Future Generation Computer Systems 176 (2026) 108121

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

Energy efficiency support for software defined networks: a serverless computing approach

Fatemeh Banaie [a], Karim Djemame [b,*], Abdulaziz Alhindi [b,c], Vasilios Kelefouras [d]

[a] School of Computing and Engineering, University of Huddersfield, UK
[b] School of Computer Science, University of Leeds, UK
[c] College of Computer, Qassim University, Kingdom of Saudi Arabia
[d] School of Engineering, Computing and Mathematics, University of Plymouth, UK

ARTICLE INFO

Keywords:
Network management
Software Defined Networks
Serverless computing
Network function virtualisation
Energy efficiency

ABSTRACT

Automatic network management strategies have become paramount for meeting the needs of innovative real-time and data-intensive applications, such as those in the Internet of Things. However, the ever-growing and fluctuating demands for data and services in such applications require more than ever an efficient, scalable, and energy-aware network resource management. To address these challenges, this paper introduces a novel approach that leverages a modular architecture based on serverless functions within an energy-aware environment. By deploying SDN services as Functions as a Service (FaaS), the proposed approach enables dynamic, on-demand network function deployment, achieving significant cost and energy savings through fine-grained resource provisioning. Unlike previous monolithic SDN approaches, this work disaggregates SDN control plane into modular, serverless components, transforming tightly integrated functionalities into independent, on-demand services while ensuring performance, scalability, and energy efficiency. An analytical model is presented to approximate the service delivery time and power consumption, as well as an open source prototype implementation supported by an extensive experimental evaluation. Experimental results demonstrate significant improvement in energy efficiency compared to traditional approaches, highlighting the potential of this approach for sustainable network environments.

1. Introduction

1.1. Background

The unprecedented growth of data traffic, driven by the expanding use of smart devices, has highlighted the challenges of managing and controlling information in existing networks [1,2]. Traditional networking infrastructures, originally designed for specific types of traffic like monotonous text-based content, struggle to support the increasing demands of interactive, dynamic multimedia streams. Furthermore, the emergence of the Internet of Things (IoT), coupled with the availability of cloud computing services, has led to a new generation of advanced services. These services not only impose stringent communication requirements but also emphasise the critical need for energy-efficient solutions to support innovative IoT use cases. Consequently, there is a clear need for adaptive, scalable, and energy-aware network management strategies. Enhancing energy efficiency not only ensures sustainable operations but also plays a vital role in improving overall network performance and user experiences (QoE) in such dynamic environments [3,4].

Software-Defined Networking (SDN) and Network Function Virtualisation (NFV) play pivotal roles in enabling the transition to next-generation IoT networks by addressing scalability, flexibility, and management challenges inherent in IoT deployments. SDN separates control logic from hardware, enabling flexible and efficient resource management. This separation allows a remote software controller to program the forwarding states in the data plane based on network policies. On the other hand, NFV enables the cost-efficient implementation of network functions by allowing shared physical resources, either locally or in the cloud [5]. The network virtualisation technique not only offers a reduced cost by sharing the network infrastructure but also improves time to market for novel applications. Therefore, these key tenets are critical for facilitating innovative applications, enabling network slicing, and ensuring a superior user experience (QoE) through simplified network management.

* Corresponding author.
E-mail address: k.djemame@leeds.ac.uk (K. Djemame).

https://doi.org/10.1016/j.future.2025.108121
Received 28 November 2024; Received in revised form 20 August 2025; Accepted 4 September 2025
Available online 9 September 2025

Despite significant advancements, current SDN solutions still face challenges due to their predominantly monolithic design. Traditional SDN controllers are often constructed as tightly integrated, single-application entities with rigidly defined programming interfaces, which restrict their adaptability and scalability. In response, the adoption of microservices-based architectures has introduced a shift toward more modular and distributed SDN solutions [6,7]. These architectures decompose the service plane into microservices, enabling scalable solutions for service and data planes. This enhances SDN's capabilities in packet forwarding, service chaining, and efficient network service deployment, as seen in microservices-based NFV architectures and modular frameworks. However, even with these advancements, existing modular SDN frameworks fail to deliver truly dynamic, on-demand service provisioning. This limitation not only restricts their ability to address the complex, variable requirements of next-generation network environments but also lowers the energy efficiency needed for sustainable, high-performance systems.

1.2. Motivation

Traditional modular SDN efforts primarily focus on decomposing the service plane or developing modular solutions for the data plane. However, these efforts often retain significant interdependence between components, meaning that changes or failures in one module might still affect others. This work introduces a paradigm shift that disaggregates the control plane into independently deployable services over a serverless architecture. Unlike conventional approaches, which rely on monolithic SDN controllers, this architecture dynamically allocates network resources and executes essential SDN functions only when needed. This disaggregation enhances resource efficiency by launching and orchestrating VNFs in a serverless environment, providing a proxy for energy consumption at the function level and making VNF instantiation and orchestration significantly more energy- and resource-efficient [8].

The proposed approach simplifies application development by removing dependencies on specific interfaces. By adopting standard APIs for seamless connectivity and enabling the on-demand deployment of services, it improves network flexibility, adaptability, and scalability. The services are typically short-lived, event-driven tasks that can be efficiently orchestrated in a distributed environment.

VNFs enable a service-specific overlay on the existing network topology, as illustrated in Fig. 1, with the potential to be deployed on a serverless platform, as noted by [8]. The serverless paradigm provides a resource-efficient alternative to virtual machines (VMs) and containers, facilitating serverless function-based network services (i.e., *Function as a Service (FaaS)*) [9]. By decoupling service execution from fixed infrastructure, serverless platforms offer cost-efficient, scalable resource provisioning with minimal operational overhead. Additionally, they offer fine-grained resource control, which enhances energy efficiency in network service delivery: 1) resources are allocated and scaled on-demand, therefore eliminating the need for pre-provisioning or over-provisioning of servers, and 2) serverless charges are based on actual execution time (usually measured in milliseconds), therefore preventing resource idling. Since serverless architectures are inherently suited for modular, event-driven applications like distributed SDN controllers, they align well with the evolving needs of next-generation Internet network [10].

1.3. Contributions

In this paper, we present a modular, microservice-based SDN architecture that leverages serverless platforms to provide energy-efficient and scalable service provisioning in network infrastructures. Building upon our earlier work [11], which demonstrated the proof of concept, this paper advances the proposal by providing an expanded experimental evaluation and introducing a comprehensive analytical model to assess performance and energy efficiency. The primary contributions of this work are as follows:

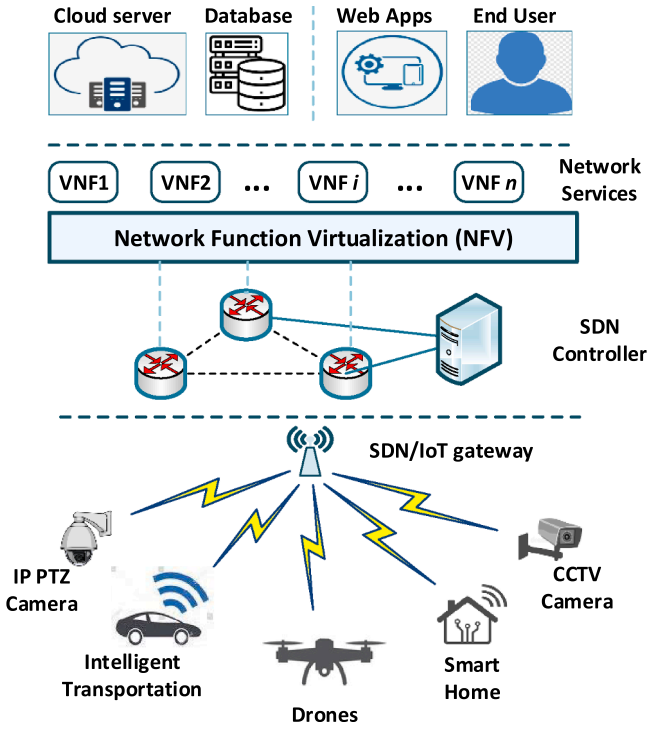


Fig. 1. Microservice-based IIoT service platform.

- **Microservice-based and Distributed SDN Architecture:** We propose a modular SDN architecture that disaggregates the control plane by breaking down SDN controllers into essential core functionalities (e.g., flow and topology management) while enabling non-core services (e.g., firewall, traffic monitoring, IoT services) to be dynamically deployed using serverless functions. This distributed approach allows network services to be divided into modular, independently deployable components that can be orchestrated in a serverless environment. The architecture ensures scalability and energy efficiency for next-generation applications, paving the way for more sustainable and efficient technological solutions.
- **Analytical model:** An analytical model is developed to approximate service delivery time and power consumption, offering insights into the efficiency of the proposed system.
- **Prototype implementation:** A prototype implementation of the modular SDN architecture is provided using widely adopted open-source software, including ONOS [6], OpenFaaS [12], and gRPC [13]. The performance of the implementation is validated through extensive experiments, comparing energy efficiency and latency against traditional Docker-based [14] SDN deployments.
- **Open-source platform:** The modular SDN platform implemented with serverless principles, is open-Source and accessible on GitHub[1], providing collaboration within the research community.

The rest of the paper is organised as follows. Section 2 provides an overview of the related literature. Section 3 describes the proposed architecture along with the functional definition of its components and interfaces. In section 4 we present the analytical model to approximate the service delivery time and power consumption, and this is followed by its evaluation in section 5. The modular SDN prototype implementation is explained in section 6, and performance results discussed in section 7. Section 8 concludes the paper.

[1] https://github.com/EDGNSS

2. Related work

Improving the energy efficiency of SDN has been a significant research focus since the introduction of this technology. [15] provides a comprehensive overview of methods to enhance SDN energy efficiency, categorised into three main strategies: traffic-aware, end-host-aware, and rule placement techniques. Traffic-aware and end-host-aware approaches focus on monitoring network traffic to identify devices and links that can be deactivated without impacting performance, thereby reducing energy consumption. Rule placement strategies, on the other hand, aim to minimise energy use by optimising the number of flow entries in Ternary Content Addressable Memory (TCAM).

[16] investigates the control placement problem, proposing an approach that partitions the network into sub-networks and assigns each to a controller. This model balances switch-to-controller latency and energy efficiency by deactivating unused links and rerouting flows through energy-efficient paths. Their coalition game-based mathematical model demonstrated a 22 % reduction in energy consumption. Similarly, [17] explores enhancing SDN controller energy efficiency by leveraging multi-core processors operating at lower frequencies. Their end-host-aware parallel implementation distributes workloads across cores, achieving a 28 % reduction in energy usage. The effort to optimise load balancing across controllers are addressed in [18], where a traffic-aware approach redistributes loads from heavily burdened controllers to lighter ones. This solution includes migrating switches through energy-efficient rerouting algorithms that preserve service-level agreements. Their EERAS model selects the shortest, most energy-efficient paths for switch migration, achieving a 25 % energy reduction and a 20 % performance improvement.

The SDN controller, often developed as a monolithic entity, performs all the control plane tasks, posing challenges for scalability and energy efficiency. Some researches have explored modular SDN controllers to simplify development and maintenance of the applications [19]. While they work on disaggregation of the SDN control plane, these approaches rely on microservices, which are inherently static and resource-intensive. The integration of serverless platforms into SDN/NFV architectures can help to enhance scalability and energy efficiency. [8] highlights how event-driven serverless functions can improve energy consumption in SDN/NFV deployments by optimising resource usage. Our work advances their concept by leveraging the serverless paradigm, enabling dynamic, on-demand execution of control-plane functions. By deploying disaggregated SDN components as serverless functions, the proposed approach achieves significant improvements in scalability, energy efficiency, and cost-effectiveness.

[20] propose a framework for efficient dispatching of stateless tasks on serverless platforms, optimising response times and fairness. Their evaluation shows that SDN controllers can benefit from serverless platforms by alleviating network congestion. Similarly, a high-performance serverless platform for NFV is proposed by Shen et al. [21], focusing on reducing latency through state management, optimised NF execution models, and reduced packet delays. The work presented in [22] proposes a methodology for application decomposition into fine-grained functions and energy-aware function placement on a cluster of edge computing devices subject to user-specified QoS guarantees.

[23] presents the EneA-FL scheme to promote smart energy management in serverless computing scenarios where Federated Learning clients are characterised by highly heterogeneous computing capabilities and energy budgets. The proposed approach dynamically adapts to optimise the training process while fostering seamless interaction between IoT devices and edge nodes. Experimental analysis shows that EneA-FL is able to reduce energy consumption yet yielding faster convergence of models to the target accuracy. [24] leverages workload awareness to optimise energy consumption and tackles the power control problem in data centres from the perspective of serverless computing. The energy-management framework automatically splits the user-provided end-to-end application Service Level Objectives into per function deadlines that minimise the total energy consumption. Functions are profiled online and, based on their deadlines, optimal core frequencies are picked.

Further research into serverless computing platforms highlights their energy efficiency compared to traditional container-based solutions. For instance, [25] demonstrate that OpenFaaS consumes significantly less power than Docker containers under memory-intensive workloads, achieving a 58 % reduction in energy consumption when compared to Kubernetes-based [26] implementations. [27] present an energy-efficient pluggable solution to scheduling in Kubernetes in a serverless computing environment through the integration of a Machine Learning-based model into the scheduler. In terms of overall metrics this solution achieved 8 % reduction in energy consumption at the cost of a directly correlated loss in performance, thus maintaining the application QoS. [28] investigates the key factors influencing serverless energy consumption in SDN by examining the correlations between the energy usage of serverless applications and traffic flow patterns, as such correlations can play a critical role in developing an energy-aware scheduling framework. The findings reveal that several traffic patterns, e.g. total number of packets in a flow, packet length, inter-arrival time between packets, strongly or moderately impact energy consumption in serverless environment.

In summary, existing work has focused on optimising energy consumption through specific mechanisms such as traffic-aware routing, controller placement, rule optimisation, and energy-efficient resource allocation in serverless contexts. This paper goes further as it aims at realising the concept of modular SDN controller based on serverless functions in an energy-aware environment. This approach not only enhances the flexibility and scalability of SDN systems but also presents an analytical model that quantifies the impact on power consumption and service delivery times, an area largely unexplored in prior research.

3. Microservice-based provisioning of network services

This section presents the proposed microservice-based network management approach. The innovation lies in the utilisation of network programmability within the context of Network Function Virtualisation (NFV) and the exploration of the benefits the serverless computing paradigm offers [29]. As noted, SDN offers scalable network management by separating the control plane and data plane in a network. The control plane consists of two main components: (i) SDN core functionalities, such as flow services and topology services, and (ii) management applications, such as traffic monitoring, load-balancing, and firewall. These components communicate requirements via Northbound and Southbound Application Program Interfaces (APIs). However, the centralised management of the network raises potential issues in terms of failure, scalability, and service latency, which aims to be resolved by distributed deployment of the controller [2,19,30]. Existing SDN controllers adopt a monolithic design that aggregates all functions into a single program. This approach constrains its ability to deploy a new service independently from other services. Unlike existing monolithic architectures, the proposed microservice-based SDN Architecture adopts a modular and flexible approach by breaking down SDN functionalities into independent microservices.

For this objective, we have developed the controller as a composition of containerised services, tightly integrated and collaborating seamlessly. By adopting this strategy, the FaaS-enabled architecture promotes a modular and scalable design that enables efficient development and management of network services. Next, a functional definition of its components and interfaces is presented.

3.1. Architecture

The modular and distributed SDN architecture utilises disaggregated software modules deployed on a serverless platform as depicted in Fig. 2.

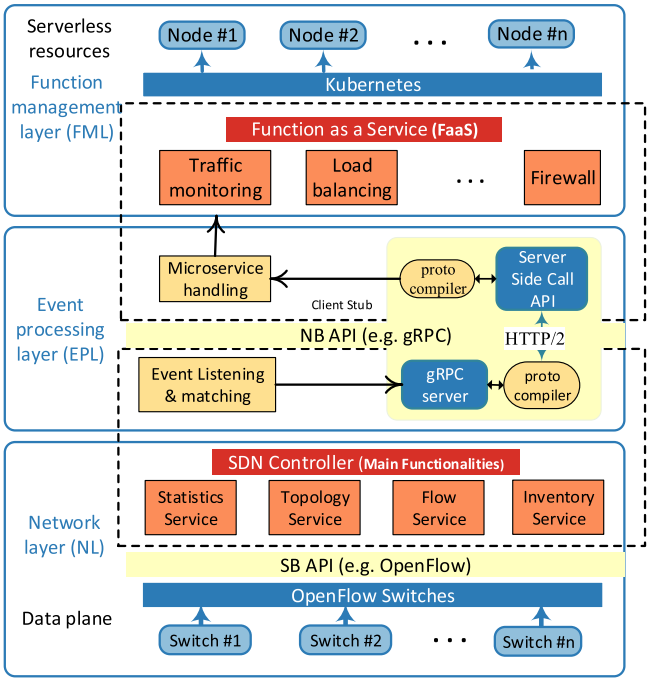


Fig. 2. SDN/NFV architecture in a serverless environment.

This model leverages the benefits of NFV to decouple networking software from the hardware that delivers it. Therefore, the software modules (i.e., management applications) can be deployed as independent microservices on the serverless platform. In this approach, SDN core services provide minimum required functionalities, and the other services can be developed as external applications in the form of collaborative software modules. Such distributed SDN/NFV architecture consists of three operational layers including the network layer (NL), event processing layer (EPL), and virtual functions management layer (VML).

3.2. Network layer

The network layer is responsible for establishing network rules and exchanging information among network elements. This layer consists of two components: (i) a data plane, i.e., OpenFlow-enabled switches, and (ii) a control plane supporting the SDN core functionalities such as topology service, flow service, inventory service, etc.

The service structure in modular SDN utilises an *event distribution* system to externalise event processing in SDN. This approach enables the division of control plane services into a set of cooperating microservices. In this method, the SDN controller operates by processing events received from both the southbound and northbound APIs. An event is defined as any changes in the network that invokes one or more management applications. Therefore, the SDN controller can be developed as an *event-driven* and *modular* software responding to these events. In Particular, the events play a crucial role in informing applications about network changes. This aim is realised by adding an *event listener module* that listens to the events coming from the underlying network. Upon receiving a new event, the network will handle invoking the corresponding functions on the serverless side. Thus, the management applications deployed on the serverless platform can be executed on demand as needed.

3.3. Event processing layer

This layer enables the externalisation of event processing in SDN by providing efficient and reliable communication among different services or components within an application or across multiple applications. The Event processing layer (EPL) is implemented as a set of independent microservices, each running in separate containers. These microservices are orchestrated using Kubernetes or other container management platforms, ensuring modular execution, dynamic scaling, and automated resource allocation. EPL consists of three main components including:

- *Event listening module* in the SDN layer is responsible for capturing OpenFlow-based events originating from the underlying network and forwarding them to the serverless connector.
- An *Event processing module* that is responsible for using a communication interface to notify SDN events to the Virtual Functions (VFs) deployed on a serverless platform. Instead of direct function-to-function calls, this module facilitates asynchronous communication via event-driven mechanisms such as gRPC-based messaging. This ensures that SDN events are efficiently routed without requiring persistent shared state.
- A *Microservices handling module* in the serverless layer that manages functions and invocations by receiving event notifications from the underlying network. Each virtual function is stateless, and interactions between functions occur through an event-driven approach rather than direct execution dependencies. When necessary, shared state management can be handled via distributed storage solutions without violating FaaS stateless execution principles.

As the main principle is effective decomposition, where network information and state remain synchronised and consistent, this layer provides an effective interaction of the functions with each other or with the event-processing module of the controller (i.e., sending and receiving the event notifications among microservices). This method enables separate component development, promoting efficient reuse across diverse deployments. Thus, it facilitates modularity, collaboration, and scalability in SDN architectures.

3.4. Function management layer

The FaaS platform allows developers to deploy functions that are triggered by specific events or requests, including packets arriving at the network layer. The functions invoked by these events can process the packets and create respective responses for them. The advantage of using FaaS for packet processing is that developers can focus on writing the logic specific to their functions, without the need to manage low-level networking infrastructure or worry about scalability. The FaaS platform handles the scaling, resource allocation, and execution of the functions based on the incoming workload.

4. Analytical model

In Section III, we outlined the details of the microservice-based network management through the modular SDN architecture. In this section, we will discuss the details of the analytical model to approximate the service delivery time and power consumption, based on the system described earlier. Our primary focus here is to obtain the steady-state metrics of the proposed system based on the system configurations and workload characteristics.

The performance model is illustrated in Fig. 3. Let us assume that the network operates in consecutive time slots $\{T_\tau | \tau = 1, 2, 3, \ldots\}$ in which a set of m functions $\mathcal{F} = \{f_1, f_2, \ldots, f_m\}$ are subscribed for events in the gateway. Similar to many other works of literature [31,32], a quasi-static scenario is adopted where the environment is considered static at each time slot but it might vary in different time slots.

After receiving a new event packet, the controller processes it and then forwards the packet toward the edge gateway. The edge gateway, in response, triggers the corresponding subscribed function on the edge computing nodes. Note that we will use packets and events interchangeably in the rest of the paper as the arriving packets from network elements are used to notify the events and changes in the network. The modularity of the system allows for deploying functions in a distributed

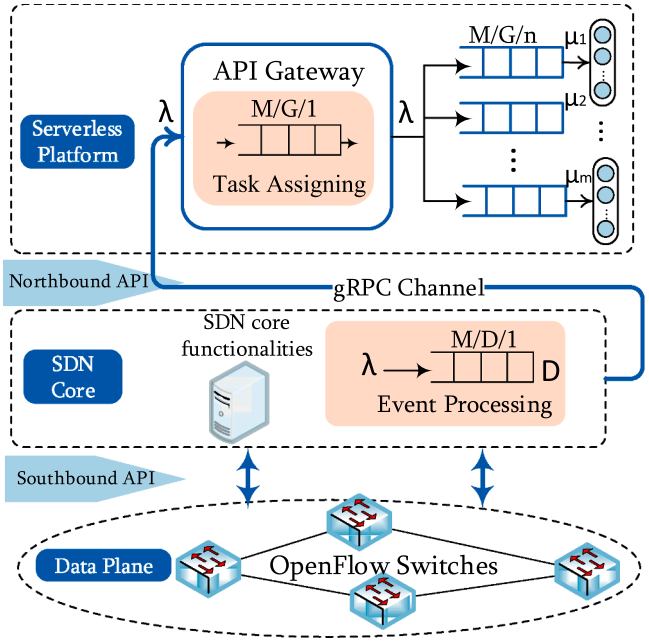


Fig. 3. System Model for distributed SDN.

environment and scaling them according to the service workload. This capability enables the initiation of multiple instances of the same service (i.e. multiple replicas of the functions), effectively addressing latency issues in microservices. We calculate steady-state estimates for various deployment characteristics based on the workload characteristics and the output model. Key notations for the most important parameters are summarised in Table 1.

4.1. Delay model

As the population size of a typical network/IoT application is relatively high, and while the probability of a given user requesting a service is relatively small, the arrival process can be modeled as a Markovian process [33]. Therefore, we assume that the packet arrival rate for each function m follows a Poisson Point Process with the average rate of λ_m, which indicates that the event inter-arrival time is exponentially distributed with a rate of $1/\lambda_m$. Accordingly, the aggregated arrival rate for all functions is also Poisson and can be denoted as

$$\lambda = \sum_{m=1}^{M} \lambda_m, \quad \forall m \in \mathcal{F} \tag{1}$$

The *event processing module* serves the arriving packets in non-preemptive First Come First Served (FCFS) discipline and directs them toward the edge gateway for more processing. The goal is to compute the latency of the service delivery on edge computing resources by receiving an event notification from the underlying network. To do so, we need to analyse the total latency of the computation and transmission entities of the network, and propose the analytical model to approximate the service delivery time and power consumption.

4.1.1. Transmission delay of controller

As discussed, the event notification packets from underlying network elements can follow the Poisson process with the aggregated arrival rate of λ at time slot τ. Meanwhile, denote μ_m as the transmission delay of the event processing module for the function m, which is deterministic and depends on the resource capacity of the controller. For simplicity, we assume the packet size ϕ_m is the same for all types of events within all functions, i.e., $\phi = \phi_m$, for each $m \in \mathcal{F}$. As ϕ_m is constant for all functions, μ_m is also constant, i.e., $\mu = \mu_m$. Hence, the transmission delay can be calculated as $\mu = \frac{\phi}{\mathcal{R}}$, where $\mathcal{R}$ is the wireless transmission rate between controller and edge gateway and can be calculated as [34]

$$\mathcal{R} = \mathcal{B} log_2(1 + \frac{\mathcal{G}w}{\sigma^2}), \tag{2}$$

Here, $\mathcal{B}$ is the bandwidth, $\mathcal{G}$ is the wireless channel gain, w is the transmitted signal power, and σ is the noise power. Given this value, the event processing of the controller can be considered as a $M/D/1$ queueing model with a deterministic service time of $D = 1/\mu$ for all functions. Thus, the average service latency of the controller (i.e., the waiting time in the queue plus the transmission time) can be derived as

$$\overline{T_S}(\tau) = \underbrace{\frac{\rho}{2\mu(1-\rho)}}_{waiting-time} + \underbrace{1/\mu}_{transmission-time} \tag{3}$$

where ρ is the offered load of the controller and can be derived as $\rho = (\sum_{m=1}^{M} \lambda_m)/\mu$.

Table 1
Parameter notation.

Parameter	Description
τ	Discrete time slots
m	Number of Functions
n	Instances of Functions
λ	Total arrival rate of events
μ	Mean service rate of events
ρ	Offered load of controller
ϕ	The packet size
$\mathcal{R}$	Wireless transmission rate
$\mathcal{B}$	Bandwidth available for channel
$\mathcal{G}$	Wireless channel gain
η	Power coefficient
x_i	binary variable indicating cold/warm start
w	transmission signal power
σ	noise power
$\overline{TS}$	Mean transmission delay of the controller
$\overline{Wg}$	Mean waiting time in the queue for gateway
$\overline{Tg}$	Mean response time of the gateway
$\overline{Wf}$	Mean waiting time for function m
$\overline{Tf}$	Average response time of function m
$\overline{T}_m$	Mean response time of the network for function m
U_s	CPU utilisation of server s
C_s	Number of cores for server s
F_s	CPU cycle frequency of server s
D_s	Cold container startup delay of server s
$Pidle$	Static power consumption of server s
$Pdyn$	Dynamic power consumption of server s
Pw	Power consumption for warm container startup
Pc	Power consumption for cold container startup
P_s	Total power consumption of server s

4.1.2. Processing delay of gateway

Algorithm 1 shows the pseudo-code of service provisioning on the serverless platform, which can be divided into three consecutive intervals for each time slot τ. Once an event is received from the controller, the gateway queries the list of functions and their interesting events, *function mapping interval*. It then maps the events into the respective function instances in the *task assigning interval*, which leads to rendering the corresponding service by invoking the functions in *execution interval*.

The edge gateway plays an important role in the service delivery process as it connects the core of the network, i.e., the SDN controller, to the external applications deployed as microservices on the serverless edge platform. This module consists of two sub-modules: a *function mapping module* which contains a list of functions and the events that they subscribed to them, and a *task assigning module* for assigning the events to the function instances and deciding about the scaling up/down of the instances according to the workload.

Hence, the processing time of the gateway can be modeled as an $M/G/1$ queuing system with the mean arrival rate of λ and independent and identically distributed (i.i.d) general service time with mean $\overline{b}$ at

Algorithm 1 Service provisioning on FaaS.

Input: Functions $\mathcal{F} = \{f_m\}$, events e
Output: $T_m(\tau)$
Initialisation :
Set values of parameters: $\lambda(\tau), \mu, m, \mathcal{I}$
Initialize $T_m(\tau), \tau$ and set $n_m = 0$
for each event e **do**
  Map e to corresponding function m
  if $n_m > 0$ **then**
    Add event to the function queue
    calculate $W_m(\tau, n_m)$ with Eq. (7)
    if $W_m \geq$ t **then**
      $n_m \leftarrow n_m + 1$ {Adding new replica for scaling up, t is a threshold for delay}
      Update $W_m(\tau, n_m)$ and $\overline{T}_m(\tau, n_m)$ with Eq. (7) and (10)
    else
      calculate $\overline{T}_m(\tau, n_m)$ with Eq. (10)
    end if
    Update parameters $\mathcal{I} = \{n_1, ..n_m\}$
  end if
end for
Update parameters τ, λ
return $\overline{T}_m(\tau, n_m)$

time τ. Thus, the total load is $\sum_{m=1}^{M} \rho_m$. The average waiting time of an event in the queue is [35]:

$$\overline{Wg}(\tau) = \frac{\lambda b^{(2)}}{2(1-\rho)}. \tag{4}$$

Here, $b^{(2)} = -\frac{d^2}{ds^2} Bg^*(0)$, where Bg^* is the Laplace Stieltjes transform (LST) of the service time for each of the functions. Consequently, the average delay for service delivery can be obtained as $\overline{Tg} = \overline{Wg} + \overline{Bg}$ where $\overline{Wg}$ is the mean waiting time in the queues and $\overline{Bg} = -\frac{d}{ds} Bg^*(0) = b$ is the average service time of edge gateway, which is denoted by

$$\overline{Tg}(\tau) = \frac{\lambda b^{(2)}}{2(1-\rho)} + b. \tag{5}$$

The average event processing time $\overline{Tp}$ of the functions at time slot τ can then be obtained by summing up $\overline{Ts}$ and $\overline{Tg}$, i.e.,

$$\overline{Tp}(\tau) = \overline{Ts}(\tau) + \overline{Tg}(\tau), \quad \forall \tau \in T_\tau \tag{6}$$

4.1.3. Processing delay of serverless nodes

The event packets served by the gateway can trigger the services, which may require invoking multiple replicas of the same function or instantiating a new instance of additional services. Let $n \in \mathcal{I} = \{n_1, n_2, \ldots, n_m\}$ denotes the number of instances for the functions independently rendering network services to the event packets. In this case, each function can be modeled as an $M/G/n$ queueing system in which the events arrive at Poisson rate λ_m and are served by any of n instances, each of whom has the service distribution G. Note that $n=0$ indicates that there is not any running instance of the function m in the system. As approximated in [35], we have

$$\overline{Wf}_m(\tau, n) = \frac{\lambda_m^n \, b_m{}^{(2)} (b_m)^{n-1}}{2(n-1)!(n-\lambda_m b_m)^2 [\sum_{i=0}^{n-1} \frac{(\lambda_m b_m)^i}{i!} + \frac{(\lambda_m b_m)^n}{(n-1)!(n-\lambda_m b_m)}]}, \tag{7}$$

where $n = n_m$ for function m, b_m and $b_m{}^{(2)}$ are calculated as $b_m = -\frac{d}{ds} B_m^*(0)$ and $b_m^{(2)} = -\frac{d^2}{ds^2} B_m^*(0)$, respectively. Let us assume that the service time of the events follows the exponential distribution with mean μ_m for each function m. Then, we have

$$\overline{Wf}_m(\tau, n) = \frac{(\rho_m)^n}{2(n-1)!\,\mu_m\,(n-\rho_m)^2 [\sum_{i=0}^{n-1} \frac{(\rho_m)^i}{i!} + \frac{(\rho_m)^n}{(n-1)!(n-\rho_m)}]}, \tag{8}$$

Note that $\rho_m = \frac{\lambda_m}{\mu_m}$ is the offered load of instances for each function. This indicates that the utilisation of each function can be calculated as the ratio of the mean number of busy instances to n_m, which is obtained by $U_m = \rho_m / n_m = \lambda_m / n_m \mu_m$. Hence, the total latency of service delivery for the instances of the function m at a serverless platform can be obtained by

$$\overline{Tf}_m(\tau, n) = \overline{Wf}(\tau, n) + b, \quad \forall n \in \mathcal{I}, \tau \in T_\tau \tag{9}$$

Accordingly, we can obtain the total latency of packet processing for the function m in this architecture as

$$\overline{T}_m(\tau, n) = \overline{Ts}(\tau) + \overline{Tg}(\tau) + \overline{Tf}_m(\tau, n) \tag{10}$$

4.2. Power model

The power consumption model for the function execution in a serverless edge computing environment can be considered in two different scenarios: a *warm container* is ready to execute the function, and a *cold start* in which a new container must be started prior to the execution [36]. Namely, *warm start* refers to reusing an existing function instance, while *cold start* involves launching a new function instance. During a cold start, the platform performs activities like initialising a new container, configuring the runtime environment, and deploying the function, which takes more time to process requests compared to warm starts. In the former case, the average energy expenditure of a function with a warm start is related to the task type and the power consumption of different subcomponents of the server, e.g. the amount of processor, memory, and disk usage for different types of tasks [37,38]. The power consumption of a running server s (i.e., physical machine (PM)) can be divided into two components: the static part and the dynamic part. $Pidle$ represents a constant value that characterises the static part when the PM is not executing any tasks. In other words, if the PM is powered on and its utilisation is nearly zero, the power consumption will be equal to $Pidle$. The dynamic power consumption can be described by a quadratic polynomial model [39]. Thus, the power consumption is approximated by:

$$Pdyn_s(\tau) = \gamma_1 U_s(\tau) + \gamma_2 U_s(\tau)^2, \quad \forall \tau \in T_\tau \tag{11}$$

where U is the CPU utilisation for the running server at time slot τ, and γ_1 and γ_2 are model parameters, the values of which are determined by linear regression during model generation. The power consumption of a running server can be modeled as

$$Pw_s(\tau) = Pidle(\tau) + Pdyn(\tau), \quad \forall \tau \in T_\tau \tag{12}$$

For the latter one, the cold container startup time should be considered along with the power for function execution. Then, the power consumption for cold starting containers at time slot τ is

$$Pc_s(\tau) = \eta F_s(\tau)^3 C_s(\tau) D_s, \quad \forall \tau \in T_\tau \tag{13}$$

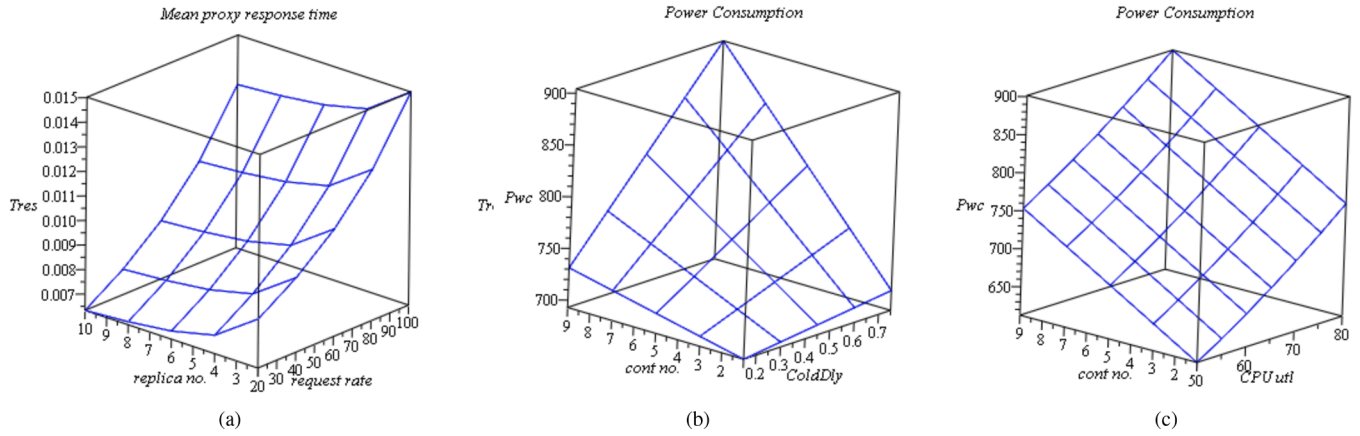
Here, η is the power coefficient, F is the CPU cycle frequency at time slot τ, C is the number of containers that require a cold start at time slot τ, and D_s denotes the cold start delay for launching an instance n on one of the server's cores [38]. Thus, the power consumption of the server at time slot τ can be expressed as:

$$P_s(\tau) = Pw_s(\tau) + \sum_{i=1}^{k} x_i(\tau) Pc_s(\tau), \quad \forall \tau \in T_\tau \tag{14}$$

where k is the maximum number of running containers on the servers, $x_i \in \{0, 1\}$ is a binary variable indicating whether the i^{th} container requires cold start at time slot τ (i.e., $x_i = 1$) or not (i.e., $x_i = 0$).

5. Analytical model evaluation

In this section, we evaluate the analytical model through mathematical analysis to provide a comprehensive understanding of its mathematical foundations, properties, behaviour, and performance. This evaluation is performed to derive theoretical insights into system behaviour.



(a) (b) (c)

Fig. 4. Performance models results (a) Mean response time with varying the number of replicas from 2 to 10. (b) Total power consumption with varying the cold start delay. (c) Total power consumption with varying the number of containers from 1 to 10.

The experimental validation of these findings is presented separately in the later evaluation section, where real-world performance is measured using the implemented architecture. The model described above has been analysed using Maple 16 from Maplesoft, Inc. [40]. Similar to previous studies [41–43], we assume that event packets arrive according to a Poisson process with a rate λ, and that service times follow an exponential distribution with a mean of μ. The arrival rate λ is varied between 20 and 100 arrivals per second. Each function instance is assigned an identical service duration, set at $(1/\mu) = 200$ milliseconds. The cold start delay, which represents the time required to launch a container on serverless edge computing nodes, is modeled within the range of $[0.15, 0.85]$ seconds. Additionally, the computational capacity of individual containers is assumed to be randomly distributed between $1Gcycle/s$ and $2Gcycle/s$, with an average capacity of $1.5Gcycle/s$. To validate and analyse the performance of our models, we consider two distinct scenarios.

In the first scenario, we analysed the mean response time of the network under varying event arrival rates and different numbers of function replicas, ranging from $n = 1$ to $n = 10$. Fig. 4 illustrates how the arrival rate of events influences the performance metrics of the model. As expected, the mean response time of the network increases as the event arrival rate rises, as shown in Fig. 4(a). Additionally, deploying more replicas of a function within the network significantly reduces the mean response time. This finding highlights the effect of modular design in mitigating delays, which is in line with what we observed in our experimental results. It is important to note, however, that the number of function replicas does not necessarily equate to the number of containers. In a serverless environment, multiple replicas may share a single container, or each replica may be hosted in a separate container depending on the platform's configuration. This distinction influences how resources are utilised and the scalability of the system. While increasing the number of replicas improves response times, this comes with the trade-off of higher resource usage, particularly in terms of power consumption.

In the second scenario, we explore the power consumption in the network with different cold container startup delays and varying numbers of containers. The analysis depicted in Fig. 4(b) reveals that increasing the number of running containers leads to higher power consumption, primarily due to the impact of cold startups. Moreover, as the cold start delay becomes longer, power consumption further increases. This relationship becomes particularly pronounced as the number of containers continues to grow.

Cold start delays are crucial in serverless computing, as they significantly affect system throughput–the rate of successfully processed requests per time unit (e.g., requests per second). For a given request rate λ and a cold start delay t_{cold}, we can approximate the throughput

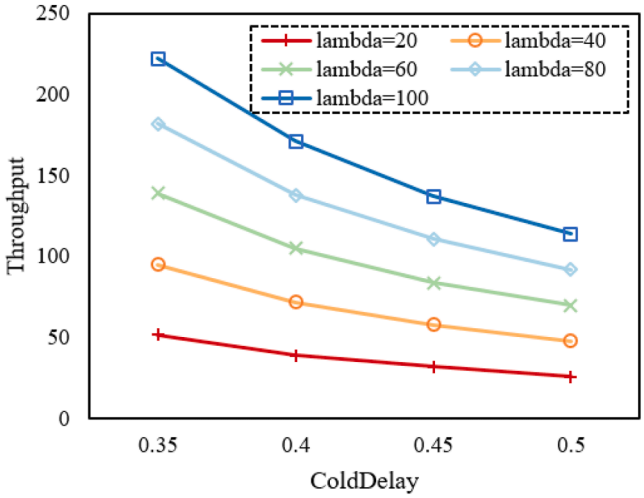


Fig. 5. The impact of cold start delay on throughput.

T as follows:

$$T = \frac{\lambda}{T_{tot} + p_{cold} * t_{cold}} \tag{15}$$

Where T_{tot} is the average time taken to handle a request, and p_{cold} is the probability of encountering a cold start. The probability of a cold start generally increases when the rate of incoming requests exceeds the number of available warm containers or instances. We can define p_{cold} as $p_{cold} = \frac{(\lambda - n)}{\lambda}$, where n is the number of concurrently available active containers.

This formulation allows us to quantify the trade-off between increased latency due to cold starts and its effect on overall throughput. As seen in Fig. 5, higher cold start delays directly reduce throughput, especially at elevated arrival rates where cold starts are more frequent. The graph indicates that as cold start delay increases, throughput declines sharply, particularly at higher request rates. The steep decline in throughput at elevated request rates underscores the importance of optimising cold start handling to maintain efficiency and responsiveness. This performance impact calls for dedicated research for developing strategies to minimise cold start delays in serverless environments.

In the next scenario, we kept the cold start delay constant while varying CPU utilisation to examine the impact of physical machine (PM) utilisation on power consumption, as shown in Fig. 4(c). Our findings

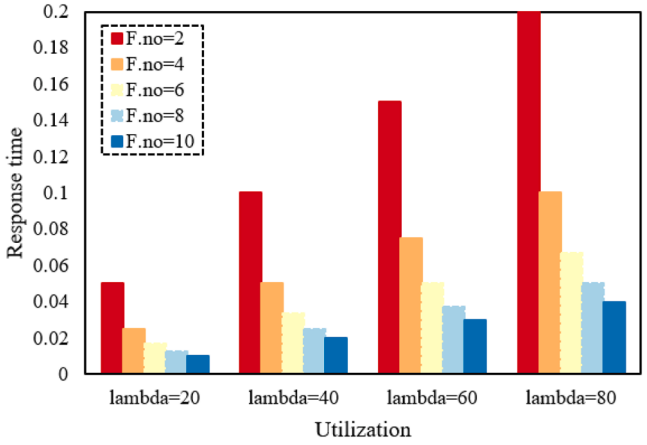


Fig. 6. Stability of the system.

confirmed that power consumption increases as server load rises. This effect becomes more pronounced when a higher number of active containers are present, primarily because of the additional power required during cold starts.

To assess system stability, we also analyse the relationship between utilisation and service delay, as shown in Fig. 6 This analysis examines how varying arrival and service rates affect response times, with different numbers of active functions ("F.no"). The graph shows that as the arrival rate increases, response time also rises, indicating the growing load of the system. At high arrival rates, performance begins to degrade significantly, eventually leading to unbounded delays. This pattern indicates potential congestion and a loss of system stability, as high response times make the system less responsive to new tasks.

To counter this, the system adapts by increasing the service rate, effectively adding more active functions to handle the load. This adjustment results in a reduction in delay, confirming that increased capacity improves response times under high load. However, the impact of adding functions diminishes beyond a certain point. As more instance functions are added, the gateway has to handle more event processing, which increases its workload and slows down response times. Cold start delays also become a problem when too many instances are launched at once, reducing the benefits of scaling. Additionally, uneven resource use and poor load balancing can cause some functions to be overloaded while others remain idle, leading to further inefficiencies. As can be seen, after six active functions, the benefit to delay reduction becomes minimal with all arrival rates. This result implies that while scaling up functions initially mitigates delay effectively, simply adding more functions beyond an optimal threshold does not significantly enhance performance. Therefore, the system should balance between adding resources and maintaining efficiency to optimise both performance and resource usage.

The results in this section are derived from an analytical model, providing a theoretical understanding of system performance under various conditions. However, to validate the accuracy and applicability of these analytical findings, we conduct an experimental evaluation in the next section following the implementation of the architecture.

6. Modular SDN prototype implementation

In this section, we introduce the software components and technologies employed in the implementation of the serverless SDN/NFV architecture. Our implementation incorporates the integration of three widely recognised open-source projects:

1) *Open Network Operating System (ONOS)* [6], an SDN controller whose architectural design follows a layered structure, including the application layer, core layer, and providers/protocols layer;

2) *general-purpose Remote Procedure Call (gRPC)* [13], an RPC system that utilises a protocol buffer and HTTP/2 as its underlying transport protocol and serves as the connection mechanism between microservices;

3) *Functions as a Service (OpenFaaS)* [12], a framework for building event-driven serverless functions on top of containers (with Docker and Kubernetes) and supporting the different architectural components as shown in Fig. 7(a).

In ONOS, we utilise the Northbound Interfaces (NBI) in the *application layer* to receive updates on network events and states. The *core part* of ONOS manages network states, maintains an inventory of connected devices, hosts, and links, and provides an overview of the network topology. Additionally, it handles the installation and management of rules in network devices. In the *provider/protocol* layer, the Southbound Interface (SBI) encompasses a set of plugins consisting of a provider interface that integrates protocol-specific libraries and a service interface. SBI therefore facilitates interaction with the network environment using different control and configuration protocols.

We leverage *Mininet* environment [44] to emulate the underlying network infrastructure, including switches and hosts, as well as generate network packets from the data plane.

In order to inform applications about network changes and externalise event processing in ONOS [19], we added an event processing layer to enable the division of control plane services into a set of cooperating microservices, see Fig. 7(b). This approach allows core services to

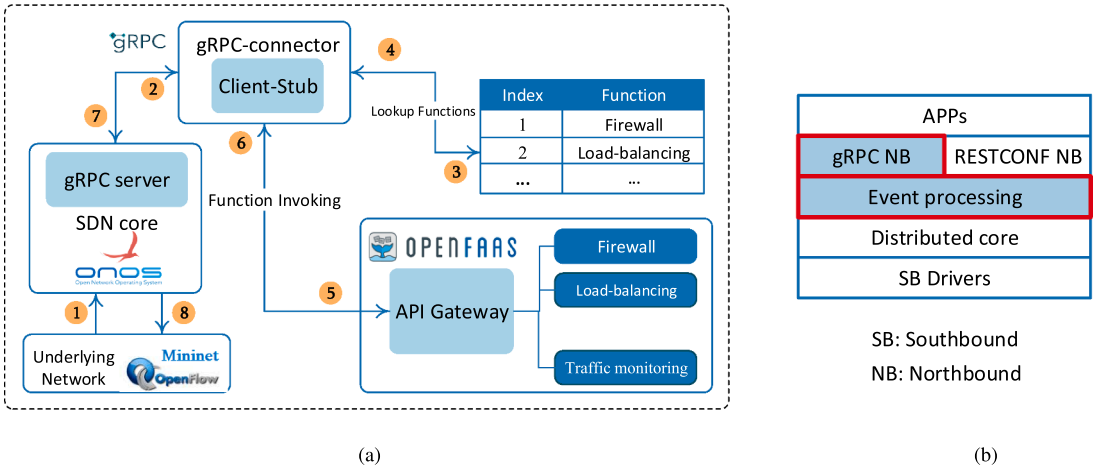


Fig. 7. Implementing the microservice-based network architecture (a) ONOS and OpenFaaS integration sequence diagram. (b) Protocol stack of ONOS.

Table 2
Latency measurement: configuration of azure servers.

Property	Value	Property	Value
vCPU	4	RAM	16GB
Network	1000 Mb/s	Ubuntu	20.04 LTS
Kubernetes	1.18	Docker	19.2
ONOS	2.7.0	OpenFaaS	v0.5
Java	11	Python	3.5

deliver fundamental functionalities, while supplementary features can be integrated using microservices within the OpenFaaS platform.

We have developed a gRPC server as an ONOS application, leveraging gRPC features such as security, authentication mechanisms and bidirectional streaming. Additionally, gRPC enables automatic code generation for both the server and client sides. The gRPC server interacts with ONOS services, converts the returned values into protobuf format and sends them to the client after serialisation. On the other hand, a gRPC-connector is developed as an external application that acts as the client stub and establishes a connection between ONOS services and OpenFaaS Functions. The latter are invoked by various types of events, including HTTP, which allows for seamless system integration, and made accessible through HTTP endpoints via the Gateway service.

Upon receiving an event from ONOS, the gRPC-connector initiates a query to the Gateway service in order to retrieve the list of available functions. It then creates a mapping between the event topic and the corresponding functions. This mapping enables the triggering of the appropriate function based on the topic associated with the received event.

7. Modular SDN prototype evaluation

To evaluate the performance of the modular SDN/NFV architecture, the deployment of its prototype is described next, considering latency and power measurement as key metrics.

7.1. Latency

To evaluate the performance of the modular SDN/NFV architecture, the prototype of the proposed architecture has been deployed on three Microsoft Azure Virtual Machines (VM), each equipped with 4 VCPUs and 16GB memory, see Table 2 for details. The first VM hosts ONOS with a gRPC server installed on it, along with Mininet which emulates a network with a number of OpenFlow-enabled switches and hosts. The second VM runs OpenFaaS as a resource pool and network functions host. The third VM has Docker containers deployed on Kubernetes as a resource pool and performs the same tasks as the OpenFaaS functions. These servers run Ubuntu 20.04 LTS and are connected via a 1Gbps LAN network.

The objective of the evaluation is to first compare the latency of the proposed distributed architecture implementation with a monolithic one that hosts local applications. A host sends a continuous flow of IP packets, which generates the event notification in the controller, and accordingly leads a function to be invoked on OpenFaaS. In this experiment, ONOS communication with VNFs is handled via gRPC protocol and the OpenFaaS orchestrates the function requests by an automatic scaling process. We have conducted the experimental evaluation of our model through a diverse set of serverless workloads. To do so, the traffic load is increased during experiments to observe the behaviour of both models in higher traffic arrivals. Fig. 8(a) illustrates the effect of number of packets per seconds on mean response time recorded during the experiment. Each observation shows the latency in both the control plane and the VNFs layer.

As can be seen, the mean response times of the local functions are lower than serverless-based functions at the start of the experiment. The reason for this is the latency imposed by grpc channel to invoke the remote functions on OpenFaas. Furthermore, both applications (i.e. local

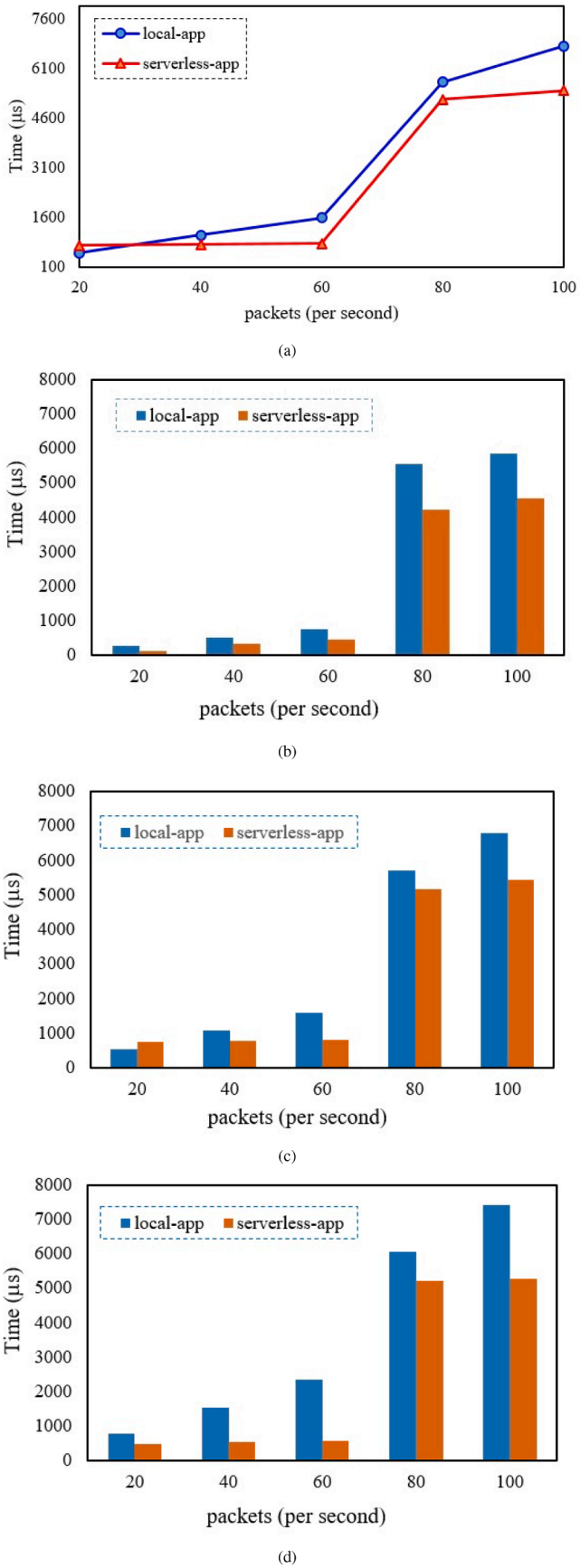


Fig. 8. The Average delay in: (a) different arrival rates. (b) service rate $= \frac{1}{b} = 25\mu s$. (c)service rate $\frac{1}{b} = 34\mu s$. (d)service rate $\frac{1}{b} = 42\mu s$.

Table 3
Power measurement: machine configuration.

Property	Value	Property	Value
CPU	AMD 5800H	RAM	16GB
Network	1000 Mb/s	Ubuntu	24.04 LTS
Kubernetes	1.18	Docker	19.2
ONOS	2.7.0	OpenFaaS	v0.5
Java	11	Python	3.5

and remote functions) experience higher amount of response times with increasing the arrival rate of packets on the data plane, because it imposes higher processing load in the network. However, it is observed that the serverless-based approach outperforms the local method in higher packet arrivals, as it can automatically scale and manage the resources according to the work load.

In the second scenario, we have considered the latency in different service rates of the functions, which is due to the processing of the different packet sizes. Fig. 8(b) compares the results in mean processing time of $\frac{1}{b} = 25\mu s$ for both local and serverless-based applications. As observed, the mean delay of both applications shows an up trend. However, it is lower for serverless-based application than the local one with the results for local functions showing a steep slop. Furthermore, with increasing the processing time of the function in Fig. 8(c) to $\frac{1}{b} = 34\mu s$, local functions win at low packet arrivals, but experience higher amounts of delay with increasing the arrival rate. As discussed previously, the reason is due to the scalability of serverless platform, which can also be seen from Fig. 8(d) in $\frac{1}{b} = 42\mu s$. Intuitively, the higher the workload in each experiment is, the more important the ability of scaling becomes. This results can clearly show the benefits of serverless-based method in latency handling and service scaling.

7.2. Power measurement – single machine deployment

To accurately measure power consumption, the prototype of the proposed architecture has been deployed on a single machine equipped with an AMD Ryzen 5800H CPU and 16GB memory running Ubuntu 24.04 LTS, see Table 3 for details. This was motivated by the difficulty measuring power consumption in virtualised environments, which is due to: 1) lack of direct hardware access: VMs run on shared physical hardware, abstracted by the hypervisor, which prevents VMs from directly accessing hardware power sensors, and 2) resource sharing and overhead: VMs share CPU, memory, and disk resources dynamically, and power usage fluctuates based on workloads, making it hard to attribute exact power consumption to individual VMs.

As in the previous experiment, ONOS and the gRPC server are deployed, along with Mininet which emulates a network with a number of OpenFlow-enabled switches and hosts. OpenFaaS is considered as a resource pool and network functions host, whereas Docker containers are deployed on Kubernetes as a resource pool and perform the same tasks as the OpenFaaS functions do. The Powerstat tool [45] is used to collect the power consumption on the machine.

The objective of the evaluation of the microservice-based SDN architecture is to compare the power / energy consumption and processing time of the serverless architecture with Docker containers in the context of processing the events generated from ONOS. In such a scenario, once a host sends a packet/event, the switch directs the incoming packet to the SDN controller. The event listening module of the controller serves the packet by sending it to the external application deployed on either the serverless platform or a Docker container via a gRPC channel. ONOS communication with VNFs is handled via gRPC protocol, while OpenFaaS and Kubernetes orchestrate the coming requests by an automatic scaling process.

Results. The experimental scenario involves simulating a network of 30 virtual OpenFlow-enabled switches and generating a workload using Mininet commands. This scenario results in a generation of approximately 10 thousand packets that are forwarded as requests to OpenFaaS or Docker containers. As shown in Fig. 9, three distinct power consumption peaks are observed during the execution of the workload under OpenFaaS and Docker containers, respectively. These peaks show the experiment being repeated three times so that the impact of cold start can be measured, while considering their short execution durations and on-demand triggering [46].

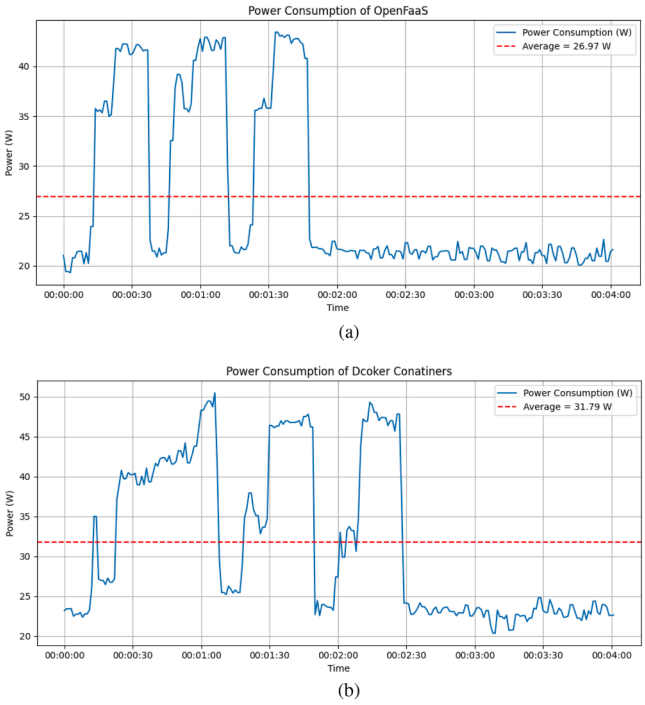


Fig. 9. Power consumption of (a) OpenFaaS deployment, (b) Docker deployment.

During the first 120 seconds of the experiment, both OpenFaaS functions and Docker containers were scaled out to process the increasing number of incoming packets. OpenFaaS exhibited a power consumption of approximately 45 Watts while handling incoming requests; see Fig. 9(a), while Docker containers reached a peak of 52 Watts over the same time interval; see Fig. 9(b). OpenFaaS significantly consumed less power than Docker containers when scaled out, which emphasises the ability of serverless functions to handle the sudden changes in traffic more efficiently than Docker containers. For the overall experiment, Fig. 9(a)) shows an average power consumption of 27 Watts for OpenFaaS, while Docker containers consumed an average of 32 Watts, as shown in Fig. 9(b)). These results clearly demonstrate that OpenFaaS is more power efficient compared to Docker containers, resulting in power saving.

Table 4 provides more insight into this experiment, presenting the energy consumption, processing time and the number of replicas. As observed in the table, despite both platforms utilising 5 replicas, OpenFaaS completes the experiment earlier than Docker and shows an improvement of 58 % in processing time. Moreover, Docker consumed 40 % more energy, indicating a significant improvement in the energy consumption of the proposed architecture.

Table 4
OpenFaaS vs. docker comparison – metrics.

Metrics	Docker	OpenFaaS
Energy (Joule)	2180.2	1560.5
Processing Time (sec)	117	74
No. of Replicas	5	5

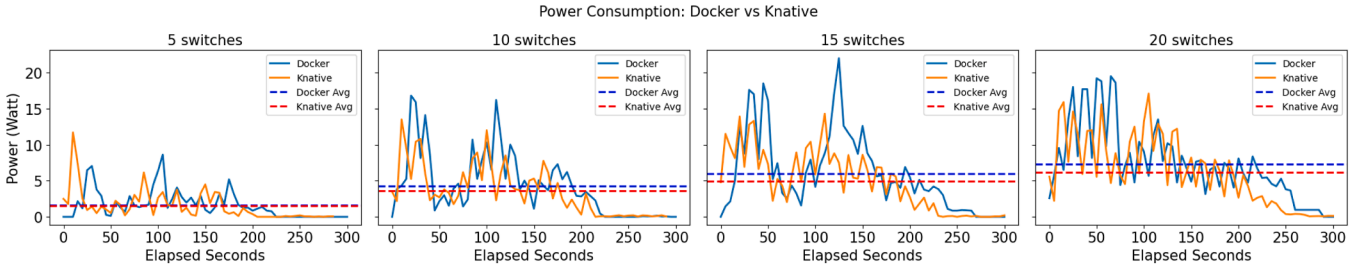


Fig. 10. Comparison of Power Consumption: Docker vs Knative.

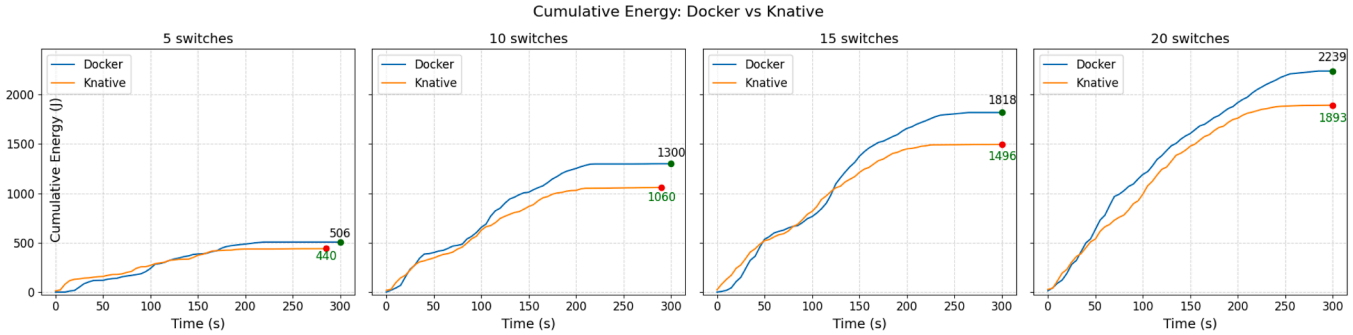


Fig. 11. Comparison of Cumulative Energy Consumption: Docker vs Knative.

7.3. Power measurement – two machine deployment

This section presents an additional experiment conducted using two machines to reflect a real-world scenario. The power measurements in a controlled single-machine environment, while necessary for accuracy, limit the applicability to distributed cloud deployments where network latency, resource contention, and heterogeneous hardware configurations could substantially impact both performance and energy efficiency. Firstly, power consumption measurements for a specific container or application can be significantly affected by other concurrently running processes. To reduce such interference, the proposed architecture was deployed across two machines. Secondly, while OpenFaaS was used in the previous experiment as the serverless platform, for generalisability Knative is selected in this study due to its broader set of features and its fully open-source and cost-free nature [47]. Thirdly, to investigate the scalability of the deployed containers and services under varying traffic loads, the experiment included four different scenarios corresponding to network configurations with 5, 10, 15, and 20 switches.

To measure power consumption, this experiment employed Kepler–a tool deployed on Kubernetes clusters–to monitor the power usage of individual Pods [48]. Kepler offers more precise and accurate measurements by leveraging eBPF technology, which enables direct access to Linux kernel features for tracking power usage at both the Pod and Node levels. By default, Kepler reports average power consumption in Watts over a 30-second interval; however, in this experiment, the interval was reduced to 5 seconds to enhance measurement accuracy. This experiment also incorporated real data traffic processed by both Docker containers and Knative services. The traffic was transmitted through four distinct network configurations comprising 5, 10, 15, and 20 switches, respectively.

The first machine, described in Table 3 hosted a Kubernetes cluster running a Knative service with scale-to-zero capability, Kepler, and a Docker container configured for scaling via the Horizontal Pod Autoscaler (HPA). The second machine, equipped with an Intel i5-10210U CPU and 16GB of RAM running RedHat 9, hosted ONOS, which controlled a virtual network created using Mininet. Both machines are directly connected using a 10 Gbps connection.

Table 5
Docker vs. knative docker comparison: number of replicas.

Switches	Docker	Knative
5	2	3
10	5	4
15	7	6
20	12	11

Both the Docker container and the Knative service executed a Python application designed to inspect each incoming packet for malicious patterns using regular expressions, and to check IP packets and ports against a blacklist. The data traffic consisted of selected flows extracted from the InSDN dataset [49] for evaluating Intrusion Detection Systems (IDS) in SDNs. For the network configuration, each switch was connected to a distinct host responsible for generating a single traffic flow. Thus, in the 5-switch scenario, ONOS received five data flows, with the number of flows increasing proportionally with the number of switches. This traffic scaling was intended to impose progressively higher loads on the container and Knative engine, allowing a robust assessment of their performance and scalability.

Results. The experiment generated four sets of power consumption measurements, corresponding to the four network configurations. Across all scenarios, Knative consistently demonstrated lower power consumption compared to Docker, despite both platforms deploying a similar number of replicas to process the increase in packet load (Table 5). Each experiment was executed for 300 seconds (5 minutes), a duration deemed sufficient to process all incoming packets. This interval also allowed for the inclusion of power consumption associated with replica scaling down and termination on both platforms. Fig. 10 presents the power consumption for Docker and Knative across the four experiments, which indicates that Knative generally consumes less power than Docker during both packet processing and replica termination, the latter being apparent towards the end of each experiment. Consequently, Knative achieved lower average power consumption, with the difference becoming more pronounced in the 15- and 20-switch configurations.

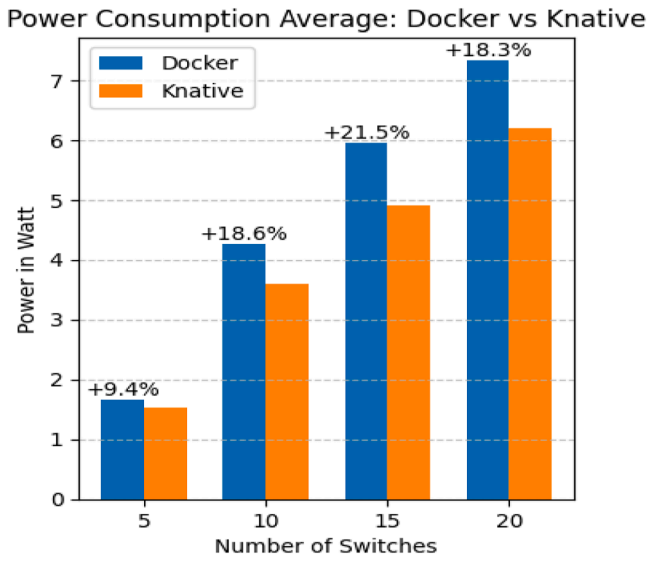


Fig. 12. Comparison of Average Power Consumption.

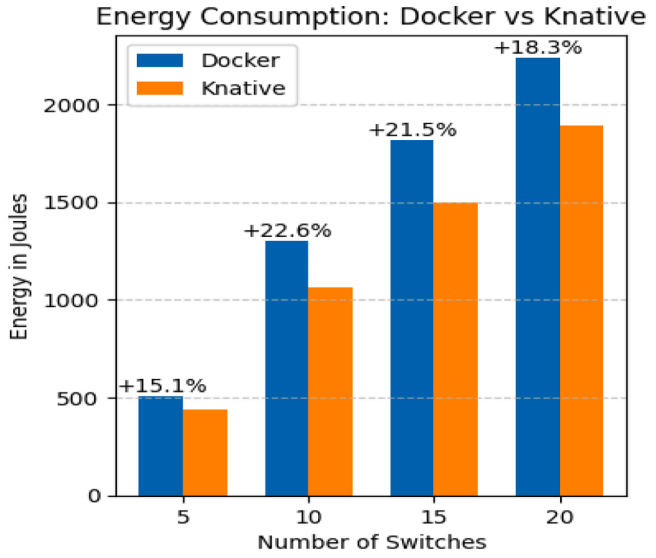


Fig. 13. Comparison of Energy Consumption.

Fig. 12 depicts the differences in average power consumption over the fixed 5-second interval, with Docker consuming more power in all four cases. Fig. 13 illustrates the differences in energy consumption between the two platforms, showing a notable increase in consumption by Docker across all four scenarios, exceeding 20 %, which highlights the efficiency improvement achieved by Knative. Furthermore, Fig. 11 illustrates the cumulative energy consumption over time, showing that both platforms initially exhibit similar energy usage. However, the divergence between them increases with time, mainly due to the higher energy expenditure associated with the creation and termination of replicas by Docker.

8. Conclusion

This paper introduces a microservice-based SDN architecture designed to provide an automatic and scalable network management solution. Using serverless edge computing, the architecture enables the distribution of SDN services, facilitating the seamless integration of new services into the network. We present an analytical model to approximate service delivery time and power consumption, and validate the proposed architecture through a prototype implementation. The evaluation results demonstrate that the architecture effectively addresses concerns about energy consumption in SDN, achieving a nearly 50 % improvement in energy efficiency compared to existing platforms such as Docker. Moreover, the serverless paradigm can decrease service latency for such a disaggregated architecture, and provides on-demand, scalable, and efficient resource management.

The architecture aims to provide fast and cost-efficient services for next-generation industrial networks. By enabling the on-demand creation of virtual networks on a shared hardware platform, the model supports faster time-to-market for new services, reducing both capital and operational costs, particularly for short-lived services. By integrating cutting-edge technologies and dynamically adapting to evolving network demands, the system contributes to the building of sustainable, energy-efficient network environments.

Future work will investigate extensive network topologies, with the aim to validate the proposed approach in an actual cloud environment to demonstrate that the energy benefits persist under realistic data center conditions with distributed infrastructure and production workloads. Moreover, it will explore the integration of energy-efficient and flexible solutions for task scheduling within serverless environments. By incorporating machine learning models, we aim to optimise resource management and further reduce energy consumption through intelligent resource configuration. This will improve the adaptability and efficiency of the architecture, paving the way for more advanced data-driven decision making in SDN environments.

CRediT authorship contribution statement

Fatemeh Banaie: Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Formal analysis, Conceptualization; **Karim Djemame:** Writing – review & editing, Writing – original draft, Validation, Supervision, Resources, Project administration, Methodology, Investigation, Funding acquisition, Conceptualization; **Abdulaziz Alhindi:** Writing – review & editing, Writing – original draft, Validation, Software, Methodology; **Vasilios Kelefouras:** Writing – review & editing, Writing – original draft, Validation, Resources, Methodology.

Data availability

Data will be made available on request.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Karim Djemame reports financial support was provided by European Commission. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

The authors would like to thank the European Next Generation Internet Program for Open INTErnet Renovation (NGI-Pointer 2) for supporting this work under contract 871528 (EDGENESS Project).

References

[1] M. AL-Makhlafi, H. Gu, A. Almuaalemi, E. Almekhlafi, M.M. Adam, RibsNet: a scalable high-performance, and cost-effective two-layer based cloud data center network architecture, IEEE Trans. Netw. Serv. Manag. 20 (2) (2023) 1676–1690.
[2] F. Bannour, S. Souihi, A. Mellouk, Distributed SDN control: survey, taxonomy and challenges, IEEE Commun. Surv. Tutor. 20 (1) (2018) 333–354.
[3] H. Balakrishnan, et al., Revitalizing the public internet by making it extensible, ACM SIGCOMM Comput. Commun. Rev. 51 (2021) 2.

[4] J. McCauley, Y. Harchol, A. Panda, B. Raghavan, S. Shenker, Enabling a permanent revolution in internet architecture, in: Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM'19, ACM, Beijing, China, 2019, p. 1–14.

[5] C. Bektas, S. Monhof, F. Kurtz, C. Wietfeld, Towards 5G: an empirical evaluation of software-defined end-to-end network slicing, in: 2018 IEEE Globecom Workshops (GC Workshops), 2018, pp. 1–6.

[6] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, G. Parulkar, ONOS: towards an open, distributed SDN OS, in: Proceedings of the Third Workshop on Hot Topics in Software Defined Networking, HotSDN'14, ACM, Chicago, USA, 2014, p. 1–6.

[7] J. Medved, R. Varga, A. Tkacik, K. Gray, OpenDaylight: towards a model-driven SDN controller architecture, in: Proceedings of IEEE International Symposium on a World of Wireless, IEEE, Sydney, Australia, 2014, pp. 1–6.

[8] K. Djemame, Energy efficiency in edge environments: a serverless computing approach, in: K. Tserpes, J. Altmann, J.Á. Bañares, O. Agmon Ben-Yehuda, K. Djemame, V. Stankovski, B. Tuffin (Eds.), Economics of Grids, Clouds, Systems, and Services, Springer, Cham, 2021, pp. 181–184. Lecture Notes in Computer Science 13072.

[9] Y. Li, Y. Lin, Y. Wang, K. Ye, C. Xu, Serverless computing: state-of-the-art, challenges and opportunities, IEEE Trans. Serv. Comput. 16 (2) (2023) 1522–1539.

[10] P. Aditya, I.E. Akkus, A. Beck, R. Chen, V. Hilt, I. Rimac, K. Satzke, M. Stein, Will serverless computing revolutionize NFV?, Proc. IEEE 107 (4) (2019) 667–678.

[11] F. Banaie Heravan, K. Djemame, A serverless computing platform for software defined networks, in: K. Tserpes, J. Altmann, J.A. Bañares, O. Agmon Ben-Yehuda, K. Djemame, V. Stankovski (Eds.), Economics of Grids, Clouds, Systems, and Services, Springer, Izola, Slovenia, 2022, pp. 113–123. Lecture Notes in Computer Science 13430.

[12] OpenFaaS Ltd., OpenFaaS - serverless functions, made simple, 2024. https://openfaas.com/.

[13] grpc, A high-performance, open-source, general-purpose RPC framework. [online]. Available:, 2024. https://github.com/grpc.

[14] D. Merkel, Docker: lightweight Linux containers for consistent development and deployment, Linux J. 2014 (239) (2014).

[15] S. Rout, K.S. Sahoo, S.S. Patra, B. Sahoo, D. Puthal, Energy efficiency in software defined networking: a survey, SN Comput. Sci. 2 (2021) 4.

[16] I. Maity, R. Dhiman, S. Misra, EnPlace: energy-aware network partitioning for controller placement in SDN, IEEE Trans. Green Commun. Netw. 7 (1) (2023) 183–193. https://doi.org/10.1109/TGCN.2022.3175901

[17] T.F. Oliveira, S. Xavier-de Souza, L.F. Silveira, Improving energy efficiency on SDN control-plane using multi-core controllers, Energies 14 (11) (2021). https://doi.org/10.3390/en14113161

[18] M. Priyadarsini, S. Kumar, P. Bera, An energy-efficient load distribution framework for SDN controllers, Computing 102 (2020) 2073–2098. https://doi.org/10.1007/s00607-019-00751-2

[19] D. Cormer, A. Rastegarnia, Toward dissagregating the SDN control plane, IEEE Commun. Mag. 57 (10) (2019) 70–75.

[20] C. Cicconetti, M. Conti, A. Passarella, A decentralized framework for serverless edge computing in the internet of things, IEEE Trans. Netw. Serv. Manag. 18 (2) (2021) 2166–2180.

[21] J. Shen, H. Yu, Z. Zheng, C. Sun, M. Xu, J. Wang, Serpens: a high-performance serverless platform for NFV, in: 2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS), 2020, pp. 1–10.

[22] A. Tzenetopoulos, C. Marantos, G. Gavrielides, S. Xydis, D. Soudris, FADE: FaaS-inspired application decomposition and energy-aware function placement on the edge, in: Proceedings of the 24th International Workshop on Software and Compilers for Embedded Systems, SCOPES'21, ACM, Eindhoven, Netherlands, 2021, p. 7–10.

[23] A. Agiollo, P. Bellavista, M. Mendula, A. Omicini, EneA-FL: energy-aware orchestration for serverless federated learning, Future Gener. Comput. Syst. 154 (C) (2024) 219–234. https://doi.org/10.1016/j.future.2024.01.007

[24] J. Stojkovic, N. Iliakopoulou, T. Xu, H. Franke, J. Torrellas, EcoFaaS: rethinking the design of serverless environments for energy efficiency, in: 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA), 2024, pp. 471–486. https://doi.org/10.1109/ISCA59077.2024.00042

[25] A. Alhindi, K. Djemame, F. Banaie, On the power consumption of serverless functions: an evaluation of openFaaS, in: Proceedings of the 15th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2022), Vancouver, IEEE, 2022, pp. 366–371. https://doi.org/10.1109/UCC56403.2022.00064

[26] K. Authors, Kubernetes documentation, 2024, (????). https://kubernetes.io/docs/home/.

[27] R. Chiorescu, K. Djemame, Scheduling energy-aware multi-function serverless workloads in OpenFaaS, in: Proc. 20th Conference on the Economics of Grids, Clouds, Software, and Services (GECON'2024), Springer, Rome, Italy, 2024. Lecture Notes in Computer Science 15358.

[28] A. Alhindi, K. Djemame, SDN traffic flows in a serverless environment: a categorization of energy consumption, in: Proc. of the 17th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2024), Sharjah, UAE, 2024. To appear.

[29] M. Falkner, A. Leivadeas, I. Lambadaris, G. Kesidis, Performance analysis of virtualized network functions on virtualized systems architectures, in: Proc. of the 21st IEEE International Workshop on Computer Aided Modelling and Design of Communication Links and Networks (CAMAD), IEEE, 2016, pp. 71–76.

[30] S.T. Arzo, D. Scotece, R. Bassoli, D. Barattini, F. Granelli, L. Foschini, F.H.P. Fitzek, MSN: a playground framework for design and evaluation of MicroServices-Based sdN controller, J. Netw. Syst. Manag. 30 (2022).

[31] M. Tang, V.W.S. Wong, Deep reinforcement learning for task offloading in mobile edge computing systems, IEEE Trans. Mobile Comput. 21 (2020) 1985–1997.

[32] Q. Tang, R. Xie, F.R. Yu, T. Chen, R. Zhang, T. Huang, Y. Liu, Distributed task scheduling in serverless edge computing networks for the internet of things: a learning approach, IEEE Internet Things J. (2022) 19634–19648.

[33] G. Grimmett, D. Stirzaker, Probability and random processes, Univ. Press, Oxford, third edition, Oxford, 2010.

[34] O. Rioul, J. Magossi, On Shannon's formula and Hartley's rule: beyond the mathematical coincidence, Entropy 16 (9) (2014) 4892–4910.

[35] M.S. Ross, Introduction to probability models, Academic Press, 13 edition, 2023.

[36] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, V. Sukhomlinov, Agile cold starts for scalable serverless, in: 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19), USENIX Association, Renton, WA, 2019, p. 21.

[37] Z. Zhou, M. Shojafar, M. Alazab, Iecl: an intelligent energy consumption model for cloud manufacturing, IEEE Trans. Ind. Inform. 18 (12) (2022) 8967–8976.

[38] H. Zhao, J.W. F. Liu, Q. Wang, W. Zhang, Q. Zheng, Power-aware and performance-guaranteed virtual machine placement in the cloud, IEEE Trans. Parallel Distrib. Syst. 29 (2018) 6.

[39] M. Aldossary, K. Djemame, I. Alzamil, A. Kostopoulos, A. Dimakis, E. Agiatzidou, Energy-aware cost prediction and pricing of virtual machines in cloud computing environments, Future Gener. Comput. Syst. 93 (2019) 442–459.

[40] Maplesoft, Maple, 2024. http://www.maplesoft.com/products/maple.

[41] M.S. Markus, et al., TppFaaS: modeling serverless functions invocations via temporal point processes, IEEE Access 10 (2022) 9059–9084.

[42] N. Mahmoudi, H. Khazaei, Performance modeling of serverless computing platforms, IEEE Trans. Cloud Comput. 10 (2022) 4.

[43] A. Sriraman, T.F. Wenisch, Mu suite: a benchmark suite for microservices, Proc. IEEE Int. Symp. Workload Characterization 1–12 (2018).

[44] Mininet Project, Mininet, 2024. http://mininet.org/.

[45] C.I. King, Powerstat, 2024. https://github.com/ColinIanKing/powerstat.

[46] J. Wen, Z. Chen, J. Zhao, F. Sarro, H. Ping, Y. Zhang, S. Wang, X. Liu, SCOPE: performance testing for serverless computing, ACM Trans. Softw. Eng. Methodol. (2025). https://doi.org/10.1145/3717609

[47] Knative Project, Knative. providing the building blocks for creating modern, cloud-based applications, 2025. https://knative.dev/docs/.

[48] Kepler Project, Kubernetes efficient power level exporter (Kepler), 2025. https://sustainable-computing.io/.

[49] M.S. Elsayed, N.-A. Le-Khac, A.D. Jurcut, InSDN: a novel SDN intrusion dataset, IEEE Access 8 (2020) 165263–165284. https://doi.org/10.1109/ACCESS.2020.3022633