years, the demand for efficient computational power has drawn the attention of both academia and industry, driving advancements in the SmartNIC domain. SmartNICs emerged as a game changer for fast packet processing.

Soon after their introduction, SmartNICs integrated features such as Transmission Control Protocol (TCP) offload engines, Receive Side Scaling (RSS), Virtual Local Area Network tagging, and Remote Direct Memory Access capabilities. This hardware-accelerated toolkit, present in modern SmartNICs, can provide processing on the order of nanoseconds. Currently, SmartNICs are synonymous with specialized and hardware-accelerated devices connected to the network and a computer that provides both Control Plane and Data Plane functionalities (DÖRING; STUBBE; HOLZINGER, 2021).

Despite the evolution of SmartNICs, state-of-the-art SoC-based models demonstrate the strength of DPU architectures in handling offloads at nanosecond-latency (KIANPISHEH; TALEB, 2022). For instance, the NVIDIA Mellanox BF2 architecture (illustrated in Figure 2) features DDR4 memory and up to eight interconnected 64-bit Acorn RISC Machine (ARM) cores. These SoCs operate independently of the packet processing pipeline, ensuring consistent and uninterrupted packet handling. The BF2, combined with the Data Center-Optimized Architecture (DOCA) toolkit, delivers both hardware efficiency and flexible task offloading. NVIDIA also provides an Operating System (OS) that runs on the SoC to manage NIC resources, offering significant flexibility for deploying applications that extend beyond traditional networking functions. Additionally, the existence of an OS enables the execution of countless applications, bridging the gap from the hardware level (e.g., DPDK-based) to the software level (C language).

Once packets arrive from the InfiniBand physical port (i.e., 100 Gbps interface), they are routed by the Embedded Switch (eSwitch). The eSwitch is a programmable virtual switch inside the NIC; it defines the internal routing and forwarding flow according to header attributes (e.g., netmask, MAC address, etc.). This flow can be addressed for processing through a Scalable Function (SF). SFs are very similar to VNFs, since they are services that perform processing on flow packets. In contrast to similar systems (e.g., Open vSwitch), eSwitch offers limited operations, which may lead to development difficulty in certain network applications (e.g., VNF).

To achieve the best performance, NVIDIA provides DOCA framework, which includes several drivers, libraries, and services that access hardware operations directly. In this work, we primarily focus on the API provided by the DOCA Flow library. DOCA Flow API provides hardware-time features named "Pipes"that compose the internal packet processing: Match, Actions, Monitor, and Forward.

- **Match/Action:** Match tables efficiently redirect network packets (based on L2/L3 headers) to actions like header manipulation, encryption/decryption, or metadata for hardware offloading.

- **Monitor:** Robust monitoring with integrated management capabilities (e.g., counters, policies, and telemetry) provides end-to-end visibility into DPU performance and security traffic.

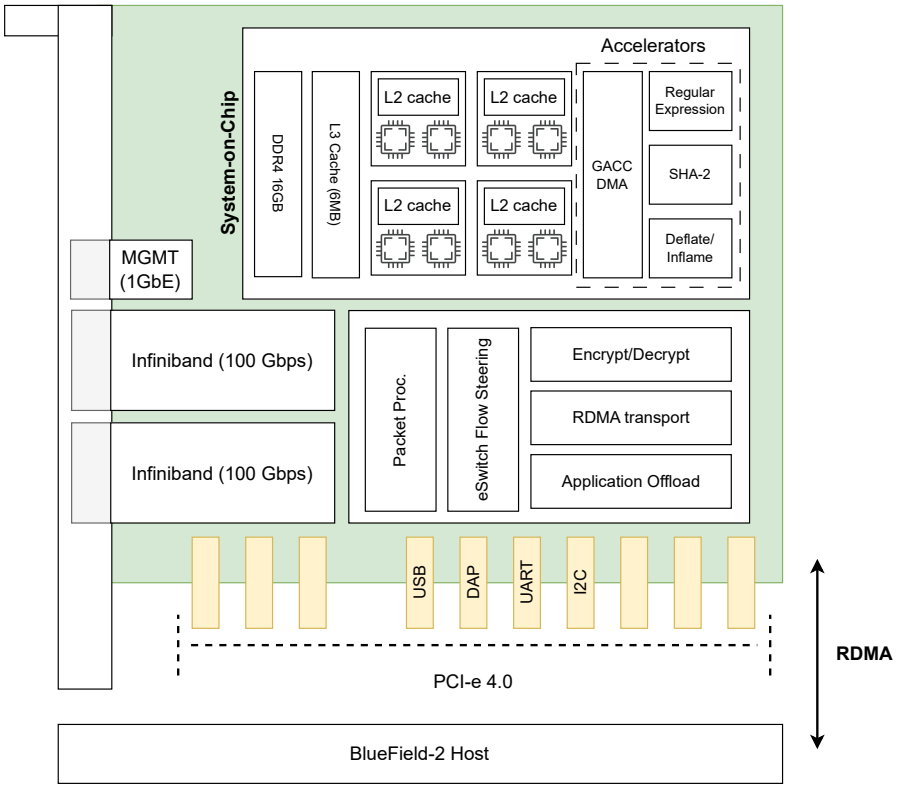


Figure 2 – BlueField-2 architecture.

Source: Adapted from (NVIDIA Corporation, 2020)

- **Forward:** Packets are routed effectively to a PCIe endpoint, a physical network port, or set of targeted ARM cores for further processing based on DPU acceleration features.

The applicability of SmartNICs goes beyond network tasks. SoC architecture provides great flexibility compared to other programmable device architectures, which makes it possible to explore hybrid solutions using both SmartNIC and server resources efficiently. Modern network applications (e.g., Open5Gs, OpenAirInterface, OpenRAN) running on CNA demand high throughput provided by SmartNICs and the resource optimization provided by container orchestration. With similar motivation, some studies propose approaches to extend system resources by offloading management tasks to SmartNICs.

2.5 In-Network Container Orchestration

Horus (YASSINI et al., 2024) creates an execution pipeline that schedules requests to a pool of workers, selecting the node for execution. It stores worker load IDs containing idle nodes. When they exist, the requests are addressed to those nodes. When no idle node is detected, Horus selects and compares two random nodes. However, this strategy requires both read and write operations on the registers, which is not possible on the

same stage in P4 pipelines. Thus, packet recirculation is necessary. Recirculation is an expensive operation in terms of throughput efficiency, as it sends the packet back to the first stage of the pipeline, causing additional latency and processing overhead. Therefore, in an attempt to mitigate this limitation, their solution only writes to the registers when a scheduling decision is affected.

Maestro (PEREIRA; RAMOS; PEDROSA, 2024) proposes a solution that automatically parallelizes VNF applications to CPU cores using RSS. RSS is a NIC strategy that handles incoming traffic distribution, reducing CPU overhead. This strategy uses hash maps that associate a flow to multiple packet headers (e.g., source MAC address, source IP address, destination IP address) and and a specific core. To pursue this, Maestro's solution analyzes the application to identify dependencies and grouping constraints according to the similarity between packets that access the same VNF state. After this, a parallelized solution is provided that claims minimal resource sharing, especially memory. However, parallelization is not a trivial task. Maintaining individual performance is challenging. Also, their solution requires a well-defined VNFs states.

RingLeader (LIN et al., 2023) operates using a specialized and scalable services architecture that runs on shared CPU resources. One of the many optimizations utilized is a demikernel-like datapath OS that overcomes I/O operations and reduces CPU overhead by fast context switching. Each service operates with multiple lightweight user tasks, named coroutines. Coroutines are scheduled and maintained by the datapath OS, facilitating multiple-core processing. The workflow consists of a pipeline of 2 components: JBSRQ (i.e., Load Balancer) selects and associates each request to a core, redirecting the flow based on the netmask. The scheduler arranges the queue according to the packet Service-Level Objective (i.e., priority), ensuring that time-sensitive requests are provided first. When a packet arrives, the Load Balancer evaluates the allocated core, considering the scheduler behavior and its rank (Equation 2.2), which considers both higher and lower priority requests in weighted scales.

$$\text{Rank}[A].\text{coreC} = \sum_{x.\text{pri} \geq A.\text{pri}} \text{Queue}[X].\text{coreC} + \lambda \sum_{x.\text{pri} < A.\text{pri}} \text{Queue}[X].\text{coreC} \tag{2.2}$$

In-ReAl (VOGT et al., 2024) is the only work that explores container orchestration offloading to SmartNICs. It proposes a scaling solution that, by monitoring the inter-packet latency and VNF metrics (i.e. CPU and memory), automatically scale resources on demand. When a certain threshold is reached, In-ReAl notifies the orchestrator to create a new container instance. However, this work does not explore scheduling techniques to distribute the requests among container instances.

Table 2 lists the key features of each solution. All use flow-based hashing to enable consistency among network functions. RingLeader makes an explicit separation between scheduling and load balancing, with optimization in request processing based

Table 2 – Comparison of In-Network Orchestration Strategies.

Characteristic	**Horus**	**Maestro**	**RingLeader**	**In-ReAl**	**SmartINCO**
Hash based on flow	✓	✓	✓	✓	✓
Scheduler			✓		✓
Autoscaler				✓	✓
SmartNIC		✓	✓	✓	✓
Containerized context				✓	✓

on packet priority. Maestro is focused on parallelism using RSS-friendly mappings, while Horus explores additional ASIC offloading, which requires pipeline design caution to avoid incurring throughput loss. These differences emphasize how each system compromises between online flexibility, performance, and hardware constraints in their unique ways. Still, none of those works evaluate their solutions in a containerized context. Only In-ReAl and SmartINCO propose container-level offloading. However, In-ReAl focuses solely on scaling without addressing scheduling. Therefore, SmartINCO is the first work to propose both scheduling and autoscaling offloaded to SmartNICs in a containerized environment.

3 SMARTINCO DESIGN

This chapter describes the functioning of the SmartINCO framework, as well as the environmental and technical aspects of our experiments. Additionally, it details the evaluation process for comparing the different scheduling approaches.

3.1 Overview

SmartINCO is a SmartNIC-based In-Network Container Orchestrator for network applications. It operates on demand, distributing incoming traffic to instances of DPDK application. The solution is also responsible for taking scaling decisions and for instructing Docker API to create or delete new instances. SmartINCO uses Docker and DPDK to better trade off efficiency and flexibility. This is because traditional container orchestrators (i.e. Kubernetes and Docker Swarm) have an additional overhead compared to Docker engine alone (PAN et al., 2019). The Docker image used exposes a VNF that is shared between the container instance (running on the host) and the SmartNIC.

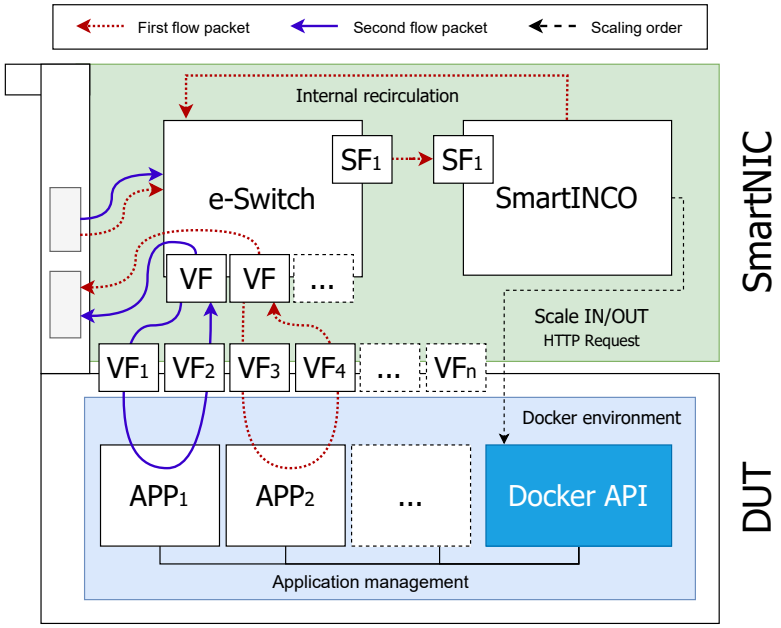


Figure 3 – SmartINCO architecture

As shown in Figure 3, SmartINCO operates directly on the SmartNIC to intercept incoming flows, assign them to container instances, and trigger scaling actions on the host through the Docker API. When a packet is received on the BF2, it is forwarded to the SmartINCO through the eSwitch. Then, the Scheduler creates an association between a flow (based on the headers) and a specific instance of the network application (represented by a VNF). The solution relies on changing eSwitch forwarding rules to send the specific flow to a specific Virtual Function (VF) after the attribution process. However, this operation alone reduces the Scheduler's control over the instances and their resource

availability, which can lead to a reduction of hardware performance or even a lack of QoS. To overcome this, an exclusive thread, denoted as Metrics Collector, is set for telemetry. It is used to monitor the flows associated with the VFs, to compute the network demand and throughput of each instance over a time period, and to provide this information to the Scheduler and Autoscaler. After the scheduling decision, a new forward rule is added to the eSwitch, sending new packets of the flow to the designated VNF. Then the DPDK application receives the packet, operates (in our case, keeping the packet unchanged), and forwarding it back through the VF. Finally, the packet is returned to the destined output physical port.

3.2 SmartNIC design

The SmartNIC component of SmartINCO is responsible for orchestrating containerized network applications. It handles incoming traffic by using the hardware acceleration available in DOCA libraries, which enables accelerated packet parsing and flow management. These features leverage hardware capabilities to offload these tasks from the host system, thereby reducing latency and improving overall performance. Furthermore, SmartINCO offloads the autoscaling logic to the SmartNIC, which focuses on collecting metrics from the eSwitch and instructing the host to create or delete container instances based on the observed workload. The SmartNIC also manages the distribution of incoming traffic to the appropriate container instances based on the selected scheduling algorithm. Therefore, to isolate the framework's functionalities, we segregate the primary features as follows:

- **Scheduler:** This component is the core of SmartINCO design. It is responsible for associating new flows to specific container instances. The decision of which instance will be used relies on the selected approach (RR, BF or WF) and the instances' availability.
- **Metrics Collector:** Runs in the background, monitoring the number of packets sent to each VNF. This telemetry component gathers eSwitch flow counters via `ovs-ofctl`. This process operates within a defined time window to estimate the workload during that interval.
- **Autoscaler:** The controller receives metrics and makes scaling decisions based on an internal heuristic. It defines a target packet rate as the application bottleneck (since VNFs are typically CPU-bound) and dynamically adjusts resources to maintain a target utilization level (50% by default).

To clarify how SmartINCO manages scheduling, telemetry, and autoscaling on the SmartNIC, Figure 4 details its internal components and their interactions with the data path and eSwitch rules.

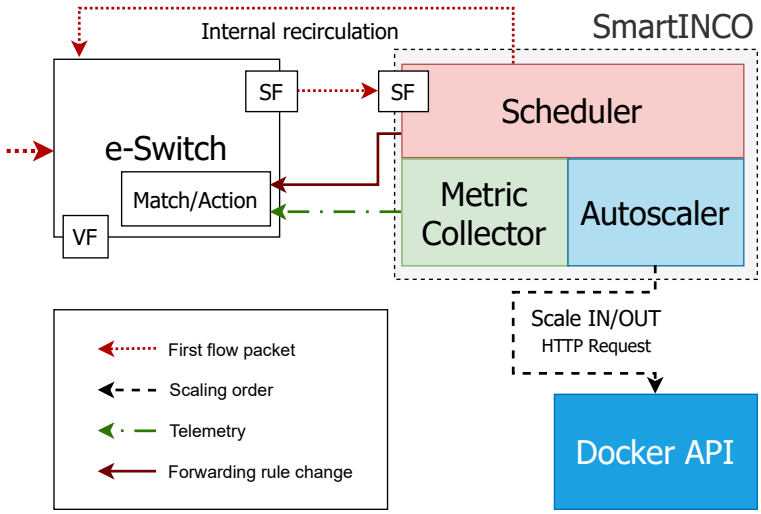


Figure 4 – SmartINCO: SmartNIC design

Our work proposes offloading and evaluating well-known scheduling algorithms to the SmartNIC to decrease the overhead of the Kubernetes environment. We selected three algorithms to evaluate in the SmartNIC environment. It is important to note that other strategies can also be evaluated or proposed during the development of this work.

- **Round-Robin:** This algorithm cycles through the available containers in order. Thus, the i-th flow is assigned to the i-th container. When the index exceeds the total number of containers, the counter resets.
- **Worst-Fit:** This algorithm assigns new flows to the least-loaded container. The goal is to evenly distribute the workload, maintaining a balanced usage percentage across all containers.
- **Best-Fit:** In contrast to WF, BF assigns new flows to the most utilized container that can still handle the load. This strategy aims to reach resource efficiency of active containers by concentrating VNF usage.

3.3 Host Design

While SmartNIC offloads container orchestration and incoming traffic coordination, the host (i.e Device Under Test (DUT)) is responsible for providing resources to run containers. Since management occurs directly in the network, the DUT only needs to execute operations by consuming resources and kernel features for container provisioning. The DUT contains a Docker environment running in user space and is responsible for creating and deleting applications instances, allocating IP addressing, managing the writable layer, and ensuring that containers are running. Docker consistently checks the

state of the containers and listens to the API waiting for scaling orders. Also, the host is responsible for storing or importing the Docker image locally.

3.4 SmartINCO Use Cases

SmartINCO provides an architecture that can be used to deploy a wide range of network applications. DPDK-based applications can be easily containerized and orchestrated using the framework, allowing for dynamic scaling based on network demand. Many of the steps required to deploy this containerized application are automated. Thus, both existing and emerging network applications can benefit from SmartINCO capabilities. To demonstrate the versatility of our solution, we highlight several use cases where the framework can be effectively applied:

- **NFV:** SmartINCO can orchestrate containerized VNF such as firewalls, load balancers, and intrusion detection systems. By offloading orchestration to the SmartNIC, it can reduce latency and improve throughput for these critical network functions.
- **SDN and Self-Organizing Network (SON):** SmartINCO can manage containerized network controllers and agents, enabling dynamic scaling based on network conditions. This is particularly useful in SON scenarios where network elements need to adapt to changing environments.
- **High-Performance Computing networking:** In high-performance computing environments, SmartINCO can orchestrate containerized applications that require low-latency and high-throughput networking. This includes applications such as distributed simulations and data analytics.
- **Edge computing:** SmartINCO can be deployed in edge environments to manage containerized applications that require real-time processing and low latency. This is particularly relevant for applications such as video streaming, augmented reality, and IoT data processing.

These are just some of the potential use cases for SmartINCO. The framework's flexibility allows it to adapt to various network applications, making it a versatile solution for modern networking challenges.

4 RESULTS

This chapter first details the deployment and testing environment. Next, we specify the traffic-generation characteristics and the SmartINCO parameters. We then evaluate SmartINCO's performance across various scheduling algorithms and scenarios. Finally, we discuss the results and compare them to existing work.

4.1 Environment

Table 3 – Software, Hardware, and Network Stack Configurations

System	Version
BlueField-2	MT42822
BF2 Processor	A72(D08) 8 Cores
BF2 Ubuntu	22.04
BF2 Kernel	5.15.0-1065-bluefield
DUT Processor	Intel(R) Xeon(R) Silver 4216
DUT Ubuntu	22.04
DUT Kernel	6.8.0-60-generic
Docker Engine	28.2.2
Docker Image	nvcr.io/nvidia/doca/doca:2.7.0-devel
Doca API	3.0.0058
TREX	3.02

Figure 5 provides an overview of the experimental testbed, showing how the traffic generator, DUT, and SmartNIC are connected through a 100 GbE link used during evaluation. It is composed of a DUT equipped with a SmartNIC. The DUT is a server using an Intel Xeon Silver 4216 CPU with 128 GiB of DDR4 RAM, running Ubuntu 22.04 with kernel version 6.8.0-60-generic and Docker engine (v28.2.2). The SmartNIC is a Mellanox BF2 (model MT42822), which includes eight Cortex-A72 cores and 16 GiB of DDR4 Random Access Memory, running Ubuntu 22.04 with kernel 5.15.0-1065-bluefield.

For the evaluation process we use a traffic generator (Cisco TREX[1]) to create network flows and evaluate the performance of the proposed solution. BF2 and the traffic generator are interconnected via a 100 GbE Ethernet interface to ensure high-speed data transfer.

[1] `https://github.com/cisco-system-traffic-generator/trex-core`

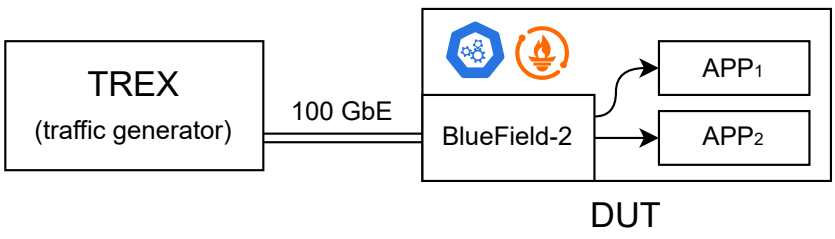


Figure 5 – Experiments architecture

4.2 Testbed Parameters and Evaluation Methodology

The traffic generator is set to produce a constant bit rate load, keeping the offered traffic stable throughout the experiments. In our experiments we use 64 B packets – unless otherwise specified – to fully stress the scheduling and autoscaling mechanisms. In all experiments, we use a constant bandwidth, resulting in a distinct number of packets for each packet size. For 64B packets, we achieved 148 Mpps; for 512B, 23 Mpps; and for 1500B packets, 8 Mpps. Following this approach, we consistently push the system to its limits by generating traffic that exceeds the combined processing capacity of all active VNF instances. This ensures we can observe how SmartINCO manages overload conditions, including expectedly high packet loss and resource contention. Furthermore, our throughput evaluations focus on packet rate (in Million packets per second (Mpps)) rather than bandwidth (in Gbps), as packet processing overhead is more relevant for containerized network functions.

The traffic generated to test SmartINCO is composed of multiple independent flows. Each flow produces the same bit rate, which together form the total offered load. A flow is defined by unique five-tuple parameters (source IP, destination IP, source port, destination port, protocol). We configure the traffic generator to create distinct source IP addresses to simulate realistic network conditions. In our experiments, we use a total of 32 flows, adding new flows every 10 seconds until reaching the maximum. This gradual increase allows us to observe how SmartINCO responds to rising demand and how effectively it scales resources.

To evaluate our solution, we run multiple experiments varying the scheduling algorithm, offered load, and packet size. Each experiment runs for a fixed duration of 330 seconds to ensure statistical significance. During this time, SmartINCO continuously monitors performance metrics and adjusts resource allocation as needed. After the experiment concludes, we process the data to extract the mean of key performance indicators. To this end, we focus on three primary metrics:

- **Throughput**: represents the number of packets processed per second by each VNF, indicating the effective compute capacity of the containerized instance. SmartINCO's telemetry thread collects per-flow counters via a series of DOCA queries associated with the eSwitch rules.

- **Packet loss**: captures requests that the container fails to serve, typically due to overload or internal failures. This metric reveals conditions that lead to performance degradation or queue buildup. It is derived by comparing packet counters at ingress and egress of each forwarding rule using the same telemetry path.

- **Resource utilization**: accounts for the number of active instances and the number of flows mapped to them. Instance count reflects operational cost, while flow count

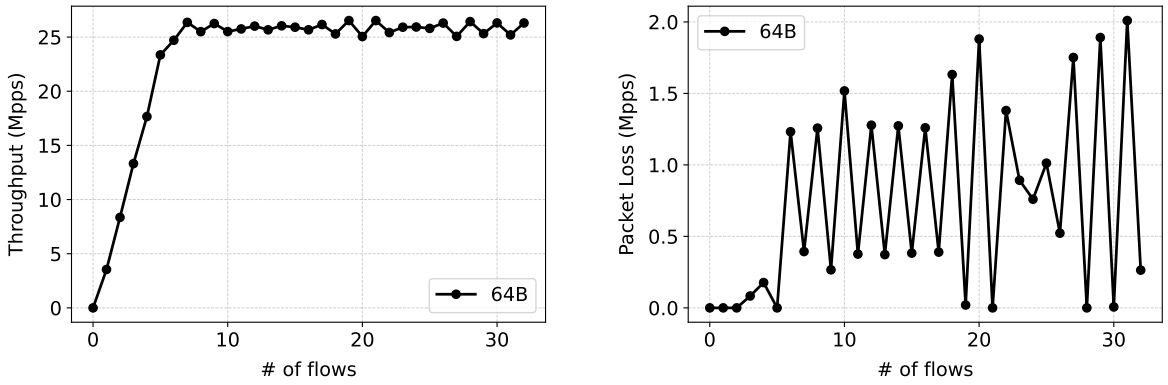
indicates the effectiveness of scheduling decisions. Both are obtained directly from the SmartNIC through eSwitch rule inspection.

The following sections present and discuss the results of each experiment in detail. First, we establish the maximum processing capacity of a single VNF, which informs the scheduling logic. Next, we analyze how each algorithm performs under increasing load, highlighting strengths and weaknesses. Finally, we explore how packet size influences throughput and resource efficiency. Our goals are to identify the optimal scheduling strategy for SmartINCO and understand its behavior across varying network conditions.

4.3 Maximum theoretical throughput per VNF

We begin by determining the maximum theoretical throughput achievable by a single VNF. Since the scheduler depends on an assumed per-instance processing limit, this value must be established before running the remaining experiments. To stress the VNF, we generate a 100 Gbps flow using 64 Bytes (B) packets (≈150 Mpps).

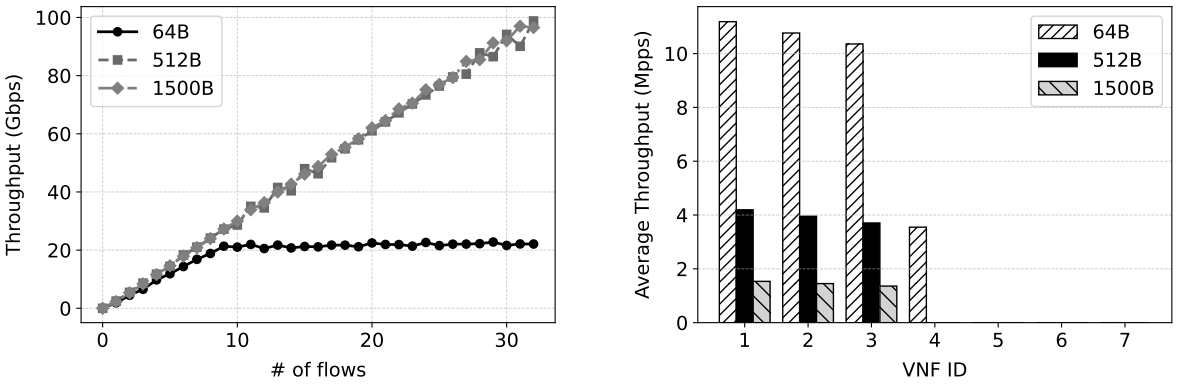
To determine the per-instance throughput limit used in all subsequent experiments, 6 reports the saturation curve of a single VNF under increasing load. After direct measurement, we observed that a single VNF sustains approximately 25 Mpps. Packet loss rises sharply at the same point, confirming saturation.



(a) Throughput over time (b) Packet loss over time

Figure 6 – Exploring maximum capacity per instance

Figure 7 extends this analysis by showing how throughput changes when multiple VNFs run concurrently, revealing shared-resource contention. In this scenario, individual VNFs process slightly above 10 Mpps, significantly below the isolated 25 Mpps. This degradation suggests contention in the underlying CNA, as co-located containers share critical system resources (e.g., memory, kernel paths), thereby limiting aggregate processing efficiency.



(a) Throughput over time (b) Throughput per VNF

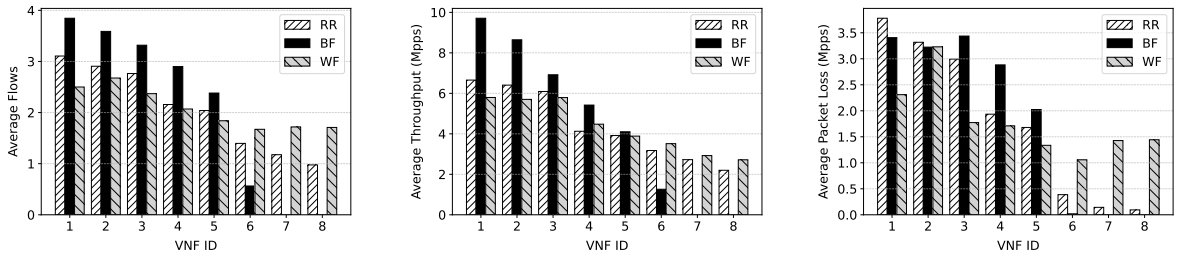
Figure 7 – Practical maximum capacity

Given this discrepancy between isolated and multi-instance behavior, we adopt a conservative limit of 10 Mpps per VNF for the remaining experiments, ensuring that scheduling and scaling decisions reliably stress the overall architecture.

4.4 Round-Robin vs Best-Fit vs Worst-Fit

To compare scheduling strategies under autoscaling, Figure 8 summarizes the flow distribution and load balance achieved by RR, BF, and WF. The experiments illustrate how each strategy behaves when combined with demand-driven autoscaling. To conduct this experiment, we generate 150 Mpps of 64 B packets.

RR distributes flows uniformly across existing instances. Because new instances are created progressively, earlier instances accumulate more traffic, leading to temporary overload and increased packet loss. Best-Fit exhibits a similar pattern but intentionally concentrates flows on the least-loaded (often earliest) instances. Despite this skew, it approaches the 10 Mpps per-instance limit. WF produces a distribution and throughput comparable to RR but achieves more balanced instance usage, as it explicitly aims to equalize load across all active VNFs.

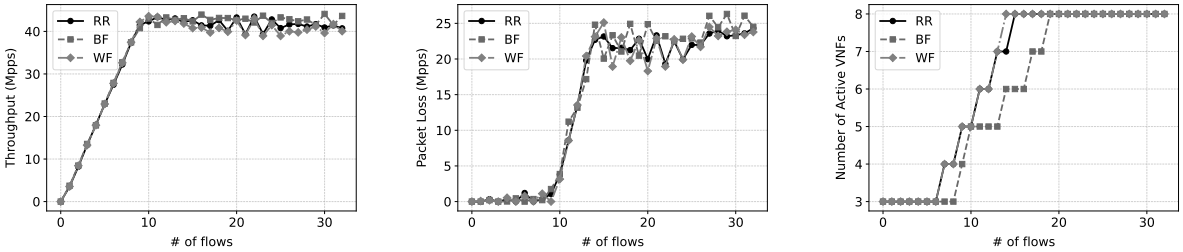


(a) Flows per VNF (b) Throughput per VNF (c) Packet loss per VNF

Figure 8 – Comparing scheduling algorithms: flow distribution

To assess which algorithm is most suitable, we compare throughput, packet loss, and VNF usage over time in Figure 9. All algorithms reach similar aggregate throughput,

stabilizing around 42 Mpps. BF performs slightly better, exceeding the others by roughly 1.5 Mpps while requiring fewer VNFs. Across all strategies, packet loss rises sharply once 9 flows are active. After saturating near 22 Mpps of dropped traffic, loss oscillates within a 5 Mpps range. Among the three, RR exhibits the most stable loss behavior.



(a) Throughput over time (b) Packet loss over time (c) VNFs over time

Figure 9 – Comparing scheduling algorithms: Performance evaluation

4.5 Exploring packet sizes: 64B, 512B, and 1500B

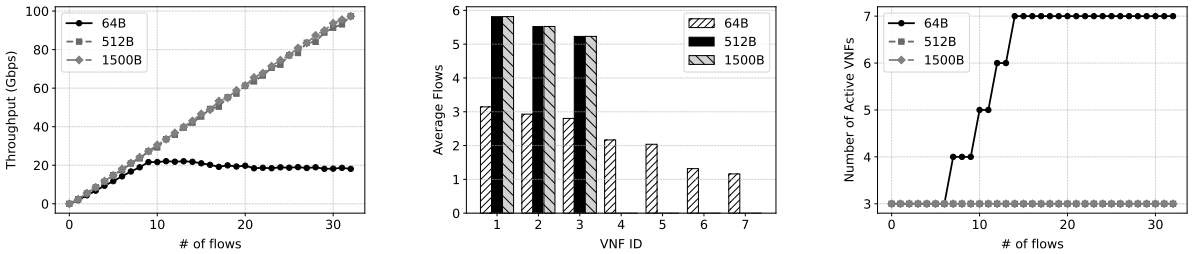
Small packets (64 B) allow significantly higher packet rates under a fixed bandwidth, making them ideal for stressing systems whose bottleneck is packet-processing capacity. They are also common in control and signaling traffic (e.g., ping, broadcast protocols).

At the other extreme, 1500 B packets represent typical CDN and video-delivery workloads. Applications such as cloud gaming, streaming, or VR require high bandwidth rather than high packet rates. Generating large volumes of 1500 B packets is challenging – commodity interfaces rarely exceed 1 Mpps – but these packets can increase latency due to higher per-packet processing cost.

Packets of 512 B represent an intermediate regime. Although less common in practice, they approximate traffic where applications fragment data into smaller chunks without reaching the full MTU. Empirical evidence shows that 512 B lies at the lower edge of a "knee" region where systems transition from overhead-bound behavior to the rapid throughput increase observed near MTU saturation (JURKIEWICZ; RZYM; BORYŁO, 2021). This location isolates the size range most sensitive to architectural effects, providing measurements that are both discriminative and theoretically meaningful.

The following experiments were evaluated using RR scheduler. The generated traffic consists on 100 Gbps equally distributed through 32 flows. Each flow starts on a 10-second interval.

Figure 10 shows that both 1500 B and 512 B packets are fully processed even when traffic reaches the line-rate limit of the interface. In both cases, SmartINCO requires only three instances to sustain maximum throughput. In contrast, 64 B traffic demands substantially more CPU cycles per bit, making full processing more challenging. As shown in Figure 10a, SmartINCO processes approximately 25 Gbps – about 49 Mpps – using eight instances.



(a) Throughput over time (b) Flows per VNF (c) VNFs over time

Figure 10 – Comparing distinct packet sizes

4.6 Discussion

These results demonstrate the effectiveness of our design: despite being a proof of concept, SmartINCO efficiently handles diverse traffic loads in a cloud-native, containerized environment.

Round-Robin and Worst-Fit exhibit similar limitations: both distribute flows across many instances, which increases resource consumption unnecessarily due to the cost of maintaining additional containers. While these strategies perform adequately in traditional CNAs, they fall short when applied to VNF workloads, where CPU cycles are the primary bottleneck. In contrast, BF consistently performs well under high traffic demand by concentrating flows to make more effective use of available resources while reducing instance churn.

Compared with similar systems such as In-ReAl, SmartINCO processes **up to 25 Gbps** of 64 B traffic using a comparable forwarding application, yielding roughly a 20% improvement on maximum throughput.

5 FINAL REMARKS

This chapter summarizes our approach and outlines the next steps required to advance SmartINCO. We begin with an overview of the obtained results, followed by a detailed description of the remaining tasks and key directions for future research.

5.1 Overview

Even with several works regarding VNF offloading to programmable devices, none of them act at the container level. RingLeader (LIN et al., 2023) offers a robust solution, however, on CPU core context (similar to Maestro (PEREIRA; RAMOS; PEDROSA, 2024)). Horus (YASSINI et al., 2024), on the other hand, performs scheduling on datacenter scale using ASIC-based programmable switches. In-ReAl (VOGT et al., 2024) provides an efficient autoscaling solution for SmartNIC in a containerized environment, but lacks an offloaded scheduling approach.

SmartINCO provides both scheduling and autoscaling offloading to SmartNICs on containerized environments. Unlike previous works limited to VNFs on core-level or datacenter-scale scheduling, we demonstrate a container-level offloading model. Containerization provides a significant flexibility to load applications, even beyond network related. The solution is based on DOCA and Docker API integration. This stack provides a powerful combination of hardware-time efficiency – provided by the SmartNIC – and low resource consumption. Our solution reduces hosts network workload, which offers more availability for new container instances.

Results demonstrate that SmartINCO achieves up to 25 Mpps per instance. After evaluating distinct packet sizes, we found that our solution can process 100 Gbps at large to medium packet sizes (i.e. 512B to 1500B). The solution scales in performance, **processing up to 40 Mpps** using multiple instances and small packet sizes. We also show the behavior of scheduling algorithms. First, we evaluate how packets are distributed over the instances, starting from 3 instances and increasing based on demand until 8 instances. We show that **Best-Fit performs slightly better than the other heuristics**. Best-Fit delivers more throughput at a lower packet loss and number of VNFs. While Round-Robin stays at the second rank.

As next steps, we will first look into other traditional and modern heuristics for scheduling and scaling decisions. The exploration of alternative strategies make it possible to achieve a higher throughput, taking the best of the available resources. Furthermore, we aim to find the optimal parametrization for both scheduling and scaling approaches to ensure availability and resource efficiency. A high scheduling threshold can lead to packet loss, while a low value underutilizes the allocated resource. The opposite occurs to scaling, when too low can lead to unnecessary resource consumption, while a high value can result in a lack of availability. Multiple containers are required on CNA context, given their failure tolerance and resource isolation. However when multiple instances are

active, their relative usage of resources to stay alive increases, which may result in a lack of performance. Additionally, we aim to expand our experiments to compare SmartINCO efficiency on different devices and architectures. Comparing results of distinct SmartNICs and applications. Also, compare the performance of SmartINCO and conventional container orchestration solutions such as Kubernetes, to find the offloading cost and improve performance. Comparing their resource consumption and performance.

As future work, we aim to integrate robust strategies (i.e. Deep Learning)to perform scheduling and scaling decisions. The power of efficient BF2 ARM core, enables second-scale inference. In that context, a model can be used to predict the future traffic or demand distribution. Then, the model can observe the past behavior of the network and provide a suggested value – or even take proactive actions – that can be used to take orchestration decisions. These solutions – based on training data – can potentially achieve near-optimal results. Besides, these algorithms can be trained to pursue a specialized task. In this scenario, they can operate separately, processing independently or in a distributed fashion. This allows multiple use-cases: a specialized pipeline-oriented topology that process packet on each network node using specialized models; a distributed topology, where scheduler operates on multiple instances or the scaling start new network nodes to process packet; or even a CNA running multiple instances on multiple nodes that are managed by a centralized orchestrator.

5.2 Challenges and Limitations

BF2 applications commonly use DOCA because of its extensive toolkit that operates on hardware timescale. These features operate on nanosecond scale, which provides a high throughput and less DPU consumption. DOCA natively supports network operations and telemetry, which can be used by SmartINCO. However for brevity our proof of concept restricts itself to using mostly DPU resources. Thus, some development limitations exist in SmartINCO due to the fact that the solution is not completely DOCA-oriented.

The limitation starts when a new flow arrives at the SmartNIC. The scheduler receive a new packet through the SF and sets a new eSwitch rule for the associated flow based on the scheduling algorithm. This process occurs on ARM core, at software level, which is significantly less effective than DOCA's hardware efficiency. On extreme scenarios, this behavior can generate packet shuffling and even packet loss. If multiple packets of a flow enter at the physical interface before the eSwitch rule being completely created, the packets may be forwarded twice to the Scheduler or to the VNF before the first packet. This limitation can occur every time a new flow arrives at the SmartNIC. The solution to this limitation relies on use the DPU without receiving packets. In this scenario, SmartINCO would only decide the next VF to forward based on the heuristics and manage container instances using the autoscaling thread. However, this approach limits SmartINCO flexibility. Some scheduling algorithms, can eventually rely on packet

attributes to decide the correct flow to allocate. This would not be possible on that DOCA-based scenario.

Furthermore, the telemetry process is also pursued on ARM core, by parsing system calls responses to obtain the number of packets in each eSwitch rule. This action is also affected by the BF2 operating system, which may lead to a lack on a precise evaluation. As future work, we aim to investigate to use DOCA API telemetry (i.e. `doca_telemetry_diag`) to collect metrics.

Also, SmartINCO uses the Docker API, that relies on TCP requests. SmartINCO currently uses DPU operations that are affected by the SmartNIC OS. This operation happens every time a container instance is created or removed by SmartINCO. We aim to investigate advanced resources to accelerate this process, ensuring packet integrity and performance.

5.3 Future Work

This section outlines enhancements that can further strengthen the capabilities, robustness, and general applicability of SmartINCO. Several dimensions of improvement are clear from the current design and experimental evaluation.

- 5G and 6G networking: SmartINCO can orchestrate containerized network functions specific to 5G and future 6G networks, such as User Plane Function and edge computing applications. This is particularly relevant as these networks demand low latency and high throughput.

- Advanced scheduling and scaling algorithms: Future work can explore Artificial Intelligence-based approaches for scheduling and scaling decisions. These algorithms can learn from historical data to optimize resource allocation dynamically.

- Integration with existing orchestration platforms: SmartINCO can be integrated with popular container orchestration platforms like Kubernetes to provide a hybrid orchestration model. This would allow users to leverage the strengths of both traditional and in-network orchestration.

- Enhanced telemetry and monitoring: Development of more sophisticated telemetry mechanisms to provide real-time insights into container performance and network conditions. This can help in making more informed scheduling and scaling decisions.

REFERENCES

BELTRE, A. M. et al. Enabling hpc workloads on cloud infrastructure using kubernetes container orchestration mechanisms. In: IEEE. **2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)**. [S.l.], 2019. p. 11–20. 24

BOSSHART, P. et al. P4: Programming protocol-independent packet processors. **ACM SIGCOMM Computer Communication Review**, ACM New York, NY, USA, v. 44, n. 3, p. 87–95, 2014. 34

BURSTEIN, I. Nvidia data center processing unit (dpu) architecture. In: IEEE. **2021 IEEE Hot Chips 33 Symposium (HCS)**. [S.l.], 2021. p. 1–20. 34

Docker. **Docker Swarm Documentation**. 2025. ⟨https://docs.docker.com/engine/swarm/⟩. Accessed: 2025-11-02. Disponível em: ⟨https://docs.docker.com/engine/swarm/⟩. 28

DOCKER, I. **Services**. 2024. ⟨https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/⟩. Accessed: 2 November 2025. 29

DÖRING, T.; STUBBE, H.; HOLZINGER, K. Smartnics: Current trends in research and industry. **Network**, v. 19, 2021. 35

ELBAMBY, M. S. et al. Toward low-latency and ultra-reliable virtual reality. **IEEE network**, IEEE, v. 32, n. 2, p. 78–84, 2018. 23

Ericsson. **Co-creating a Cyber-Physical World**. [S.l.], 2024. Accessed: 2025-06-15. Disponível em: ⟨https://www.ericsson.com/4a32a8/assets/local/reports-papers/white-papers/2024/co-creating-a-cyber-physical-world.pdf⟩. 23

Ericsson. **Ericsson Mobility Report: November 2024**. [S.l.], 2024. Accessed: 2025-06-14. Disponível em: ⟨https://www.ericsson.com/en/reports-and-papers/mobility-report⟩. 23

GANNON, D.; BARGA, R.; SUNDARESAN, N. Cloud-native applications. **IEEE Cloud Computing**, IEEE, v. 4, n. 5, p. 16–21, 2017. 27

HOLMA, H.; TOSKALA, A.; NAKAMURA, T. **5G technology: 3GPP new radio**. [S.l.]: John Wiley & Sons, 2020. 23

Intel. **P416 Intel Tofino Native Architecture — Public Version**. 2021. ⟨https://github.com/barefootnetworks/Open-Tofino⟩. Accessed: 2025-06-14. 34

JURKIEWICZ, P.; RZYM, G.; BORYŁO, P. Flow length and size distributions in campus internet traffic. **Computer Communications**, Elsevier, v. 167, p. 15–30, 2021. 47

KIANPISHEH, S.; TALEB, T. A survey on in-network computing: Programmable data plane and technology specific applications. **IEEE Communications Surveys & Tutorials**, IEEE, v. 25, n. 1, p. 701–761, 2022. 35

KREUTZ, D. et al. Software-defined networking: A comprehensive survey. **Proceedings of the IEEE**, Ieee, v. 103, n. 1, p. 14–76, 2014. 34

Kubernetes. **Kubernetes Documentation**. 2024. ⟨https://kubernetes.io/docs/⟩. Accessed: 2025-06-14. Disponível em: ⟨https://kubernetes.io/docs/⟩. 28, 29

Kubernetes. **Horizontal Pod Autoscaling**. 2025. ⟨https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/⟩. Accessed: 2025-05-07. 31

KUON, I.; ROSE, J. Measuring the gap between fpgas and asics. In: **Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays**. [S.l.: s.n.], 2006. p. 21–30. 34

LIN, J. et al. {RingLeader}: efficiently offloading {Intra-Server} orchestration to {NICs}. In: **20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)**. [S.l.: s.n.], 2023. p. 1293–1308. 25, 37, 49

LIU, J. et al. Performance characteristics of the bluefield-2 smartnic. **arXiv preprint arXiv:2105.06619**, 2021. 34

MAO, M.; HUMPHREY, M. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In: **Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis**. [S.l.: s.n.], 2011. p. 1–12. 24

NGMN Alliance. **NGMN 5G White Paper: A Deliverable by the NGMN Alliance**. [S.l.], 2015. Version 1.0; Published February 2015. Disponível em: ⟨https://www.ngmn.org/wp-content/uploads/NGMN_5G_White_Paper_V1_0.pdf⟩. 23

NVIDIA. **NVIDIA BlueField-3 DPU Controller User Manual**. [S.l.], 2025. Accessed: November 2025. Disponível em: ⟨https://docs.nvidia.com/networking/display/nvidia-bluefield-3-dpu-controller-user-manual.pdf⟩. 23

NVIDIA Corporation. **NVIDIA Mellanox BlueField-2 DPU: Product Brief**. [S.l.], 2020. Accessed: 2025-06-15. Disponível em: ⟨https://network.nvidia.com/files/doc-2020/pb-bluefield-2-dpu.pdf⟩. 36

Open5GS. **Open5GS: The Open Source 5G Core Network**. 2017. ⟨https://open5gs.org/⟩. Accessed: 2025-06-14. 24

O-RAN Alliance. **O-RAN: Towards an Open and Smart RAN**. [S.l.], 2018. October 2018. 24

PAN, Y. et al. A performance comparison of cloud-based container orchestration tools. In: IEEE. **2019 IEEE International Conference on Big Knowledge (ICBK)**. [S.l.], 2019. p. 191–198. 39

PASUPULATI, R. P.; SHROPSHIRE, J. Analysis of centralized and decentralized cloud architectures. In: IEEE. **SoutheastCon 2016**. [S.l.], 2016. p. 1–7. 24

PEREIRA, F.; RAMOS, F. M.; PEDROSA, L. Automatic parallelization of software network functions. In: **21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)**. [S.l.: s.n.], 2024. p. 1531–1550. 25, 37, 49

PONOMAREV, D. V.; GHOSE, K. A comparative study of some network subsystem organizations. In: IEEE. **Proceedings. Fifth International Conference on High Performance Computing (Cat. No. 98EX238)**. [S.l.], 1998. p. 436–443. 34

TIBBETTS, N.; IBTISUM, S.; PURI, S. A survey on heterogeneous computing using smartnics and emerging data processing units. **Future Generation Computer Systems**, Elsevier, p. 108207, 2025. 23

VOGT, F. G. et al. Poster: Towards in-network resource scaling of vnfs. In: **Proceedings of the 20th International Conference on emerging Networking EXperiments and Technologies**. [S.l.: s.n.], 2024. p. 29–30. 25, 37, 49

YASSINI, P. et al. Horus: Granular {in-network} task scheduler for cloud datacenters. In: **21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)**. [S.l.: s.n.], 2024. p. 1–22. 25, 36, 49