Information and Software Technology 197 (2026) 108180

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Choreography-defined networks: Concepts and a case study on AI-based attack detection☆

Saverio Giallorenzo [a,b], Jacopo Mauro [c], Andrea Melis [a,*], Fabrizio Montesi [c], Marco Peressotti [c], Marco Prandini [a]

[a] Università di Bologna, Italy
[b] INRIA, France
[c] University of Southern Denmark, Denmark

ARTICLE INFO

Keywords:
Software-defined networks
Virtual network functions
Choreographic programming
Network security
Denial-of-service

ABSTRACT

Modern network infrastructures increasingly rely on Software-Defined Networking (SDN) and Network Function Virtualisation (NFV) to achieve flexibility, scalability, and efficiency. While these paradigms facilitate the deployment of Cloud-native Network Functions (CNF), they lack tools for high-level programming and guarantees on correct multi-component compositions.

We introduce Choreography-Defined Networking (CDN), a methodology that applies choreographic programming to the specification and implementation of SDN compositions. In CDN, developers write a single global choreography that describes interactions among CNFs and a compiler generates endpoint code that coordinate them as specified in the choreography. CDN delivers correctness-by-construction guarantees – including deadlock freedom and communication-type safety – while eliminating the need for a centralised orchestrator, replaced by direct, parallel communication among CNFs.

To evaluate our methodology, we use CDN to design and implement a case study on a distributed, AI-enhanced SDN composition for volumetric attack detection and mitigation, in which four CNFs collaboratively analyse traffic using volumetric anomaly inspection, machine-learning classification, and signature matching. We compare this CDN implementation against two SDN baselines: a classical controller-driven chain and a hybrid solution that repurposes network traffic as a management channel.

Experiments across four representative attack scenarios show that the CDN approach reduces mean decision latency by approximately 15% over both baselines, while generating up to 80% less management traffic. These results confirm that CDN allows to raise the abstraction level at which one writes distributed SDN compositions without compromising – actually improving – runtime performance in real-world network deployments.

1. Introduction

Software-Defined Networks (SDNs) [1] and Network Function Virtualisation (NFV) [2] have revolutionised network architectures. SDN decouples the control plane from the data plane, enabling the dynamic management and configuration of network resources through a programmable software controller. NFV complements this paradigm by replacing specialised hardware appliances (e.g., firewalls, load balancers, intrusion detection systems) with software-based implementations known as Virtual Network Functions (VNFs), which run on commodity hardware. Together, SDN and NFV enable flexible deployment, rapid service provisioning, and improved scalability compared to traditional hardware-centric networks.

Despite these advances, VNFs are typically composed using an orchestrated, sequential service-chaining model inspired by service-oriented computing. This design choice mainly stems from resource constraints found in early SDN infrastructures, which limited the feasibility of running multiple VNFs in parallel — even though studies show that many enterprise network functions could logically operate concurrently [3]. Moreover, distributed composition is inherently complex: coordinating independent network functions through message passing is a well-known challenge in concurrent and distributed systems, often leading to inconsistent behaviour, communication mismatches, and subtle implementation errors [2].

☆ This article is part of a Special issue entitled: 'AI&SoC' published in Information and Software Technology.

* Corresponding author.
E-mail addresses: saverio.giallorenzo@unibo.it (S. Giallorenzo), mauro@imada.sdu.dk (J. Mauro), a.melis@unibo.it (A. Melis), fmontesi@imada.sdu.dk (F. Montesi), peressotti@imada.sdu.dk (M. Peressotti), marco.prandini@unibo.it (M. Prandini).

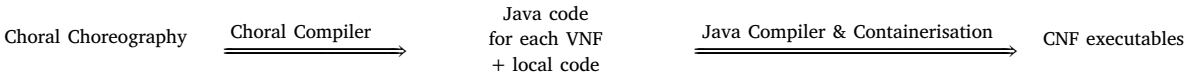
https://doi.org/10.1016/j.infsof.2026.108180
Received 4 June 2025; Received in revised form 2 March 2026; Accepted 29 April 2026
Available online 14 May 2026
0950-5849/© 2026 Published by Elsevier B.V.

Nowadays, Cloud-native Network Functions (CNFs) can replace VNFs. A CNF is a network service (typically) implemented as a containerised microservice designed to run in cloud environments and scaled automatically through tools such as Kubernetes. Indeed, unlike traditional VNFs, which often replicate monolithic appliances in virtual machines, CNFs are built as small, loosely coupled components that can be independently deployed, scaled, updated, and rolled back. CNFs leverage cloud primitives – containers and deployment orchestrators as well as service meshes and CI/CD pipelines – to automate lifecycle management and improve agility. This architectural model enables greater scalability, resource efficiency, and resilience.

Yet, CNFs still struggle to become standard practice mainly because their execution and coordination essentially constitute distributed software based on message passing, whose correct implementation is notoriously challenging even for experts [4]. It is easy, for example, to write communication actions in different programs that fail at interacting because of wrong timing or mismatches in expected payload types. Avoiding these bugs with code analysis tools is often impractical because of the state explosion problem of concurrent software [5]. Furthermore, editing the code of one VNF might break compatibility with other VNFs, so their deployment needs to be coordinated carefully. Therefore, the full potential of SDNs and VNFs deployment and management via CNFs remains untapped due to the challenge of writing and managing distributed software.

Choreography-defined networks. We address the problem of *correctly implementing distributed CNF architectures for SDN systems* by connecting the fields of SDNs and programming languages. Specifically, we take a step towards taming the complexity of developing correct CNF architectures with a development process for SDNs based on Choreographic Programming [6] (CP), a recent programming paradigm for concurrent and distributed software. CP allows developers to write a coordination plan (a 'choreography') for a set of distributed roles (abstractions of communicating processes), which are then automatically translated by a compiler into an executable program for each role. This approach greatly reduces code complexity because planned communications become syntactically manifest, expressed succinctly. Furthermore, the compilation of choreographies is backed by well-understood mathematical theories that focus on the correct matching of message send and receive actions in the generated programs. As a result, CP can guarantee important safety and liveness properties, like the absence of communication mismatches (messages have the expected type, are sent on the right channels, etc.) and deadlock-freedom [7].

To reify the approach we propose, we use the most advanced choreographic language to date: Choral [8]. Practically, Choral extends the Java language with locality information about data. In Choral, `T@A` denotes data of type `T` located at a participant, also called *role*, `A`. Given a collection of located data, we can move any of these values from one role to another with methods that take data at a role and return it at a second one. Following this abstraction, we propose to model a CNF as a role and have multiple CNFs participate in a choreography to implement a desired distributed behaviour. Our main contribution is a prototype software development method for SDNs based on Choral, called *Choreography-Defined Network* (CDN), which we schematically represent as follows.



In CDN, developers write a choreography that collectively defines the overall behaviour of multiple CNFs. Then, leveraging Choral's compiler, they obtain the implementation of each VNF for the target SDN as a Java program that they can link to local code (which implements the specific logic, e.g., traffic filtering), compile it to Java executables and containerise it to obtain a deployable CNF.

To showcase how choreography programming can be effectively used, we developed an AI-enhanced volumetric attack mitigation platform by expressing the entire workflow as a single choreography. In our prototype, incoming traffic is mirrored to a sequence of VNFs that perform packet sampling, feature aggregation, and AI-based anomaly detection; once an attack signature is identified, subsequent VNFs automatically apply filtering rules at the network edge. Because each VNF's behaviour is generated from the same high-level choreography, we guarantee that messages are exchanged in the correct order and that the code for message exchange does not present deadlocks or message losses (e.g., due to unmatched inbound and outbound requests). To quantify the benefits of the proposed choreographic approach, we compare our system against two baselines: a classical SDN controller-driven VNF chaining framework and a hybrid SDN/orchestrator solution. We measure the latency from attack onset to mitigation decision across a range of volumetric attack profiles.

From a qualitative standpoint, CDN is the only approach that simultaneously provides a global formal behavioural specification, automatic generation of local VNF implementations from that specification, direct inter-VNF communication without controller mediation, and the absence of a mandatory runtime orchestrator introducing a centralised bottleneck.

In contrast, the classical SDN chain forces every packet to traverse the entire VNF chain regardless of necessity, supporting forward-only inter-VNF communication, and conflating management and data-plane traffic. The hybrid solution partially alleviates mandatory chain traversal by routing VNF decisions through the controller via dedicated messages, but it introduces a single-entry-point queue whose overhead grows with the number of connected switches, making the approach inherently non-scalable and highly dependent on the specific controller version.

Quantitatively, the advantages of the CDN approach translate into noticeable gains: across four representative attack scenarios (a high-volume TCP SYN flood, a low-rate bursty DDoS, a ShellShock signature-based intrusion, and a signatureless Modbus anomaly), the choreographic approach consistently achieves the lowest decision latency, improving on both alternatives by approximately 15% on average, while generating up to 80% less management traffic.

These findings underscore that, qualitatively, choreographic programming simplifies the development of correct, complex multiparty workflows, while quantitatively delivering performance gains in real-world network security deployments.

This article extends previous work [9] where we originally introduced the concept of Choreography-Defined Networks. Besides a revision and expansion of the original presentation, we (i) provide full implementation details of the Distributed Denial of Service Attacks (DDoS) mitigation case study using Choral and our CDN methodology, (ii) describe the development of two alternative architectures – a classical SDN controller-driven function chain and a hybrid orchestrator-based solution – and (iii) showcase a comprehensive quantitative evaluation that compares decision latency and performance across all three implementations.

Structure of the article. We start, in Section 2, by providing the necessary background knowledge, and compare our approach with related work in Section 3. Then, in Section 4, we present a case study focused on mitigating DDoS that we use to showcase the practical application of CDN. In Section 5, we describe the implementation of the case study to illustrate both the ergonomics of the approach and how it naturally lends itself to translating workflow-like schemas into code artefacts that generate the implementation of the system. Following the obtained implementation, we introduce, in Section 6, alternative implementations of the case study, specifically, a traditional one, that follows the classical SDN function-chaining approach, and an ad-hoc one, where we optimise the communication flow among functions by hard-coding custom traffic flows. We use these three alternative implementations of the case study to present a quantitative comparison among the three approaches in Section 7, showing that the choreographic approach can consistently reduce decision latency. In Section 8, we discuss the advantages and challenges of the choreographic approach, followed by closing remarks and comments on future work in Section 9.

2. Background

We first introduce preliminary concepts regarding software-defined networking, monitoring with dynamic network reconfiguration, and choreographic programming.

2.1. Modern networking

Modern architectures are the product of two (r)evolutionary waves of innovation. The first wave saw the advent of layering and "softwarisation" of network functions. This movement began with the separation of duties between the control plane (managing sessions and signalling) and user plane (handling data traffic) – as, e.g., adopted in the Software-Defined Networking model, which places the burden of network programming fully on a controller that gives detailed forwarding instructions to devices via a dedicated protocol [10] – and proceeded with the introduction of Virtual Network Functions, i.e., software-based network components such as routers and security gateways.

Network Function Virtualisation, i.e., the process of replacing specialised devices with VNFs that can be deployed, e.g., on a virtual machine or a container, consistently integrates with the SDN paradigm [11]. Separating VNFs from their underlying hardware introduces various management challenges, such as mapping services to NFV networks, placing VNFs correctly to fulfil service objectives, dynamically allocating and scaling hardware resources, monitoring the location of VNF instances, and managing fault detection and recovery across the network.

To support the development of NFV components, the Linux Foundation, in cooperation with ETSI, launched an open-source reference platform called the Open Platform for Network Function Virtualisation (OPNFV)[1] in 2014, later expanded to include the Management and Orchestration (MANO) section.

The second wave saw some intelligence put back on a *programmable* data plane (PDP) by leveraging devices that can execute code, e.g., P4-enabled switches [12,13], to re-enable line-rate traffic analysis. To avoid losing the advantages of the SDN/NFV approach, the management paradigm should integrate these devices, as seen in the various approaches proposed for runtime interaction with P4 devices [14] – Real Time Pipeline Reconfiguration is the latest frontier of network programmability [15]. Clearly, choosing *how* the data plane devices should behave in different conditions is a decision that needs to be coordinated with all the higher-level network management decisions.

2.2. Choreographic Programming

Choreographic Programming [7,16–19] sinks its roots in service-oriented programming. Service-orientation distinguishes between two ways of implementing the coordination logic of services that belong to a distributed system: *orchestration* and *choreography*.

In orchestration, one service, called the *orchestrator*, coordinates the actions of the other services involved in an architecture. The orchestrator encapsulates and executes the distributed system's logic, managing all interactions among the participating services. While orchestration simplifies implementation and verification against a reference specification, it has several drawbacks. The orchestrator acts as a single control point, thus it can become a twofold bottleneck: its computational resources may reduce the efficiency at which it dictates operations to other services, and it may add latency in scenarios with network limitations, since it must mediate all data exchanges. Furthermore, it is a potential single point of failure and a highly valuable target for cyberattacks, putting system resilience at risk.

As an alternative to orchestration, choreographies distribute the logic of a multiparty system among the participants in the architecture. Like a choreographed performance, each service in a choreography plays a specific role and performs the corresponding actions, implementing its part in the architecture's overall interaction scheme. In this article, we follow an interpretation of choreographies called *choreographic programming*, whereby developers specify the actions and interactions of all the involved services as a choreographic program. Then, given a source choreography, the developers use a compiler to automatically generate the correct code of all the services that participate therein.

CP differentiates itself from neighbouring approaches, such as using choreographies as specifications or as types [20], by the fact its artefacts are written in a fairly concrete language. For instance, a choreographic language usually allows programmers to specify the distribution of values among the participants, message exchanges, and distributed branching behaviours. The hallmark characteristic of choreographic programming is that programmers cannot inadvertently introduce deadlocks in message exchanges — thanks to the fact that interactions syntactically pair the sending and reception of messages. Then, compilers that support behaviour-preserving properties can generate the code of the participants from a given choreography, guaranteeing that their combined, distributed execution faithfully follows the semantics of the source, including the absence of message deadlocks.

In this work, we apply CP to Software-Defined Networks using Choral, a state-of-the-art choreographic programming language that is fully interoperable with (and compiles to) Java. The next section provides an overview of Choral, including its key concepts and practical usage.

[1] https://www.opnfv.org/.

2.3. Choral

Choral [8] is an object-oriented choreographic programming language: in Choral classes and interfaces represent distributed data types parametric in the processes (*roles* in Choral terminology) participating in them. The syntax of Choral is based on that of Java, the only main difference is the introduction of notation for process parameters (recognisable by their `@` prefix) in types, as exemplified in the snippet below, which contains the definition of a distributed pair storing two values at different roles.

```
public class DPair@(A,B)<L@C,R@D> {
  public final L@A left; public final R@B right;
  public DPair(L@A left, R@B right) { this.left = left; this.right = right; }
}
```
Choral

The Choral class `DPair` is distributed over two processes represented by the parameters `A` and `B`. The definition is also parametrised on two data types `L` and `R`, which we parametrise on exactly one process, respectively abstracted by `C` for `L` and `D` for `R`. These type parameters are instantiated in the definitions of the fields `left` and `right` to locate them respectively at `A` (`L@A`) and `B` (`R@B`).

The Choral compiler generates a Java implementation for each process participating in a Choral type. Specifically, given the class `DPair` above, the Choral compiler produces the following two Java classes.

```
// Implementation of DPair for A
public class DPair_A<L,R> {
  public final L left;
  public DPair_A(L left) {
    this.left = left;
  } }
```
Java

```
// Implementation of DPair for B
public class DPair_B<L,R> {
  public final R right;
  public DPair_B(R right) {
    this.right = right;
  } }
```
Java

Unlike other choreographic models and programming languages, Choral does not fix a communication primitive or a middleware. Instead, communication mechanisms can be programmed directly. The choral standard library provides a framework with several kinds of channels organised around a hierarchy of interfaces that document standard capabilities (e.g., uni- or bidirectional channels). Programmers can thus rely on Choral's standard implementations or deploy their own solutions written directly in Choral or in Java. The snippet below reports the interface defined in the Choral standard library to represent a generic directed channel between two processes (abstracted by `A` and `B`) for transmitting data of a given type (abstracted by the type parameter `T`) from the first to the second process.

```
public interface DiDataChannel@(A,B)<T@C> {
  <M@C extends T@C> M@B com(M@A message);
}
```
Choral

Data transmission is performed by invoking the generic method `com`, which takes as input any value of a subtype `M` of `T` located at `A` and returns as output a value of the same type but located at `B`. For instance, in the expression `ch.<Integer>com(5@A)`, the value of the expression `5@A` – i.e., `5` evaluated at `A` – is communicated to `B` using the channel `ch`; the whole expression evaluates to the value `5` located at `B`. The interface `DiDataChannel` is compiled to the following Java interfaces.

```
// Implementation for A
interface DiDataChannel_A<T> {
  <M extends T> void com(M message);
}
```
Java

```
// Implementation for B
interface DiDataChannel_B<T> {
  <M extends T> M com();
}
```
Java

The interface implementing `A`'s part contains a method for sending a value (it takes in a value and returns void) while `B`'s part contains a method for receiving a value (it has no inputs and returns a value of the expected type).

Communications can be mixed with other expressions and statements. For instance, in `int@B x = ch.<Integer>com(5@A)`, the variable `x` is declared to hold integers located at `B` and is initialised with the value received by `B` from `A`. Compiling that assignment results in the following Java code.

```
// Implementation for A
ch.<Integer>com(5);
```
Java

```
// Implementation for B
int x = ch.<Integer>com();
```
Java

A key element of choreographic programming is implementing the *knowledge of choice*, i.e., when a choreography describes a choice between two possible branches made by a process, all affected participants must be (made) aware of the outcome of that choice, to ensure that they agree on which branch to execute. To illustrate this situation, consider the following method.

```
consumeItems(DiDataChannel@(A,B)<Item> ch, Iterator@A<Item> it,Consumer@B<Item> cons){
  if (it.hasNext()) {
    it.next() >> ch::<Item>com >> cons::accept;
    consumeItems(ch, it, cons);
  } }
```
Choral

Method `consumeItems` takes a channel from `A` to `B`, an iterator of items located at `A`, and a consumer function at `B`. In the conditional, `A` checks if the iterator can provide a new element and chooses whether to continue or not. In the first case, `A` fetches the item and sends it over the channel

ch to B, which then processes it (the pipe operator >> is syntactic sugar for chained method calls). However, this implementation is wrong, and the Choral compiler would report a compilation error: role B has no knowledge of the outcome of the choice made at A and yet it needs to decide whether to wait for an item to process or stop. This situation, known as knowledge of choice, is typically addressed in choreographic programming with *selections* i.e., dedicated primitives for communicating constants that the compiler can track to ensure that the necessary parties are made aware of the choice outcome. In Choral, selections can be realised as any method annotated with @SelectMethod that accepts and returns enumerated values, like the one defined by the following interface.

```
interface DiChannel@(A,B)<T@C> extends DiDataChannel@(A,B)<T> {
  @SelectMethod <M@C extends Enum@C<M> > M@B select(M@A message);
}
```
Choral

Using DiChannel, we can update consumeItems to ensure that B is made aware of which branch has been selected by A.

```
enum Choice@A { GO, STOP }

consumeItems(DiChannel@(A,B)<Item> ch, Iterator@A<Item> it,Consumer@B<Item> cons){
  if (it.hasNext()) {
    ch.<Choice>select(Choice@A.NEXT);
    it.next() >> ch::<Item>com >> cons::accept;
    consumeItems(ch, it, cons);
  } else {
    ch.<Choice>select(Choice@A.STOP);
  } }
```
Choral

The Choral compiler can then generate the implementations for A and B.

```
// implementation for A
consumeItems(DiChannel_A<Item> ch,
             Iterator<Item> it){
  if (it.hasNext()) {
    ch.<Choice>select(Choice.NEXT);
    ch.<Item>com( it.next() );
    consumeItems(ch, it);
  } else {
    ch.<Choice>select(STOP);
  } }
```
Java

```
// implementation for B
consumeItems(DiChannel_B<Item> ch,
             Consumer<Item> cons){
  switch(ch.<Choice>select()) {
    case NEXT -> {
      cons.accept(ch.<Item>com());
      consumeItems(ch, cons);
    }
    case STOP -> { }
  } }
```
Java

Besides 'enum selections', Choral supports 'type selections', which allow programmers to represent the outcome of a choice by exchanging values of disjoint types instead of values of an enumerated type [21]. Using type selections, we can optimise the code of consumeItems above by bundling together the information about the choice with the item transmitted in the then branch of the conditional.

The Choral standard library defines a hierarchy of channels that provide different types of interactions. In particular, the interface below, which we use in our implementation, defines a symmetric full-duplex channel with enumeration-based selections.

```
public interface SymChannel@(A,B)<T@C> extends DiChannel@(A,B)<T>, DiChannel@(B,A)<T> {
  <M@C extends T@C> M@B com(M@A message); // DiChannel@(A,B)<T>
  <M@C extends T@C> M@A com(M@B message); // DiChannel@(B,A)<T>
  <M@C extends Enum@C<M>ÂÂ> M@B select(M@A message); // DiChannel@(A,B)<T>
  <M@C extends Enum@C<M>ÂÂ> M@A select(M@B message); // DiChannel@(B,A)<T>
}
```
Choral

To support the definition of new communication mechanisms, the Choral standard library also includes a range of utilities for handling common tasks, such as serialisation to various text and binary formats.

3. Related work

Since the first release of the Open-Source MANO framework by the ETSI Foundation in 2016, the focus of most of the related research has been on enhancing the adaptability, efficiency, and security of VNF deployments in increasingly complex network environments. Contrarily, only a few works consider the communication logic between NFVs, mainly looking at standardising the identification and representation of an NFV through descriptors.

Nguyen et al. [22] introduced an AI-driven approach to VNF chain orchestration, which optimises resource allocation through predictive analysis of network demands and conditions. He et al. [23] expanded on the integration of VNFs with edge computing, proposing a decentralised orchestration model that enables more efficient data processing and reduces the strain on core network resources. This model leverages edge nodes to perform local data processing before transferring information to centralised servers, thereby enhancing the responsiveness of network services. He et al.'s approach improves the ability to automate the process of NFV deployment via resource allocation analysis. However, there is no reference to the possibility of automating the generation of the VNFs themselves through a more structured, high-level language, which is one of the main advantages of our solution.

To the best of our knowledge, the only approach for VNF definition that can be considered at a similar level of abstraction as ours is Intent-Based Networking (IBN) [24]. IBN aims to apply automation intelligence to devise network configuration plans, replacing the manual processes of initial set up and reaction to issues. Similar to the choreographic approach, IBN can abstract and define the behaviour of the network functions at a higher level; yet, in its current state, it requires specific hardware support, making its implementation dependent on dedicated hardware deployments [24].

Table 1
Comparison between CDN and related approaches.

Approach	Formal Spec.	Auto Gen.	Direct VNF Comm.	No Central Bottleneck
Classical SDN chain [23]	–	–	–	–
Hybrid SDN/Orchestrated [11]	–	–	Partial	–
AI-based NFV orchestration [22]	–	–	–	–
Edge-aware NFV [23]	–	–	Partial	Partial
Intent-Based Networking [24]	Partial	Partial	–	–
SDNShield-like frameworks [25]	–	–	–	–
CDN (this work)	✓	✓	✓	✓

Legend: Formal Spec. = global formal behavioural specification; Auto Gen. = automatic generation of local implementations; Direct VNF Comm. = direct inter-VNF communication without controller mediation; No Central Bottleneck = absence of mandatory runtime orchestrator.

Focussing on security, Hasneen and Sadique [26] surveyed the security challenges 5G must face when implementing its slicing capabilities with SDN and VNF technologies. Among the work mentioned by Hasneen and Sadique, Lakshmanan et al. [27] and Sun et al. [3] provide deployment solutions that can prevent a chain misconfiguration or vulnerability by design, but they consider the simplified scenario in which functions are not invoked in parallel.

Considering multidomain VNF deployment, Huff et al. [28] address the challenge of the management of the reliability of the network deployment in different domains (e.g., cloud providers, on-premises servers, etc.) with an architecture that can connect to the different chains in the cloud through tunnels (VPN or VXLAN) and guarantee a certain level of reliability. With the choreographic approach, the reliability level can be natively introduced in the choreographic logic in a way that is explicit, terse, and reusable.

Regarding the case study we present to validate and evaluate of our approach (cf. Sections 4 to 7), the closest related work is SDNShield [25], a network solution presented by Chen et al. based on NFV technologies that enforces comprehensive defence against potential DDoS attacks on SDN control plane. Chen et al. implement their scheme by deploying VNFs, but, differently from us, they rely on a centralised SDN controller that has to manage the flow on the chain. Hence, in their implementation, the controller is a point of failure that can reduce the reliability of the network and constitute a performance bottleneck. In addition, Chen et al. hardcode the coordination logic into the VNFs, preventing the usage of their implementation in local scenarios and difficult to adapt to other attacks. Our choreographic approach allows both more flexibility and adaptability to different scenarios, and it increases system resilience by eliminating, by construction, problems like deadlocks and races on messages.

Table 1 summarises the positioning of Choreography-Defined Networks (CDN) with respect to existing SDN/NFV approaches along four structural dimensions: (i) the presence of a global formal behavioural specification, (ii) automatic generation of local implementations, (iii) support for direct inter-VNF communication without controller mediation, and (iv) the absence of a mandatory central bottleneck at runtime.

The comparison highlights that classical SDN chaining, hybrid solutions, and AI-driven orchestration frameworks primarily focus on deployment or resource optimisation, but do not provide formal coordination models or automatic correctness-preserving code generation. Intent-Based Networking offers partial abstraction at the specification level, yet lacks strong communication guarantees and decentralised interaction. In contrast, CDN uniquely combines a global formal model with compiler-generated implementations and direct VNF communication, eliminating centralised orchestration bottlenecks while preserving correctness by construction.

4. Proposed methodology

We introduce our methodology for using choreographic programming to develop SDNs and showcase it through a case study using traffic analysis to detect volumetric network attacks. In particular, we aim to detect attacks using anomaly detection techniques, such as flow asymmetry [29], characterised by the possibility of generating many false positives depending on the anomaly threshold that is set or the efficiency of the detection engine [30]. In these scenarios, an effective strategy is to combine multiple detection engines with different sensitivities and thresholds [31] to obtain a deeper, more certain result rather than relying on a single oracle.

We consider a network topology in which traffic flows through a switch, programmed to mirror it towards virtual network functions deployed on the edge, to avoid sending it towards, e.g., a cloud computing centre. As visualised in Fig. 1, the network relies on the following four VNFs:

- `Split&Agg (SA)`. This function sends the traffic to the VNFs in charge of analysing it, possibly selecting which ones to involve, and then deciding which packets should be forwarded depending on the received responses.
- `Volumetric Anomaly Traffic Inspection (VOL)`. This function is configured with a set of anomaly detection rules[2] that try to identify the maliciousness of a specific flow. The output is a "Benign/Malicious" answer to tell whether the flow must be further analysed or it is a legit flow.
- `ML Detection Engine (ML)`. This function uses ML techniques (e.g., a neural network [32]) to inspect and detect if a flow is malicious or not, with a known level of reliability. When it detects a malicious flow, it reports the finding to the first VNF.
- `Signature Attack Detection (SIG)`. Every attack (e.g., DoS, Spoofing) can be represented with a signature (e.g., a hash of the payload). As commonly done by antivirus software, this function checks the flows against a database of signatures to find indications of known attacks. This detection mechanism is the fastest and less prone to false positives, but it cannot detect attacks that have not been previously classified.

As presented in Fig. 1, the traffic is first mirrored by SW to SA, which filters the packets to forward to all the other three VNFs. The VOL, ML, and SIG functions receive the filtered flow from SA and perform their analysis independently and in parallel. When each VNF reaches a decision and

[2] https://www.ibm.com/docs/en/qradar-on-cloud?topic=rules-anomaly-detection.

classifies the flow either as malign or benign, it independently informs the SA about the decision. Notice that the interaction could also be more complex since VOL may not be able to make a decision due to insufficient data, and, as a consequence, SA can increase the amount of traffic to collect (Increase Window Size arrow).

To further illustrate the advantage of the choreographed approach w.r.t. the orchestrated one, on the right side of Fig. 1 we represent the same workflow implemented with a classic SDN NVF chain and a centralised controller. The red dotted arrows on the left side of the Figure show two more inter-VNF communications that can happen without the mediation of the controller. In principle, these communications would be useless since the interactions do not belong to the process of attack detection. However, in this case, these messages could update SIG when a new attack is confidently detected, and its signature can be added to the database. These actions improve the performance of the system without overloading the controller. The same interactions are represented with the same red dotted arrows on the right side. Note that, differently from the left-side scheme, communications must always go through the centralised controller and NFVs cannot act independently — following the direction of the chain flow indicated by the yellow dashed arrow. As illustrated in the Figure, in the choreographic approach (left side), any NFV can independently interrupt the analysis, if appropriate (e.g., high-confidence attack detection), while in the classical approach (right side) communications need to go through the whole chain before producing a result.

Closing the scenario, the reaction to the attack can bypass the controller in the interest of timeliness (leftmost left-pointing arrows). SA can use P4Runtime[3] to instruct the switch to stop monitoring benign flows, or to implement a mitigation action (e.g., packet filtering) against a malicious flow.

5. Implementation of the case study

We apply our methodology to the scenario discussed in the previous section as a case study that fully implements it. In Section 5.1, we apply the choreographic step of the methodology by implementing the case in Choral, and in Section 5.2, we apply the network step and deploy the choreography as a system of VNFs by instantiating the required network functionalities.

5.1. Choral implementation

We illustrate the experience of programming the scenario from Section 4 using Choral by focusing on the multiparty interaction between the volumetric anomaly traffic inspection function `VOL`, the ML detection engine `MLE`, and the signature attack detection function `SIG` for updating the attack signature (the red, dot-dashed arrows within the chain workflow in Fig. 1). The interested reader can find the full code that implements the case study at https://doi.org/10.5281/zenodo.15519089.

Recalling the relevant exchanges in Fig. 1, ML and VOL send to SIG their analysed data signatures, which then SIG processes to label the flow. A possible Choral implementation of said exchange is the following.

```
Optional@SIG<DataSignature> s1=ch_ml_sig.com(ml_analyser.genSignature());
Optional@SIG<DataSignature> s2=ch_vol_sig.com(vol_analyser.genSignature());
sig_analyser.labelFlow(s1, s2);
```
Choral

In the first line, on the right of the assignment, we write that ML sends to SIG the result of the analysis of the data it previously processed, found in the object (located at ML) `ml_analyser` and obtained through the invocation of the method `genSignature`. The communication happens by passing to the method `com` of the object `ch_ml_sig` the result of `genSignature`. As mentioned in Section 2.2, `ch_ml_sig` is a symmetric channel shared between ML and SIG, which transmits the data returned by `genSignature` – an `Optional` that can contain the `DataSignature` of the attack, if any – to `SIG`. At the left of the assignment, we find the variable `s1`, local to `SIG`, where it stores the data sent from ML. Similarly, in the second line, we find that VOL sends a possible attack signature to `SIG`, which stores said data in `s2`. At the third line, SIG invokes the method `labelFlow` of its analyser (`sig_analyser`) to update its set of attack signatures.

The Choral code above is compiled into separate Java implementations for VOL, ML, and SIG, as shown below.

```
// Implementation for SIG
Optional<DataSignature> s1 = ch_ml_sig.<>com();
Optional<DataSignature> s2 = ch_vol_sig.<>com();
sig_analyser.labelFlow( s1, s2 );
```
Java

```
// Implementation for ML
ch_ml_sig.<>com( ml_analyser.genSignature() );
```
Java

```
// Implementation for VOL
ch_vol_sig.<>com( vol_analyser.genSignature() );
```
Java

We conclude our example by contrasting the distributed implementation above with the one below, which implements the same logic in the traditional orchestrated way, where SA is the orchestrator. The main takeaway is that the orchestrator needs to mediate the interactions between VOL, ML, and SIG, both imposing an unnecessary bottleneck and increasing the total number of communications (wasting time and bandwidth and exposing the system to increased risk of communication failures).

[3] https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html.

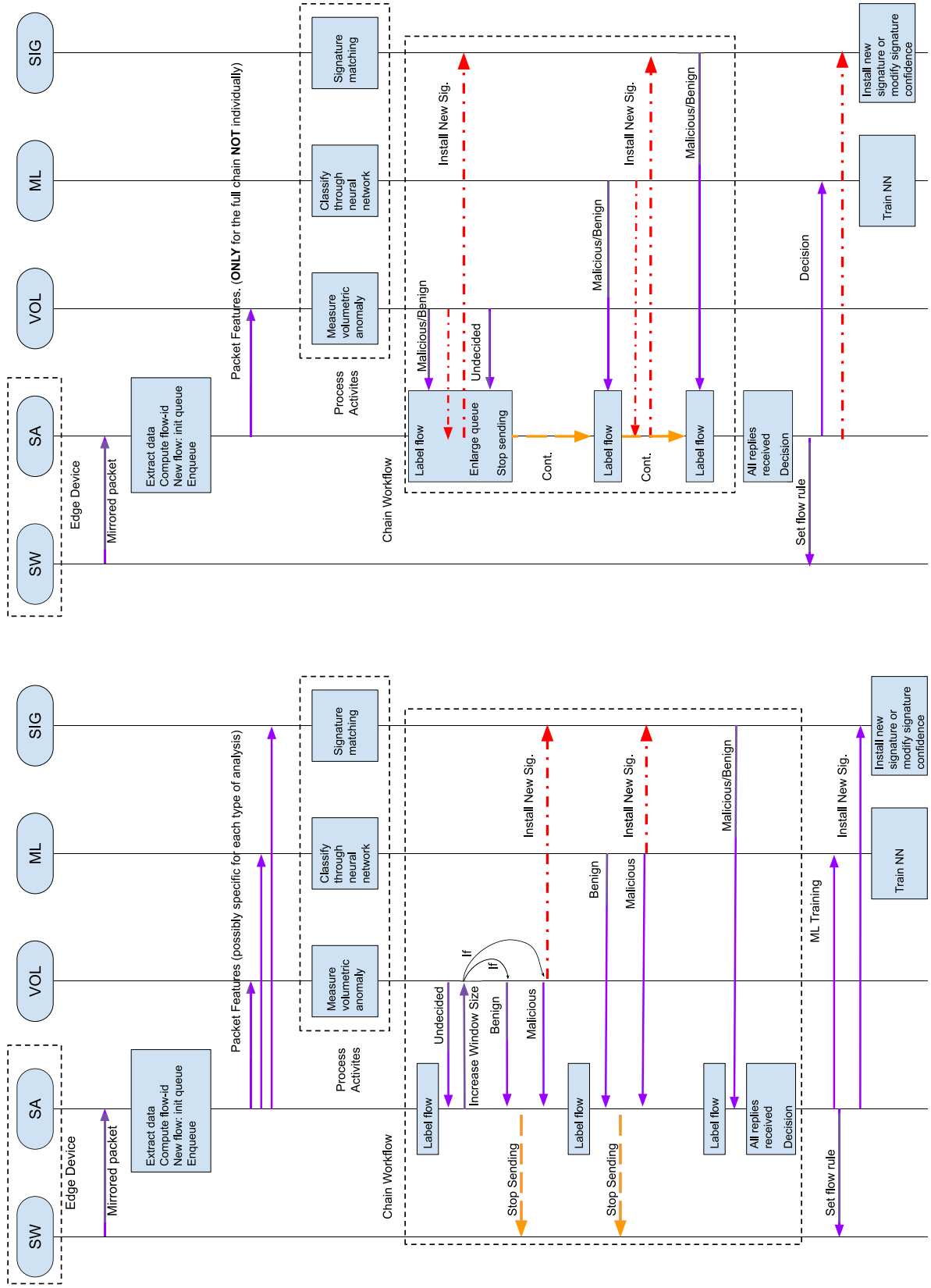


Fig. 1. A comparison of the case study workflow using the choreographic approach (left side) vs. the classic SDN orchestrated one (right side). The red dotted arrows represent a new attack signature generated by each VNF, and the yellow arrows represent the workflow possibilities (i.e., when the VNF can stop the detection process) (cf. Section 5).

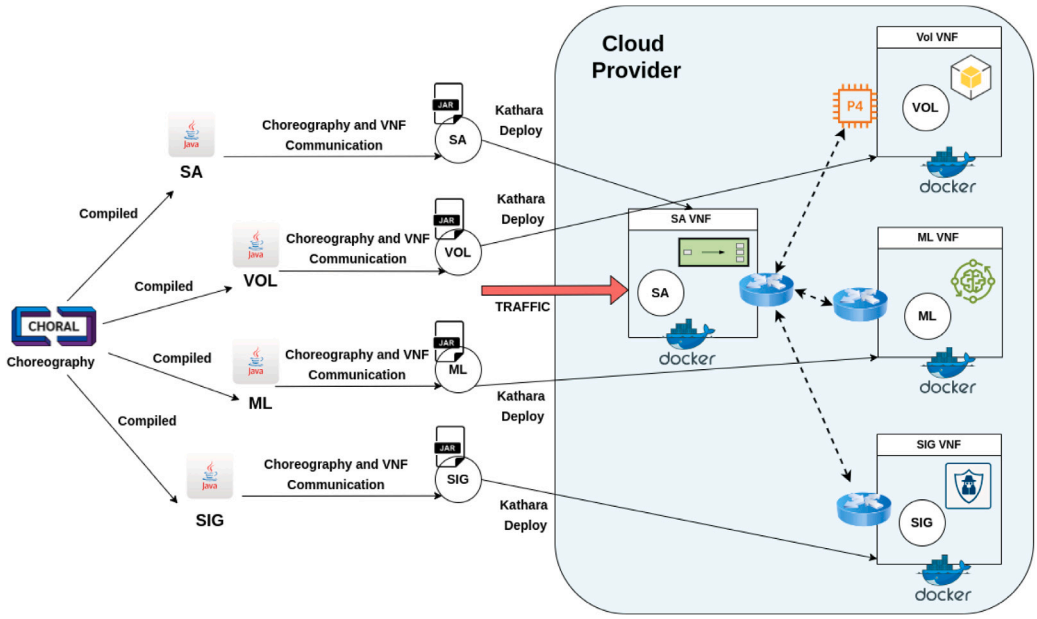


Fig. 2. The case study workflow, from the Choral code to the infrastructure deployment.

```
// orchestration at SA
Optional@SA<DataSignature> t1=ch_ml_sa.<>com(ml_analyser.genSignature());
Optional@SA<DataSignature> t2=ch_vol_sa.<>com(vol_analyser.genSignature());
Optional@SIG<DataSignature> s1=ch_sa_sig.<>com(t1);
Optional@SIG<DataSignature> s2=ch_sa_sig.<>com(t2);
sig_analyser.labelFlow(s1, s2);
```

Choral

5.2. Deployment of the network

For the deployment of the SDN, as practised in cloud network development, each VNF can be instantiated within a container. We base the infrastructure on dedicated docker Linux images, connected via a local Docker network capable of handling up to 14MB/s bandwidth. We execute all the code on a PC with Ubuntu 22.04, 16 GB RAM, and an Intel® Core™ i7 processor. For the creation and management of the infrastructure, we use Kathara,[4] an open-source container-based network emulation system for testing production networks in a sandboxed environment. We create an architecture composed of 5 containers, one for each of the 4 VNFs and one for the switch. We obtain the containers of the VNFs by creating Docker images for each VNF that includes both the executable Java files generated from the Choral implementation and the tools that implement the specific VNF behaviour. We choose a P4-capable virtual switch using the v1 model architecture for P4 and its virtualised version BMv2,[5] enabling it to execute the flow rules needed to monitor anomalies.

We implement the `VOL` VNF with a modified version of an algorithm based on Count-min Sketch [33], designed to efficiently detect the characteristic asymmetry of traffic flows exhibited by volumetric DDoS attacks.

The `ML` VNF uses a standard Random Forest Classifier implemented using the scikit-learn python library,[6] with the ability to read a process real-time traffic with the scapy library.[7] For the training set, we use a custom dataset composed of 10% of benign traffic (taken from the CIC-IDS2017 dataset [34]) and 90% of DDoS traffic (generated with the hping3[8] Linux utility).

`SIG` uses a light version of Suricata[9] threat detection software. While initially equipped with a set of default rules taken from the nuclei-discover repository [35], the Suricata database can be enriched with new signatures, either waiting a final decision identifying a new malicious flow, or even more timely if ML or VOL reach a confident early decision.

We show the deployment workflow for the described scenario in Fig. 2. The overall choreography written in Choral is projected into a set of Java files, one for each VNF. We compile these Java files and the additional, external classes they need to interact with the aforementioned tools, obtaining JAR applications that we include in dedicated containers. Kathara creates the entire infrastructure, loading the containerised applications and installing the necessary tools to execute the VNF at runtime, as well as creating the network configuration instructions for infrastructure deployment on a provider — in our case a local Docker environment managing both containers and virtual network segments.

[4] https://www.kathara.org/.
[5] https://github.com/p4lang/behavioural-model.
[6] https://scikit-learn.org/stable/.
[7] https://scapy.net/.
[8] http://wiki.hping.org/.
[9] https://suricata.io/.

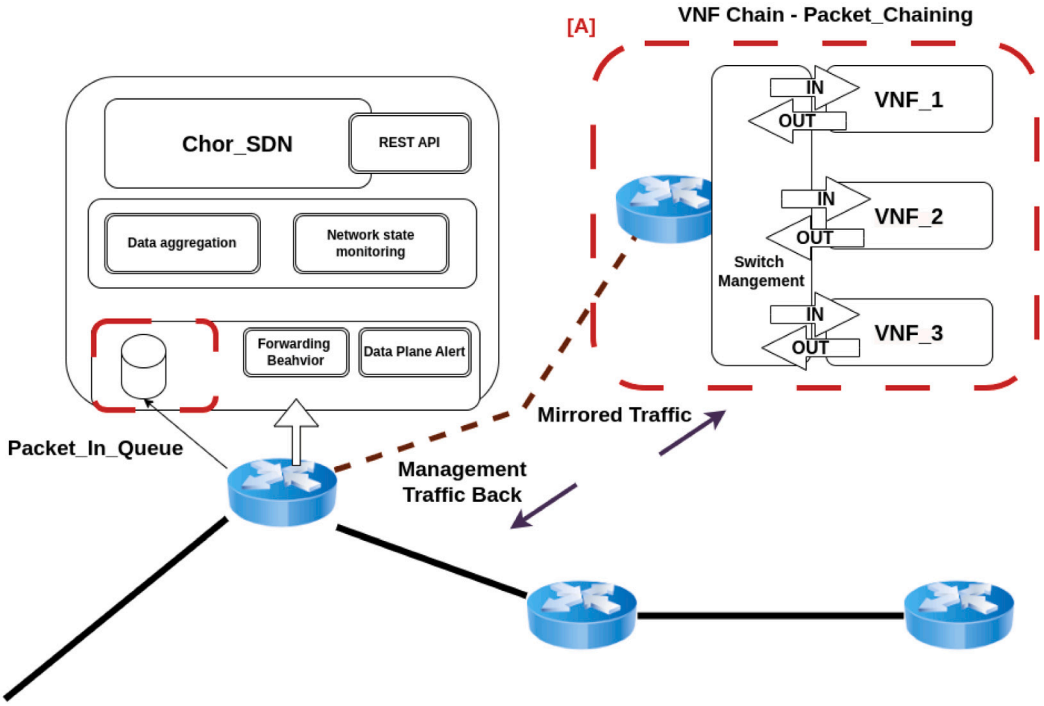


Fig. 3. The case study workflow, implemented with a classic SDN Architecture.

6. Alternatives to choreography-defined networking

Software-Defined Networking marks a distinct approach to network flow management: devices distributed on the data plane are simple actuators of policies centrally decided on the controller. Thus, the chaining of virtual functions must be achieved by letting each switch send a message to the controller whenever a new kind of flow has to be established. In the standard implementations,[10] these messages contain header-level information. To implement flexible routing decisions that involve the content of the message, the controller can be equipped with additional logic. In our scenario, the controller would be the orchestrator, which, upon deep packet inspection of a message from a VNF, can determine what is the following link in the function chain, i.e., to which VNF the message has to be delivered. In a choreographed scenario, the application-level decision is instead distributed among the VNFs and coordinated by the choreographic program.

6.1. Comparison with classic SDN approach

The classic SDN chain solution, illustrated in Fig. 3, implements a traditional version of a VNF Chain. In this implementation, traffic is mirrored from the application to the controller through specific mirroring rules on the OVS (Open vSwitch) switch. The OVS that receives the mirrored traffic forwards the traffic according to the designated ports of the first VNF and sequentially connecting the subsequent VNFs in the chain (e.g., VNF1 port out to VNF2 port in, and so on). The configuration introduces additional complexity compared to the choreographic implementation. Specifically, the forwarding rules the controller pushes to the OVS only govern traffic routing to the ports connected to the individual VNF interfaces. Therefore, for each VNF, it is necessary to adjust traffic management to complete the chain manually. In particular, for this scenario, the chain implementations (written in Python), require hardcoded traffic flow management to ensure correct operation, as illustrated in Listing 1.

Listing 1: Traffic flow management hardcoding in Python.

```
# Open raw socket to capture packets on INPUT interface
input_sock = socket.socket(socket.AF_PACKET,
    socket.SOCK_RAW, socket.ntohs(0x0800))
input_sock.bind((INPUT_INTERFACE, 0))

# Open raw socket to send packets on OUTPUT interface
output_sock = socket.socket(socket.AF_PACKET, socket.SOCK_RAW)
output_sock.bind((OUTPUT_INTERFACE, 0))

while True:
    packet, _ = input_sock.recvfrom(65535)  # Receive packet
    analyse_packet(packet)  # Analyse for asymmetry
    forward_packet(packet, output_sock)  # Forward to next VNF
```

[10] https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.2.pdf.

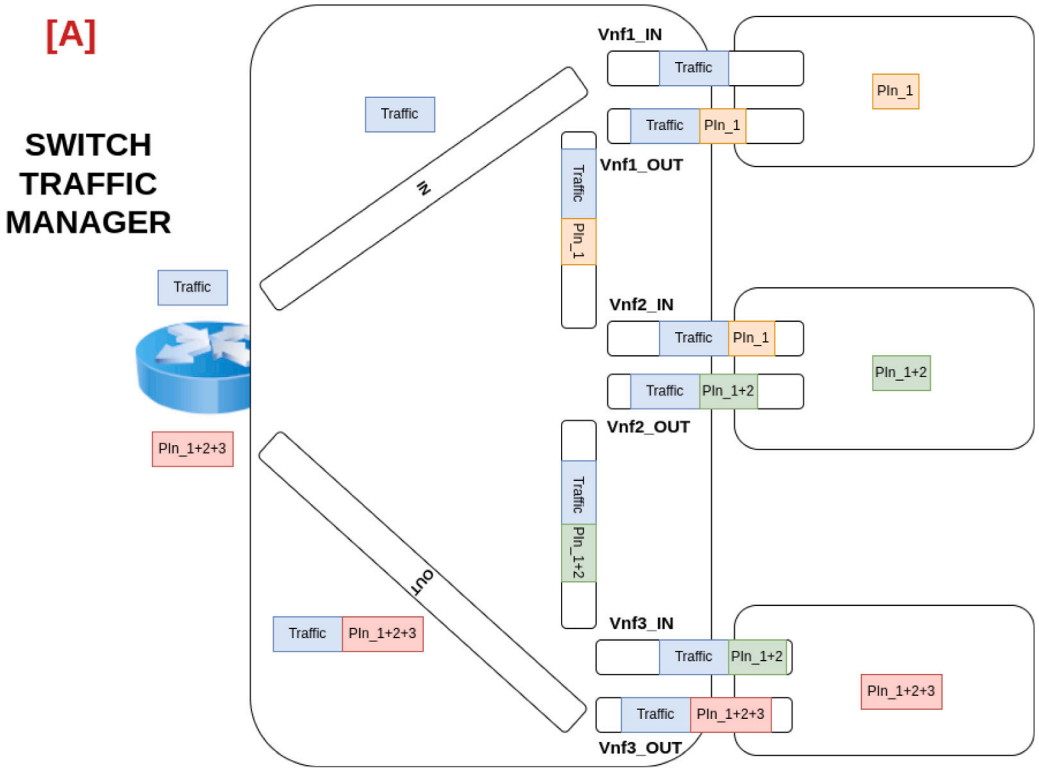


Fig. 4. Detailed vision of the chaining workflow.

As illustrated in Fig. 4, an essential aspect of this solution is that the mirrored traffic traverses the VNF chain and must ultimately return to the original switch S1 as a custom packet generated by the VNFs. Each VNF in the chain analyses the traffic and, based on its processing logic, can generate an additional packet to be inserted into the traffic flow. This additional packet is then forwarded along the chain to the next VNF. As a result, each VNF in the chain may receive the mirrored traffic and custom packets generated by the preceding VNFs based on their respective analyses. At the end of the processing pipeline, the final analysis result is returned to the original switch for further handling.

This configuration represents the only feasible option when implementing a service function chaining architecture over a traditional SDN approach. The following limitations emerge from this setup.

Mandatory traversal of the entire chain. In a classic SDN-based VNF chain, it is not possible to terminate the chain execution after the first VNF. Each data packet is required to sequentially traverse the entire chain of VNFs, even if subsequent functions are not strictly necessary for the given flow. This rigid chaining model reduces flexibility and increases processing overhead, especially in scenarios where selective or conditional VNF execution would be more efficient.

Communication between VNFs must Be forward-only. The architecture mandates a forward-only communication model between VNFs. This means that if two VNFs need to exchange information or coordinate their actions, the only viable method is through packet forwarding along the defined service path. Asynchronous or bidirectional communication patterns between VNFs are not supported natively. This constraint severely limits the design of dynamic, interactive network services that rely on feedback loops or event-driven behaviour.

Management and data traffic are mixed. Another critical limitation is the lack of separation between management traffic (used for orchestration, monitoring, and control) and regular data plane traffic. Both types of traffic share the same network paths and infrastructure resources. As a result, any delay, congestion, or performance degradation affecting the data traffic can also propagate to the management traffic, leading to cascading effects on service quality and control responsiveness. This co-mingling complicates troubleshooting, impairs scalability, and increases the risk of systemic failures.

6.2. Comparison with a hybrid SDN/orchestrated architecture

During the design and development phases, another possible strategy emerged for replicating the implementation of the choreographic approach. This approach can be termed as hybrid, since it does not strictly adhere to the SDN paradigm, but instead overlays an additional application layer that generates new messages to collect data from the VNFs, process the gathered information, and return the results. This approach does not align with the classical SDN model, as it leverages a construct designed initially to implement the concept of a learning switch and, more broadly, for network device management. We illustrate the corresponding scenario in Fig. 5.

In this implementation, traffic is mirrored on the switch to which the VNFs are connected using PACKET_IN message. A PACKET_IN is a message sent by an SDN switch to the central controller, containing a data packet. It is a notification that the switch has received a packet lacking forwarding instructions and are typically used to initiate handling of new data flows, prompting the controller to make forwarding decisions. In our case, we

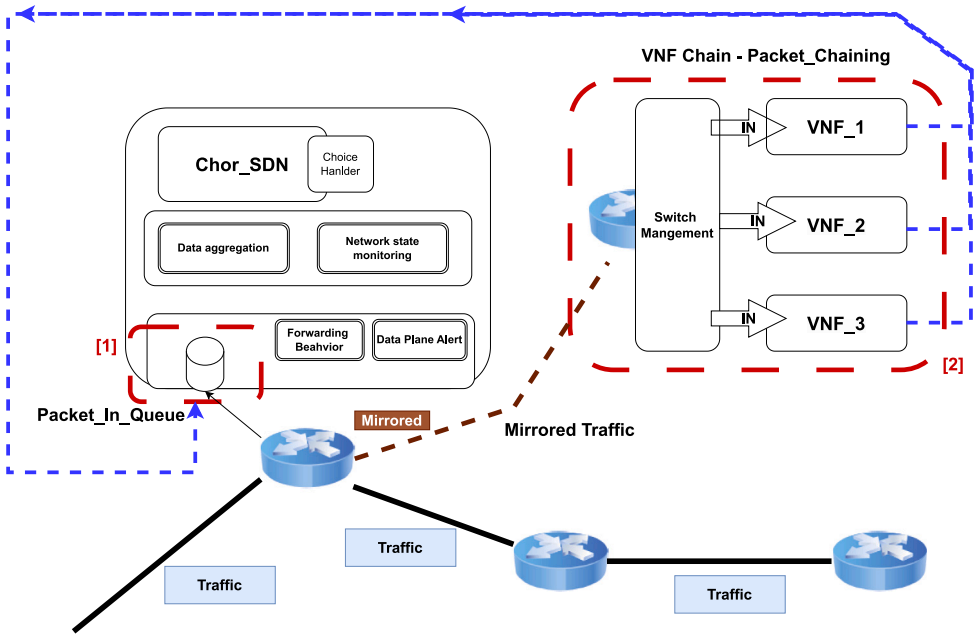


Fig. 5. High-level view of the chaining VNF and the Packet_In queue workflow.

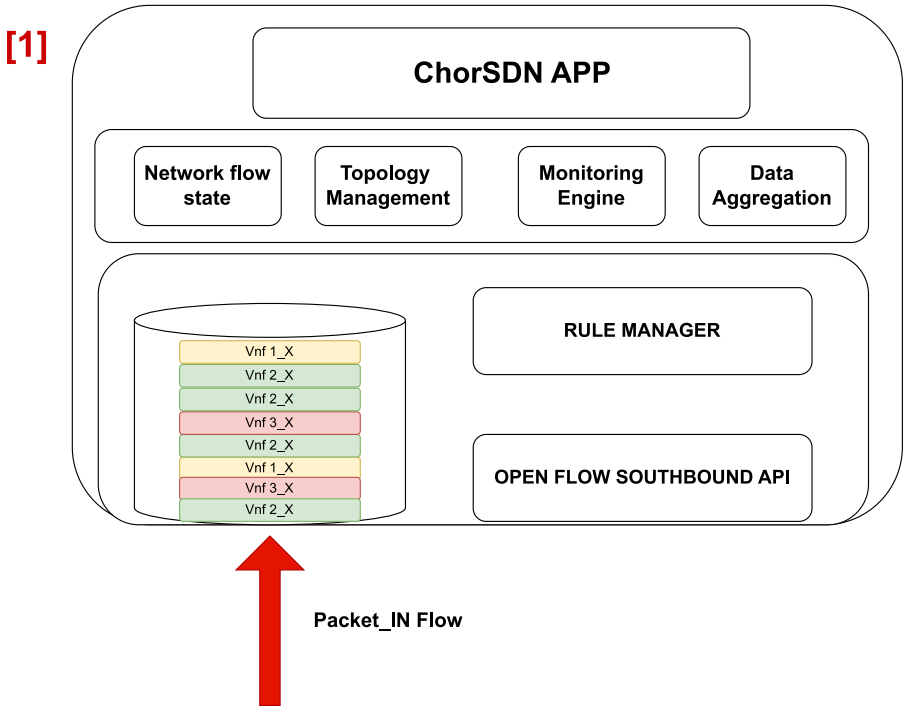


Fig. 6. Detailed view of the Packet_In queue inside the controller and rule engine. It represents the VNF flow as a set of queued Packet_In.

can "over-exploit" them to allow each VNF to send a specific PACKET_IN message to the controller upon deciding on a dedicated management interface. The controller can then collect and process these packets before forwarding them to higher layers.

As illustrated in Fig. 6, a significant drawback of this architecture is the bottleneck at the PACKET_IN Queue. The PACKET_IN interface for the controller is a single entry point that receives all PACKET_IN messages from all connected switches within the network. These packets arrive in non-deterministic order, depending on factors such as the distance from the controller and the specific routing paths taken by the switches. Additionally, since PACKET_IN messages are queued, efficient handling and distribution across multiple applications require the implementation of a priority mechanism and deep packet inspection from the controller. Such mechanisms introduce significant complexity and can lead to substantial performance degradation if PACKET_IN messages from different switches simultaneously are not properly managed.

This hybrid solution, albeit representing a non-standard use of the SDN paradigm, partially mitigates the problem of SDN solutions requiring the traversing of the entire chain of VNFs since it allows VNFs to communicate directly with the controller.

7. Evaluation: Performance analysis

In this section, we detail the performance differences in the setup and deployment processes among the three approaches: classical SDN, hybrid SDN/orchestrated, and choreographic (that we propose).

For the experiments, we used a PC with Ubuntu 22.04, 16 GB RAM, and an Intel® Core™ i7 processor. We assume that the necessary applications have been downloaded and compiled for all approaches and that the network devices (e.g., p4_switch and ovs_switch) are properly configured, and their respective connections are active.

7.1. Classical SDN setup

The first step under the classical SDN approach involves loading and instantiating the controller. One can generally perform this task in two ways:

- Downloading and installing the controller from source and its dependencies.
- Downloading and running a pre-configured container with the controller already set up.

For simplicity, we adopt the second approach in our tests. Specifically, we deploy the RYU controller[11] and instantiate it within our deployment.

After setting up the controller, one needs to create and instantiate the network switches. Once instantiated, the switches are visible to the controller, and the controller must be configured to operate with the appropriate flow rules. Specifically, the learning switch policy for all instantiated OVS switches must be set and properly configured to recognise mirroring between switches. Then, for our use case, three containers (one per VNF) are loaded by instantiating three pre-built container images and mounting a shared folder containing the function files inside the containers. The VNF chain then must be physically created by installing all the flow rules that allow the VNFs to redirect traffic to and from the switch and between VNFs themselves.[12] Finally, the application can be deployed on the controller by copying all application files into the corresponding reference directory that allows the application to be launched.

7.2. Hybrid SDN/orchestrated setup

The infrastructure setup for the hybrid SDN/orchestrated version is identical to the classical configuration in terms of underlying components, except for two key changes.

The first change requires to manually adapt the VNFs to generate control messages in the form of PACKET_IN messages. Since PACKET_IN messages carry packet payloads, they can be repurposed in this setup to act as privileged management messages for transmitting custom instructions to the controller along a predefined and protected path.

The second change affects instead the controller itself. By default, controllers interpret PACKET_IN messages strictly within the context of flow rule installation. Leveraging them to convey arbitrary or extended types of information requires modifying the controller's processing engine to accommodate a different semantic interpretation of the message content.

These adaptations are generally challenging to scale and highly dependent on the specific version and architecture of the SDN controller. As such, estimating the effort required to implement and maintain these changes is complex, context-dependent, and certainly non-negligible.

In particular, we note that since the queue of incoming PACKET_IN messages would consist of packets from multiple switches, defining the logic of the controller would require consideration of various factors, including: (a) network topology (i.e., the number of switches capable of sending PACKET_IN messages), (b) frequency of message transmission from the switches, and (c) average queuing time, delay, and controller processing time.

7.3. Choreographic setup

In the choreographic approach, unlike what was done for the Classical SDN approach, the Choral sources are compiled into four Java executables (.jar files) that represent the three VNFs and the controller function. Their deployment is then managed through an Infrastructure-as-Code (IaC) paradigm, defined by the Kathara script available at https://doi.org/10.5281/zenodo.15519089 that simplifies the infrastructure instantiation process.[13] From this step onward, it is only necessary to instantiate a container for each VNF and connect them to their corresponding virtual switches. The execution time for this operation is comparable to the container instantiation process for the VNF chain using the classical approach. The only additional requirement in this case is the need to instantiate a container with a pre-installed Java Virtual Machine, which results in a minimum container size of approximately 300 MB.

When starting the container, we pre-mount the folder of executable files, following the same approach used for the VNF chain for the Classical SDN approach. Therefore, at this stage, it is sufficient to execute the files, starting the application, and waiting for the results.

[11] https://github.com/faucetsdn/ryu.

[12] This task is automated by the script available at https://doi.org/10.5281/zenodo.15519089.

[13] Please note that Kathara cannot be used in the Classical SDN approach since it must issue specific flow rules to manage packet forwarding, and network behaviours such as service chaining must be explicitly orchestrated via flow tables.

Table 2
Decision time (in seconds) for the four attack scenarios across the three architectures.

Attack type	SDN classic	Custom SDN	Choreographic
High volume DDoS	12.6	10.3	4.2
Slow DDoS	15.4	15.0	14.2
Signature-Based attack	0.8	0.6	0.5
No-Signature attack	17.2	16.5	15.1

7.4. Traffic measurement

We conducted a series of 4 experiments to evaluate the efficiency of the three approaches in terms of decision time, defined as the interval between the start of an attack and the first detection signal by any VNF, reported back to the controller. We performed all tests in a controlled, virtualised environment, executing the following attacks:

- *High-Volume DDoS Attack.* This attack overwhelms the system with high-rate traffic, generated using `hping3`.[14] We configure 50 concurrent TCP SYN flood flows, each using randomised source IPs and ports, with a total packet rate of 50,000 per second. The traffic targeted the first-line VNF responsible for basic filtering and rate-limiting, stressing its ability to identify real-time volumetric threats.
- *Low-Rate DDoS (LDoS) Attack.* Using `hping3`, we emulate an LDoS scenario, sending traffic in intermittent bursts, with low average throughput but high burstiness. The attack aims to exploit timing-based weaknesses, bypassing the firewall and signature-based VNFs, and to trigger the detection by the ML-based VNF due to abnormal flow timing and packet variance.
- *Signature-Based Attack.* We perform a precise HTTP-based intrusion using a well-known ShellShock payload.[15] We craft the attack using `curl`, embedding the payload in the User-Agent header of an HTTP request. We intend to flag this traffic by the signature-based VNF, which equips a Suricata engine and a rule set containing the relevant exploit signature.
- *Signatureless Attack.* This scenario targets the ML-based VNF with an anomalous behaviour pattern lacking a known signature. A custom Modbus client implemented through the ModBusSploit tool[16] sends progressively increasing volumes of legitimate Modbus "read register" requests to a Modbus TCP server. The slow accumulation of unusual behaviour is designed to be detected based on deviation from the learned traffic baseline. To generate this traffic, we use a simple Python loop request.

Table 2 reports the detection times (in seconds) for the four attack scenarios across the three architectural approaches. The choreographic implementation consistently achieves the lowest decision time in all cases.

For the High-Volume DDoS attack, the choreographic approach reduces the detection time to 4.2 s, compared to 12.6 s in the classical SDN chain and 10.3 s in the custom SDN version. This corresponds to a reduction of approximately 66.7% with respect to the classical architecture and 59.2% compared to the custom SDN solution. In the Slow DDoS scenario, the improvement is more moderate: 14.2 s for the choreographic implementation versus 15.4 and 15.0 s for the classical and custom versions, respectively (a reduction of about 7.8% and 5.3%). For the Signature-Based attack, all approaches detect the threat rapidly, yet the choreographic version still provides the lowest latency (0.5 s), compared to 0.8 s for classical SDN and 0.6 s for the custom version. Finally, in the No-Signature attack scenario – where detection relies heavily on behavioural and ML-based analysis – the choreographic solution achieves a decision time of 15.1 s, improving over 17.2 s (classical) and 16.5 s (custom), corresponding to reductions of approximately 12.2% and 8.5%, respectively.

The most significant gain appears in high-volume attack scenarios, where rapid mitigation is critical to prevent service degradation or infrastructure overload. Reducing the detection time from 12.6 to 4.2 s substantially limits the attack window and reduces the number of malicious packets processed before mitigation rules are installed. Even in scenarios where improvements are more moderate (e.g., Slow DDoS and No-Signature attacks), the consistent reduction in decision latency demonstrates that eliminating controller-mediated coordination and enabling direct inter-VNF communication translates into measurable operational benefits. Moreover, these gains are achieved without introducing centralised bottlenecks, improving both scalability and responsiveness in distributed deployments.

For reproducibility purposes, all the scripts used for deploying and running the experiments are publicly available.[17] To conclude this section, we would like to underline some potential threats to the validity of our experiments.

Internal Validity The test environment's specific configurations may influence the accuracy of the results. Although we carefully controlled for variables such as traffic volume, attack type, and orchestration timing, subtle interactions between VNFs or timing discrepancies due to virtualization (e.g., scheduling delays across containers or VMs) could affect detection latency. Additionally, we generated the attacks in a synthetic and isolated environment, which may not fully capture the variability of real-world network noise and concurrent background traffic. As depicted in the deployment of the orchestrated solutions, a real-world scenario would need to include several network devices, not only the one deployed to create a clean testing environment.

External Validity Our findings may not be directly generalisable to other network environments. Our experiments represent a specific deployment of VNFs with a selected set of orchestration logic and traffic patterns. Different hardware, orchestration tools, or real-world implementations (e.g., hybrid cloud or SDN-based infrastructure) could yield different performance results. Furthermore, while we included representative DDoS and signature-based attacks, the attack set is not exhaustive and may not capture the full spectrum of adversarial behaviours in the wild.

Construct Validity While the chosen metric based on decision time (as defined in 7.4) is relevant for response improvement, it does not fully capture other dimensions of effectiveness, such as false positive rate, resource overhead, or resilience to adversarial evasion.

[14] https://github.com/antirez/hping.
[15] https://github.com/jeholliday/shellshock.
[16] https://github.com/C4l1b4n/ModBusSploit.
[17] https://doi.org/10.5281/zenodo.15519089.

8. Discussion: Advantages and limitations

We conclude by discussing the advantages and open challenges of using choreographic programming and looking at future work. We structure the discussion by comparing our proposal against the traditional SDN implementation with a centralised controller orchestrating all the function chains.

Advantage: Direct Intra VNF Communications The choreographic approach allows direct communication between VNFs. In classical SDN architectures, such communication is not possible unless hardcoded directly into the VNFs, which is discouraged since hardcoding communication makes the component difficult to port and extend. The best practice chosen by ETSI is instead to run all requests through the controller, following a star-like architecture where the controller mediates all communications. Using direct communication between the VNFs can save traffic (e.g., no need to have two communications with the orchestrator if a VNF has to send data to another one).

Advantage: No SDN Controller/Orchestrator The orchestration of VNF chains is typically implemented within the controller itself or as an application layer. The controller (often seen as a Network Operating System) is designed to interact with applications through a so-called northbound interface, similar to an operating system kernel that accesses device drivers. Traditionally, to create a VNF chain, it is necessary to create a new northbound application, that implements the communication logic between VNFs. This requires implementing the communication logic and adapting it to the proper controller like ONOS [36]. With the choreographic approach, we are not tied to any particular type of controller, and we are not required to follow any specific design pattern. We are not bound to use libraries and controller code that must be compatible with the rest of the environment and applications. Since choreographic programming allows independent communication among VNFs, the choreographic solution avoids the bottlenecks typical of controller-based SDN architecture.

Advantage: Security by Design In a classical SDN approach, one can verify the validity of network policies with formal model techniques such as reachability graphs [37] or by using atomic predicates [38] at the data plane level. With choreographic programming, we use a security-by-design approach for network development that avoids the typical communication problems of distributed systems (e.g., deadlocks, race conditions, etc.). Moreover, with a choreographic approach, the availability of a global overview of the entire system eases the task of verifying the global properties of the system at the application/logic level.

Advantage: Parallel VNF execution In the traditional approach, the VNFs workflow is often rigid and sequential: each VNF is executed one after another, leading to a linear progression of tasks. While this method is effective in ensuring that each function is processed in a controlled manner, it may also introduce bottlenecks and inefficiencies, especially when dealing with complex network architectures or high volumes of data. Choreographic programming removes the constraint of executing VNFs sequentially. Multiple VNFs can be initiated and processed simultaneously, without the need to wait for the completion of preceding tasks, unlocking new possibilities for optimising network performance and resource utilisation.

Challenge: failure handling Choreographic languages assume reliable communications. The only exception is the language theory presented in [39], which shows that one can relax this assumption, by allowing the choreographic language to handle local exceptions. Choral follows this strategy relying on the exception mechanism of Java and local failure recovery code [8, Sec. 2.5] which results in codebases that mix high-level choreographic interactions and low-level recovery strategies. Although Choral's object-orientation allows programmers to encapsulate the latter into high-level APIs, its type system can offer limited support to reason about the robustness of recovery strategies. Indeed, supporting programmers in writing robust and effective choreographies is still an open issue beyond Choral or even choreographic programming.

Challenge/Advantage: Handling Traffic Computational Load A concern regarding the use case we presented is the potential computational overhead introduced by packet replication to multiple VNFs. As illustrated in Section 5, the SA function distributes traffic to VOL, ML, and SIG concurrently, which might suggest increased resource consumption compared to traditional SDN implementations that merge multiple VNFs into a single pipeline to achieve line-rate processing speeds. However, this concern can be effectively addressed by exploiting Choral's flexibility in defining communication channels. The channel abstraction in Choral is not bound to a specific communication mechanism or middleware (cf. Section 2.3). Rather than implementing communications exclusively as point-to-point message passing, developers can define custom channel implementations that leverage broadcast or multicast capabilities available in the underlying network infrastructure. For instance, in a local network deployment, a single broadcast channel can efficiently replace multiple individual point-to-point communications from SA to the detection VNFs, significantly reducing both computational overhead and network traffic. This deployment-time flexibility allows the same choreographic specification to be instantiated with different channel implementations suited to the target environment. In scenarios where hardware-level multicast is available (e.g., edge deployments with programmable switches), the code generated from the choreographic specification can exploit these capabilities transparently. Conversely, in cloud environments, where direct multicast may not be supported, the same choreography can fall back to dedicated point-to-point communications or application-layer multicast.

Challenge: Knowledge of Choice When a choreography describes a choice between two possible branches, all affected participants must be (made) aware of the outcome to ensure that their local implementations agree on which branch to execute. In choreographies, this is called knowledge of choice (KoC). The standard solution for achieving KoC is communicating the choice outcome to the affected participants using special messages used by choreographic compilers to check that KoC is indeed achieved. Because of limitations in the current compilers, some of these communications might be redundant and ongoing work [21,40–42] is trying to address this by detecting and reducing them.

Advantage: Choreography-defined networks, beyond security. While our case study demonstrates the CDN approach in the context of network security – specifically for distributed DDoS detection and mitigation – we deem that this methodology's applicability can cover general SDN scenarios.

To briefly illustrate the versatility of CDN's approach, we present an additional application scenario that can benefit from the choreographic approach. The example showcases how CDNs can address coordination challenges in resource management, traffic optimisation, and service orchestration, hinting at other SDN-specific contexts where our methodology can help.

Let us consider a distributed load balancing scenario between three SDN network servers: S1, S2, S3. We include at the end of this section a Choral choreography that implements this scenario.

In the choreography, S1 monitors its CPU usage locally, if it detects overload, it informs both S2 and S3 simultaneously about its status, triggering the offload protocol. Following that protocol, S2 and S3 provide their CPU metrics so that S1 can check which of them can more easily accommodate part of its traffic load — the one with the lowest CPU usage.

Thus, S1 compares S2's and S3's CPU usages and indicates which of them it exclusively chooses to help manage its traffic load — notice that, in the Choral code, we use the enumerated values `ACCEPT` and `REJECT` alternatively to make it clear (from the point of view of the programmer) which server S1 accepts to share its traffic with (and which it rejects).

We can see the benefits of the choreographic approach also in this scenario, such as the absence of a centralised controller (S1, S2, and S3 coordinate directly without routing all messages through an SDN controller, eliminating bottlenecks) and high parallelism (e.g., the CPU usage requests to S2 and S3 can run in parallel, unlike the sequential VNF chaining in classical SDN).

```
class VNFOptimisation@(S1, S2, S3) {
    public void optimiseLoad() {
        if ( s1Metrics.cpuUsage > 90@S1 ) {
            ch12.< Status >select( Status@S2.OVERLOADED );
            ch13.< Status >select( Status@S3.OVERLOADED );
            Integer@S1 s2CpuUsage = s2Metrics.cpuUsage >> ch12::<Integer>com;
            Integer@S1 s3CpuUsage = s3Metrics.cpuUsage >> ch13::<Integer>com;
           if ( s2CpuUsage < s3CpuUsage ) {
                ch12.< Response >select( Response@S2.ACCEPT );
                ch13.< Response >select( Response@S3.REJECT );
                commitTraffic( S1, S2 );
           } else {
                ch12.< Response >select( Response@S2.REJECT );
                ch13.< Response >select( Response@S3.ACCEPT );
                commitTraffic( S1, S3 );
           }
        } else {
            ch12.< Status >select( Status@S2.OK );
            ch13.< Status >select( Status@S3.OK );
        }
    }
}
```

Choral

In the example above, we assume reliable communication, but real-world deployments must account for potential node failures. The CDN approach can naturally accommodate such scenarios through Choral's type system and channel abstractions. For instance, the responses from S2 and S3 could be wrapped in `Optional` types, allowing S1 to handle cases where either or both servers fail to respond within a designated timeout. In this way, if communication timeouts occur, the choreography implements graceful-degradation protocols.

Advantage: Efficient compilation and deployment. The choreography compiler in Choral is highly efficient, with compilation times that are negligible for the choreographies presented in this work. Benchmarks reported in [8] demonstrate that even complex choreographies involving 10 participants and multiple synchronisation and choice points compile in the order of a few milliseconds. This efficiency ensures that the choreographic approach is not only theoretically sound but also practically feasible for real-time applications, as the overhead introduced by the compilation process is minimal.

Beyond compilation, the deployment and redeployment of choreography-defined networks are streamlined by the modular and automated nature of the approach. Since the Choral compiler generates executable code for each participant, updates or changes to the choreography can be quickly propagated to the system without manual intervention. This makes the approach particularly well-suited for Continuous Integration/Continuous Deployment (CI/CD) pipelines, where rapid iteration and automated testing are critical. The ability to redeploy updated choreographies with minimal downtime further enhances the agility and maintainability of networked systems, making it easier to adapt to evolving requirements or emerging threats.

9. Conclusion

We present a novel methodology for service composition in Software Defined Networks and Network Function Virtualisation, specifically tailored for cloud environments. Departing from conventional sequential service chaining, the approach uses choreographies to model Virtual Network Functions' roles and interactions. We showcase several advantages of the proposed approach, such as a holistic view of interactions and automatic code generation for each VNF, which eliminates the need for a centralised control node, reducing concurrency issues and communication overhead with the controller.

The validity of the proposed approach derives from the choreographic programming paradigm, which guarantees the implementation of a correct-by-construction VNF architecture given a choreography. We demonstrate the feasibility of the proposed approach via a practical case study where we used the state-of-the-art choreographic language Choral to develop a distributed composition of several VNFs collaborating to analyse network traffic and detect security threats.

We envision two direct future directions. The first one encompasses challenges related to error handling and recovery (cf. Section 8). The second one envisages the definition of a meta-choreography that could define the infrastructural interactions needed to deploy the VNFs, by interacting with the SDN controller, and the dynamic and flexible forwarding of traffic by means of programmable data plane devices. Further extensions of this undertaking include the development of a new compiler for Choral that, instead of generating Java, could output P4 code, so that some parts of the distributed application may be executed on programmable switches instead of containers or virtual machines.

CRediT authorship contribution statement

Saverio Giallorenzo: Writing – review & editing, Writing – original draft, Validation, Software, Investigation. **Jacopo Mauro:** Writing – review & editing, Validation, Supervision, Project administration, Conceptualization. **Andrea Melis:** Writing – review & editing, Writing – original draft, Software, Methodology, Investigation, Conceptualization. **Fabrizio Montesi:** Writing – review & editing, Validation, Supervision. **Marco Peressotti:** Writing – review & editing, Writing – original draft, Validation, Software, Investigation. **Marco Prandini:** Writing – review & editing, Investigation, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

Partially supported by Villum Fonden, Denmark (grant no. 29518). Co-funded by the European Union (ERC, CHORDS, 101124225). Partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU. Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

Appendix. Our case study in Choral

In this Appendix, we report the Choral code for implementing the case study we discuss in Section 4. We focus on the choreographic code and omit the auxiliary Java implementations of the local functionalities (e.g., extraction of packet features at SA, volumetric analysis at VOL) as this is standard.

```
public class ChorSDN@( SW, SA, VOL, ML, SIG ){

 // Network topology as point-to-point symmetric channels
 SymChannel@(SW, SA)  sa_sw;   SymChannel@(SA, VOL) sa_vol; SymChannel@(SA, ML)  ml_sa;
 SymChannel@(SA, SIG) sa_sig;  SymChannel@(VOL,ML)  ml_vol;
 SymChannel@(VOL,SIG) sig_vol; SymChannel@(ML, SIG) ml_sig;

 /* Java classes implementing local functionalities */
 Switch@SW sw;                   // placeholder interface (only for type checking)
 FeatureAnalyser@SA sa_a;        // packet feature extraction as SA
 SignatureAnalyser@VOL vol_a;    // analysis at VOL
 SignatureAnalyser@ML ml_a;      // analysis at ML
 FlowLabellerAnalyser@SIG sig_a; // analysis and labelling at SIG

 /* constructor omitted */

 void analyseTraffic() {
  Packet@SA p = sa_sw.< Packet >com( sw.getPacket() );
  PacketFeature@VOL vol_f = sa_vol.<PacketFeature>com(sa_a.extractFeaturesVOL(p));
  PacketFeature@ML  ml_f  = ml_sa.<PacketFeature>com(sa_a.extractFeaturesML(p));
  PacketFeature@SIG sig_f = sa_sig.<PacketFeature>com(sa_a.extractFeaturesSIG(p));

  Result@VOL vol_r = vol_a.analyse( vol_f );
  Result@ML  ml_r  = ml_a.analyse( ml_f );
  Result@SIG sig_r = sig_a.analyse( sig_f );

  switch( vol_r ) {
   case MALICIOUS -> {
    sa_vol.< Result >select( Result@VOL.MALICIOUS );
    ml_vol.< Result >select( Result@VOL.MALICIOUS );
    sig_vol.< Result >select( Result@VOL.MALICIOUS );
    sa_sw.< Result >select( Result@SA.MALICIOUS );
    sa_a.genStm() >> ml_sa::< DataStream >com >> ml_a::filterAndAnalyse;
    Optional@SIG<DataSignature> s1 = ml_a.genSignature()
                                  >> ml_sig::<Optional<DataSignature>>com;
    Optional@SIG<DataSignature> s2 = vol_a.genSignature()
                                  >> sig_vol::<Optional<DataSignature>>com;
    sig_a.labelFlow( s1, s2 );
   }
   case BENIGN -> {
    sa_vol.< Result >select( Result@VOL.BENIGN );  // SA
    ml_vol.< Result >select( Result@VOL.BENIGN );  // SIG
    sig_vol.< Result >select( Result@VOL.BENIGN ); // ML
    sa_sw.< Result >select( Result@SA.BENIGN );    // SW
   }
   default -> {
    sa_vol.< Result >select( Result@VOL.UNDECIDED );  // SA
    ml_vol.< Result >select( Result@VOL.UNDECIDED );  // SIG
    sig_vol.< Result >select( Result@VOL.UNDECIDED ); // ML
    sa_sw.< Result >select( Result@SA.UNDECIDED );    // SW
    analyseTraffic();
   } } } }
```

Choral

References

[1] L. Yang, R. Dantu, T. Anderson, R. Gopal, Forwarding and Control Element Separation (ForCES) Framework, Tech. Rep., 2004.
[2] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, R. Boutaba, Network function virtualization: State-of-the-art and research challenges, IEEE Commun. Surv. & Tutorials 18 (1) (2015) 236–262.
[3] C. Sun, J. Bi, Z. Zheng, H. Yu, H. Hu, NFP: Enabling network function parallelism in NFV, in: Proc. Conference of the ACM Special Interest Group on Data Communication, 2017, pp. 43–56.
[4] T. Leesatapornwongsa, J.F. Lukman, S. Lu, H.S. Gunawi, TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems, in: T. Conte, Y. Zhou (Eds.), ASPLOS, ACM, 2016, pp. 517–530, http://dx.doi.org/10.1145/2872362.2872374.
[5] E.M. Clarke, W. Klieber, M. Novácek, P. Zuliani, Model checking and the state explosion problem, in: B. Meyer, M. Nordio (Eds.), Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures, in: Lecture Notes in Computer Science, vol. 7682, Springer, 2011, pp. 1–30, http://dx.doi.org/10.1007/978-3-642-35746-6_1.
[6] F. Montesi, Choreographic Programming (Ph.D. thesis), IT University of Copenhagen, 2013.
[7] F. Montesi, Introduction to Choreographies, Cambridge University Press, 2023, http://dx.doi.org/10.1017/9781108981491.
[8] S. Giallorenzo, F. Montesi, M. Peressotti, Choral: Object-oriented choreographic programming, ACM Trans. Program. Lang. Syst. 46 (1) (2024) 1:1–1:59, http://dx.doi.org/10.1145/3632398.
[9] S. Giallorenzo, J. Mauro, A. Melis, F. Montesi, M. Peressotti, M. Prandini, Choreography-defined networks: A case study on DoS mitigation, in: W. Gaaloul, M. Sheng, Q. Yu, S. Yangui (Eds.), Service-Oriented Computing - 22nd International Conference, ICSOC 2024, Tunis, Tunisia, December 3-6, 2024, Proceedings, Part II, in: Lecture Notes in Computer Science, vol. 15405, Springer, 2024, pp. 243–259, http://dx.doi.org/10.1007/978-981-96-0808-9_18.
[10] F. Hu, Q. Hao, K. Bao, A survey on software-defined network and openflow: From concept to implementation, IEEE Commun. Surv. & Tutorials 16 (4) (2014) 2181–2206.
[11] F. Callegati, W. Cerroni, C. Contoli, R. Cardone, M. Nocentini, A. Manzalini, SDN for dynamic NFV deployment, IEEE Commun. Mag. 54 (10) (2016) 89–95, http://dx.doi.org/10.1109/MCOM.2016.7588275.
[12] A. Liatifis, P. Sarigiannidis, V. Argyriou, T. Lagkas, Advancing sdn from openflow to p4: A survey, ACM Comput. Surv. 55 (9) (2023) 1–37.
[13] E.F. Kfoury, J. Crichigno, E. Bou-Harb, An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends, IEEE Access 9 (2021) 87094–87155.
[14] J. Xing, K.-F. Hsu, M. Kadosh, A. Lo, Y. Piasetzky, A. Krishnamurthy, A. Chen, Runtime programmable switches, in: USENIX, 2022, pp. 651–665.
[15] A. Al Sadi, M. Savi, A. Melis, M. Prandini, F. Callegati, Unleashing dynamic pipeline reconfiguration of P4 switches for efficient network monitoring, IEEE Trans. Netw. Serv. Manag. TBD (TBD) (2024) TBD, http://dx.doi.org/10.1109/TNSM.2024.TBD.
[16] M. Carbone, F. Montesi, Deadlock-freedom-by-design: multiparty asynchronous global programming, in: R. Giacobazzi, R. Cousot (Eds.), POPL.
[17] M. Dalla Preda, M. Gabbrielli, S. Giallorenzo, I. Lanese, J. Mauro, Dynamic choreographies: Theory and implementation, Log. Methods Comput. Sci. 13 (2) (2017) http://dx.doi.org/10.23638/LMCS-13(2:1)2017.
[18] A.K. Hirsch, D. Garg, Pirouette: higher-order typed functional choreographies, Proc. ACM Program. Lang. 6 (POPL) (2022) 1–27, http://dx.doi.org/10.1145/3498684.
[19] G. Shen, S. Kashiwa, L. Kuper, HasChor: Functional choreographic programming for all (functional pearl), Proc. ACM Program. Lang. 7 (ICFP) (2023) 541–565, http://dx.doi.org/10.1145/3607849.
[20] H. Hüttel, I. Lanese, V.T. Vasconcelos, L. Caires, M. Carbone, P. Deniélou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, H.T. Vieira, G. Zavattaro, Foundations of session types and behavioural contracts, ACM Comput. Surv. 49 (1) (2016) 3:1–3:36, http://dx.doi.org/10.1145/2873052.
[21] L. Lugovic, F. Montesi, Real-world choreographic programming: Full-duplex asynchrony and interoperability, Art Sci. Eng. Program. 8 (2) (2023) http://dx.doi.org/10.22152/PROGRAMMING-JOURNAL.ORG/2024/8/8.
[22] T.-M. Nguyen, M. Minoux, S. Fdida, Optimizing resource utilization in NFV dynamic systems: New exact and heuristic approaches, Comput. Netw. 148 (2019) 129–141.
[23] L. He, L. Li, Y. Liu, Towards chain-aware scaling detection in nfv with reinforcement learning, in: 2021 IEEE/ACM 29th International Symposium on Quality of Service, IWQOS, IEEE, 2021, pp. 1–10.
[24] A. Leivadeas, M. Falkner, A survey on intent-based networking, IEEE Commun. Surv. & Tutorials 25 (1) (2023) 625–655, http://dx.doi.org/10.1109/COMST.2022.3215919.
[25] K.-Y. Chen, S. Liu, Y. Xu, I.K. Siddhrau, S. Zhou, Z. Guo, H.J. Chao, Sdnshield: nfv-based defense framework against ddos attacks on sdn control plane, IEEE/ACM Trans. Netw. 30 (1) (2021) 1–17.
[26] J. Hasneen, K.M. Sadique, A survey on 5G architecture and security scopes in SDN and NFV, in: ICCET, Springer, 2022, pp. 447–460.
[27] S. Lakshmanan Thirunavukkarasu, M. Zhang, A. Oqaily, G.S. Chawla, L. Wang, M. Pourzandi, M. Debbabi, Modeling NFV deployment to identify the cross-level inconsistency vulnerabilities, in: 2019 IEEE International Conference on Cloud Computing Technology and Science, CloudCom, 2019, pp. 167–174, http://dx.doi.org/10.1109/CloudCom.2019.00034.
[28] A. Huff, G. Venâncio, V.F. Garcia, E.P. Duarte, Building multi-domain service function chains based on multiple NFV orchestrators, in: IEEE NFV-SDN, 2020, pp. 19–24, http://dx.doi.org/10.1109/NFV-SDN50289.2020.9289849.
[29] A. Melis, S. Layeghy, D. Berardi, M. Portmann, M. Prandini, F. Callegati, P-scor: Integration of constraint programming orchestration and programmable data plane, IEEE Trans. Netw. Serv. Manag. 18 (1) (2020) 402–414.
[30] D. Arp, E. Quiring, F. Pendlebury, A. Warnecke, F. Pierazzi, C. Wressnegger, L. Cavallaro, K. Rieck, Dos and don'ts of machine learning in computer security, in: 31st USENIX Security Symposium, USENIX Security 22, 2022, pp. 3971–3988.
[31] A.A. Sadi, M. Savi, A. Melis, M. Prandini, F. Callegati, Unleashing dynamic pipeline reconfiguration of P4 switches for efficient network monitoring, IEEE Trans. Netw. Serv. Manag. (2024) 1, http://dx.doi.org/10.1109/TNSM.2024.3377538.
[32] R. Doriguzzi-Corin, S. Millar, S. Scott-Hayward, J. Martínez-del Rincón, D. Siracusa, Lucid: A practical, lightweight deep learning solution for DDoS attack detection, IEEE TNSM 17 (2) (2020) 876–889, http://dx.doi.org/10.1109/TNSM.2020.2971776.
[33] A. Al Sadi, M. Savi, D. Berardi, A. Melis, M. Prandini, F. Callegati, Real-time pipeline reconfiguration of P4 programmable switches to efficiently detect and mitigate ddos attacks, in: ICIN, IEEE, 2023, pp. 21–23.
[34] CICIDS2017 Dataset, https://Www.Unb.Ca/Cic/Datasets/Ids-2017.Html.
[35] Nuclei discovery project, https://Github.Com/Projectdiscovery/Nuclei-Templates/.
[36] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, et al., ONOS: towards an open, distributed SDN OS, in: Proceedings of the Third Workshop on Hot Topics in Software Defined Networking, 2014, pp. 1–6.
[37] D. Berardi, F. Callegati, A. Melis, M. Prandini, Technetium: Atomic predicates and model driven development to verify security network policies, in: 2020 IEEE 17th Annual Consumer Communications & Networking Conference, CCNC, IEEE, 2020, pp. 1–6.
[38] H. Yang, S.S. Lam, Real-time verification of network properties using atomic predicates, IEEE/ACM Trans. Netw. 24 (2) (2015) 887–900.
[39] F. Montesi, M. Peressotti, Choreographies meet communication failures, 2017, http://dx.doi.org/10.48550/arXiv.1712.05465, CoRR abs/1712.05465, arXiv:1712.05465, URL https://arxiv.org/abs/1712.05465.
[40] S. Basu, T. Bultan, Automated choreography repair, in: FASE, in: Lecture Notes in Computer Science, vol. 9633, Springer, 2016, pp. 13–30.
[41] L. Cruz-Filipe, F. Montesi, A core model for choreographic programming, TCS 802 (2020) 38–66, http://dx.doi.org/10.1016/J.TCS.2019.07.005.
[42] S. Jongmans, P. van den Bos, A predicate transformer for choreographies - computing preconditions in choreographic programming, in: I. Sergey (Ed.), ESOP, in: Lecture Notes in Computer Science, vol. 13240, Springer, 2022, pp. 520–547, http://dx.doi.org/10.1007/978-3-030-99336-8_19.