Latency-Sensitive Edge/Cloud Serverless Dynamic Deployment Over Telemetry-Based Packet-Optical Network

István Pelle, Francesco Paolucci, Balázs Sonkoly, and Filippo Cugini

***Abstract*— The serverless technology, introduced for data center operation, represents an attractive technology for latency-sensitive applications operated at the edge, enabling a resource-aware deployment accounting for limited edge computing resources or end-to-end network congestion to the cloud. This paper presents and validates a framework for automated deployment and dynamic reconfiguration of serverless functions at either the edge or cloud. The framework relies on extensive telemetry data retrieved from both the computing and packet-optical network infrastructure and operates on diverse Amazon Web Services technologies, including Greengrass on the edge. Experimental demonstration with a latency-sensitive serverless application is then provided, showing fast dynamic reconfiguration capabilities, e.g., enabling even zero outage time under certain conditions.**

***Index Terms*— Serverless, FaaS, Function as a Service, cloud, edge, IoT, AWS, Lambda, Greengrass, 5G, P4, telemetry.**

I. Introduction

EMERGING computationally intensive and latency-sensitive 5G applications are driving the evolution of both networking and computing technologies. Latency-sensitive data have to be elaborated in close proximity to where they are produced, making inadequate the traditional approach of fully delegating all computations to remote cloud resources. However, the availability of computing and networking resources at the edge is limited for cost reasons. Moreover, such resources have to be shared among multiple competing and dynamic 5G applications. An example of resource-hungry applications is represented by machine learning inferences, such as video analytics, offloaded from end terminals but often requiring near real-time capabilities in object/event recognition.

At the network side, the optical 5G converged fronthaul and backhaul network architecture is evolving to support high data plane programmability and accurate monitoring capabilities. In particular, the P4 [1] technology has emerged as one of the most relevant candidates to provide network programmability over packet-optical infrastructures. The P4 (as a short form for Programming Protocol-Independent Packet Processors [2]) language and runtime provide high level direct access for configuring the behavior of the data plane of packet forwarding elements in general (not selectively for optical instruments). P4 can enable the programmability of elements like hardware and software switches, network interface cards, or other network appliances. In the optical domain, [3] demonstrates a P4-defined data path for L2/L3 transport for sliceable optical transport, while [4] showcases monitoring and capturing latency performance of applications by using P4 in-band telemetry. In addition, P4 telemetry has been exploited to monitor network resources with higher accuracy at limited bandwidth compared to traditional approaches [5].

At the computing side, Function as a Service (FaaS), a specific realization of the serverless concept, has emerged to improve deployment flexibility. In serverless, traditional monolithic applications are engineered as a combination of (sub)functions to be deployed and combined or chained in a flexible way. FaaS makes the process of creating such applications easier for developers by lifting the burden of managing compute resources and networking. Execution platforms provide a *fixed runtime environment* where user code can be executed automatically. In a usual setting, upon invocation of a function (for which *diverse built-in invocation methods are available*), an instance in idle state is reused for handling the request leveraging *built-in load balancing* options for instance selection. When there are no idle instances, a new one is created. This *built-in automatic scaling* behavior provides an easy and natural method for *handling parallelization tasks*. As expected, the latency of serving requests can depend on the used instance [6]. "Cold" starting a new instance is always slower than serving a request with an idle (already "warmed up") instance. Execution platforms also provide *automatic life cycle management* that clears up idle instances to free up unused resources. The massive parallelization capabilities and automatic clear-up implicate, however, that FaaS functions should be stateless and store state information in external data stores. Recently, serverless options gained track in public cloud

Manuscript received June 1, 2020; revised November 20, 2020; accepted January 5, 2021. Date of publication March 10, 2021; date of current version August 19, 2021. This work was supported in part by the AWS Cloud Credits for Research Program and the BRAINE Project through ECSEL Joint Undertaking (JU), the JU receives support from the European Union's Horizon 2020 research and innovation programme and from MIUR, Italy, under Grant 876967. *(Corresponding author: István Pelle.)*

István Pelle is with the MTA-BME Network Softwarization Research Group, Budapest University of Technology and Economics, 1111 Budapest, Hungary (e-mail: pelle@tmit.bme.hu).

Francesco Paolucci and Filippo Cugini are with the National Inter-University Consortium for Telecommunication (CNIT), 56127 Pisa, Italy (e-mail: francesco.paolucci@cnit.it; filippo.cugini@cnit.it).

Balázs Sonkoly is with the Department of Telecommunications and Media Informatics, MTA-BME Network Softwarization Research Group, Budapest University of Technology and Economics, 1111 Budapest, Hungary, and also with the Faculty of Electrical Engineering and Informatics, Budapest University of Technology and Economics, 1111 Budapest, Hungary (e-mail: sonkoly@tmit.bme.hu).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/JSAC.2021.3064655.

Digital Object Identifier 10.1109/JSAC.2021.3064655

environments where they offer clear benefits for providers and users alike. Using FaaS, public cloud providers can ease application development and *offer compute resources with fine-grained resource assignment control*. Users can take advantage of the usual pay-as-you-go pricing scheme as well, as they *only pay for active instances and not idle ones*. In situations where requests arrive in bursts, serverless can handle them more efficiently and at a cheaper price compared to traditional container, virtual machine, or physical server-based solutions. As serverless options gain more popularity in centralized cloud (data center) environments, trials are also being conducted in order to leverage these advantages in on-premises remote facilities or edge scenarios as well.

In this paper, we take advantage of the benefits offered by the serverless concept in a latency-sensitive 5G scenario. We use an edge node and public cloud resources for performing computation tasks, realistic packet-optical networking technology to provide connection and exploit accurate monitoring features across multiple network layers from edge to cloud. To the best of our knowledge, serverless technology has never been experimented at the edge in a latency-sensitive packet-optical scenario provided with detailed and comprehensive monitoring capabilities.

In this work, we significantly expand upon our previous three-page conference paper [7] and present four main contributions. First, we give a high-level design and decomposition of our system capable of deploying serverless applications in scenarios where edge and public cloud resources are used. Our system employs three modules. *i*) The *Layout and Placement Optimizer (LPO)* that accounts for application requirements and resource availability to define the optimal placement of serverless functions. *ii*) The *Serverless Deployment Engine (SDE)* which performs the actual coordinated FaaS deployment on either cloud or edge resources. *iii*) The *Telemetry Service (TS)* that collects real-time statistics from the network, the application, and the compute infrastructure to provide feedback for changing application component placement decisions. As our second contribution, we show implementation and operation details of two of the designed modules, the SDE and the TS. Our third contribution is a latency-sensitive serverless application that remotely controls a robot leveraging a machine learning-based (ML) latency-sensitive subfunction for person recognition. Our final contribution is the evaluation of our system utilizing this control application in a hybrid scenario with on-premise edge, and public cloud compute resources as well as optical metro networking. This transport infrastructure includes both packet and optical transport technology, thus reproducing a realistic heterogeneous edge to cloud interconnection system. Our framework allows the ML subfunction to be deployed either at the edge or in the cloud, and to be dynamically reconfigured in case of lack of computing resources at the edge or excessive latency in the end-to-end route to the cloud. We provide performance metrics concerning the application's execution and deployment in two scenarios: utilizing complete redeployment and a much faster reconfiguration option. Our paper discusses these contributions according to the following structure. First, an analysis of both serverless and networking related technologies is given in Sec. II. In Sec. III, we present our high-level system decomposition and in sections III-A–III-F we review the operational phases of the system and highlight implementation details. Due to space reasons, we give only a short summary of our LPO module in this part. We present our experimental control application and experimental setup and evaluate key aspects of our system as well as application component offloading in Sec. IV. Finally, we conclude our paper in Sec. V by discussing lessons learned and future improvement possibilities.

II. Related Work

Edge computing being a hot topic, multiple works try to answer open questions introduced by this field from both academia and industry. In the following, we give a short summary of these.

A. Networking for 5G Edge and Cloud Latency-Sensitive Applications

Networking architectures for 5G and edge enabling low latency applications closer to the user are hot topics nowadays. A number of works propose solutions to accelerate low-latency traffic in the context of the Mobile Edge Computing (MEC) architecture. Low-latency scheduling in the MEC involves optical network access solutions, such as the proposed MEC-enabled fiber wireless access networks (MFWAN) based on Wavelength Division Multiplexing Passive Optical Network (WDM-PON) technology. The work in [8] proposes a planning model minimizing the latency for delay-sensitive services for different functional split options and a novel scheduling heuristic.

MEC dynamic task offload is proposed by the work in [9], in which specific Internet of things (IoT) applications are moved within the edge domain (between end devices and MEC servers) to save processing and memory usage at the terminal sensors. The work targets both offloading selection and scheduling resorting to the different application requirements in terms of computation resources and bounded latency.

Software-Defined Networking (SDN) architectures and functions have also been proposed to optimize service flows subject to latency constraints. The automotive vertical service scenario has been covered in the work in [10], where fog, edge, and cloud systems are under a distributed SDN domain and service slices are implemented with the aim of providing the lowest latency for critical autonomous driving functions. SDN data plane programmability opens the way to advanced operation offloaded inside network devices, targeting performance optimization and advanced data monitoring (e.g., in-band telemetry). At the network edge, applications for augmenting edge nodes by providing dynamic traffic engineering and cybersecurity functions without involving the SDN controller have been proposed in the work in [11]. In-band telemetry services to monitor the SDN network liveness were introduced in [12]. In particular, a proactive network telemetry platform was proposed for latency performance monitoring. Analysis of in-band telemetry covering all possible network paths have been presented in [13] leveraging a source-based path monitoring system. Data collection coordination mechanisms have been shown in [14], using machine learning for processing.

SDN telemetry is also a hot topic in Optical Networks, targeting the disaggregated scenario [5] and the emergency and post-disaster recovery scenarios thanks to the utilization of Google Remote Procedure Calls (gRPC), gRPC Network Management Interface (gNMI), and Network Configuration Protocol (NETCONF)/YANG [15].

So far, networking solutions for latency-sensitive applications at the edge have never been integrated with FaaS solutions, enabling dynamic and automated relocation of serverless functions according to the currently experienced performance. Moreover, while the aforementioned works certainly make application component relocation and monitoring tasks easier, from an application developer standpoint, they are still lacking high-level management features. In this regard, the proposed FaaS solution provides an effective alternative, making software development and maintenance easier as they offer a well-organized framework with advanced monitoring capabilities.

B. Serverless at the Edge

As latency-sensitive applications motivate the migration of computation tasks from the cloud to edge locations, interest in transferring FaaS solutions to the edge is also increasing. One possible option is reusing open source FaaS options, such as Apache OpenWhisk [16], Knative [17], OpenFaaS [18] or Kubeless [19], in a self-managed environment with as little modification as possible [20]. Other works focus more on improving some aspects of FaaS to adapt existing solutions to edge-specific issues, such as [21] that builds an extension over OpenWhisk. Here authors identify problematic areas in serverless edge scenarios based on diverse use-cases (e.g., applications supporting augmented reality, health monitoring, and autonomous vehicles). They touch issues related to task offloading based on type (i.e., latency-sensitiveness, computation or data-intensiveness), collaboration among edge nodes, data or state locality, and handling. In further works, novel methods are suggested for instantiating and moving edge serverless functions among nodes. WebAssembly [22] is used for low latency function and small memory footprint instance creation [23]. Other authors explore the possibility of using Named Data Networking (NDN) [24] concepts to move functions among nodes in a request-oriented manner [25]. An auction-based function placement algorithm is shown in [26], while [27] uses an osmosis inspired method. A different architecture utilizing distributed function dispatching is discussed in [28].

These solutions, however, lock users in to edge resources or private cloud options and thus users still have to manage and pay for the underlying compute and network resources. This is in stark contrast with public clouds' pay-as-you-go pricing model. Firm integration options between edge and public cloud options can alleviate this problem, although, edge-focused solutions do not have support for this yet.

C. Hybrid Edge and Public Cloud Serverless Solutions

From a software maintenance and deployment management standpoint, the best solution is to have a common execution platform for running application components shared by cloud and edge resources. Major public cloud providers, AWS, Google, and Microsoft, are working on moving their services (not only serverless solutions) closer to end-users. They are setting up their own edge locations in densely populated areas [29]–[31] and are preparing offerings to be deployed within the 5G edge locations of telecommunication operators [32], [33]. Two of them also provide options for managing user-owned equipment as well. Microsoft's Azure Functions [34] and AWS's Lambda [35] both have extensions for delegating computation to such devices. (Neither Google's Cloud Functions [36] nor IBM Cloud Functions [37] can be executed on edge nodes. Although the latter is also OpenWhisk-based, it does not provide integration with user-managed OpenWhisk deployments.) Azure IoT Edge [38] provides the possibility of remote deployment of business logic to the edge using the Moby [39] container engine. Scaling and cloud to edge data communication are, however, severely limited in the current implementation. As the largest player, AWS provides the most versatile service. AWS Lambda offers built-in support for multiple programming languages and an option for creating custom runtime environments for executing code written in other languages. The service uses AWS's own Firecracker [40] virtualization environment for running on-demand functions in the cloud and provides a single parameter for adjusting memory size and available virtual CPU resources in fixed steps. It integrates well with other AWS services of which many have built-in support for invoking the service with highly varying delays, as reported by [41]. Unused function instances in an idle state are cleared up by the AWS Lambda framework using an undisclosed algorithm. AWS IoT Greengrass [42] is part of AWS's IoT services and as such, it provides an extension for Lambda to execute functions at the edge. The Greengrass Core is the central element of the service. It runs on the edge node and manages deployment, execution, and communication tasks for Lambda functions deployed to edge devices. Additionally to on-demand functions, the service enables the creation of long-running ones that never time out. Using the service, existing cloud-based Lambda functions can be assigned to edge nodes. As the service can change Lambda configuration for edge-deployed Lambdas, the same code can be executed in the cloud and at the edge using different configurations (e.g., assignment of computation power and environment variables). The Core provides access to edge-local resources such as files and devices, while also providing full-fledged cloud to edge, reverse, and edge to edge communication using a publish/subscribe message exchange mechanism. For communication between functions on the same edge node, the Greengrass Core utilizes its own interprocess communication methods, while the cloud and other edge nodes can be accessed via an interface using the Message Queuing Telemetry Transport (MQTT) protocol. As communication between edge nodes is not direct and has to traverse the message broker in the cloud, this type of data exchange is less efficient and is expected to be subject to higher latency. The Greengrass Core supports Lambda execution with three different containerization options: Greengrass or Docker containers, or without containerization. While each of these can have advantages based on the use-case

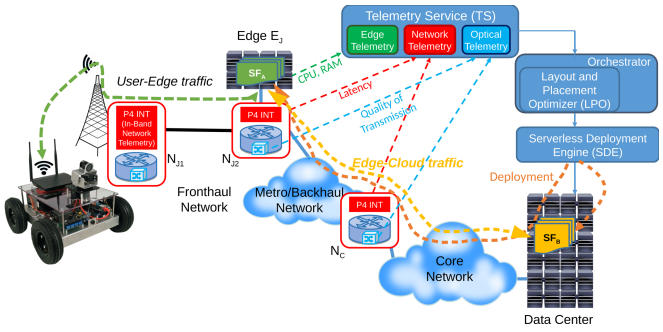

Fig. 1. 5G and edge telemetry: architecture showcasing the placement of serverless functions SF_A and SF_B [7].

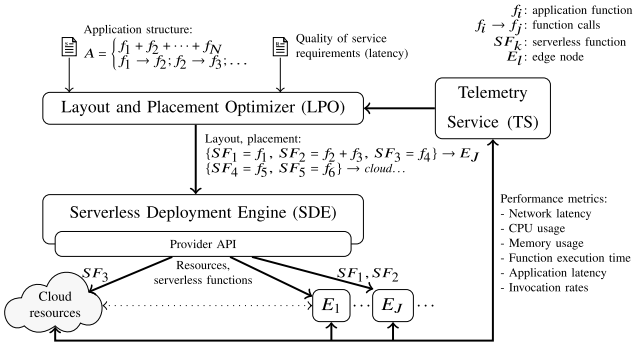

Fig. 2. Schematics of code deployment and monitoring (extending on our previous work [7]).

at hand, most of the features provided by the service can be accessed only via the Greengrass containerization option. As for monitoring deployed code, the service provides options for logging data locally and in the cloud. Configuration of the Core, the edge Lambda functions, other deployment resources, and monitoring is performed through the AWS provider API for which a configuration endpoint is available in the supported AWS regions. AWS CloudFormation (CF) [43], AWS's general deployment service, is also able to access this endpoint, thus resource provisioning of cloud-only and edge related functions and resources can be controlled by a single entity. However, both Greengrass and CF platforms put less emphasis on moving code between edge nodes. While CF excels at resource setup, it lacks a high-level interface for specifying application components and their intended setup. While monitoring data can be collected at the same centralized location in the cloud, monitoring deployed FaaS code, especially in hybrid edge cloud scenarios is still cumbersome, thus it is also a weakness of the platform.

III. Proposed Approach

Fig. 1 shows the reference packet over optical edge-to-DC converged fronthaul and backhaul network architecture. It extensively exploits telemetry and data plane programmability to provide deep network awareness to serverless control systems. Telemetry-based monitoring is activated from all network elements (i.e., optical and packet systems). P4-enabled network interface cards (NIC) equipped with optical pluggable transceivers [44] are used in edge compute nodes, providing direct optical connectivity augmented with in-band monitoring capabilities. In particular, P4 programmability is exploited to introduce ad hoc header extensions directly by the NIC, thus accounting for latency contributions (e.g., queuing time) directly from the server and updated throughout the whole network up to the data center. The proposed architecture encompasses a serverless control system based on three main functional modules, shown in Fig. 2. The *Layout and Placement Optimizer* (LPO) is responsible for determining the placement and layout of the components of an application that is given by its functions (smallest individually deployable components, f_i) and the call hierarchy among those. Based on quality of service (QoS) indicators measured through telemetry on the 5G transport and computing infrastructure, the LPO computes an optimal grouping (SF_i) of the application's functions and assigns these to either edge or cloud resources while satisfying the application's QoS requirements. The actual deployment is performed by the *Serverless Deployment Engine* (SDE) component via calls through the *Provider API* that has direct connections to cloud and edge resources. After deployment, the *Telemetry Service* (TS) module monitors application and network performance and reports related metrics to the LPO. As completing a whole optimization and (re)deployment cycle in the above system is expected to be relatively slow, we envision bidirectional communication between the application components and the TS. This provides the application components with (limited) capabilities for reconfiguration without waiting for complete redeployment.

Fig. 1 also shows the use case of serverless function instantiation deployed at the edge and in the cloud. *Serverless Function A* (SF_A) is instantiated close to the user equipment to perform low-latency operations (e.g., image processing and inference for critical object recognition), while SF_B is deployed in the cloud performing offloading operations subject to bounded latency constraints (e.g., image upload to a repository). The Telemetry Service operates on different transport segments, spanning different layers. The *Optical Telemetry* (OT) module collects Quality of Transmission (QoT) monitoring information from the optical layer leveraging disaggregated streaming of Optical Signal-To-Noise Ratio (OSNR) values at the coherent receiver and power values at intermediate nodes [5]. The *Network Telemetry* (NT) module collects packet statistics of target traffic in the form of in-band telemetry mirrors, exploiting P4 switches. The *Edge Telemetry* (ET) module collects IT statistics, such as edge CPU rate and memory usage. This way, all layers are monitored to detect QoS degradation. For example, if the TS detects a latency increase in the metro segment due to network congestion, the SDE may trigger dynamic reconfiguration of serverless functions from the cloud to the edge, guaranteeing adequate latency performance for the time-sensitive application.

In the following, we discuss the design and implementation details of the overall system. Fig. 3 shows the complete process of deploying an application and its related telemetry data collector functionality to edge and cloud resources. Here we summarize the main steps, operation phases and components

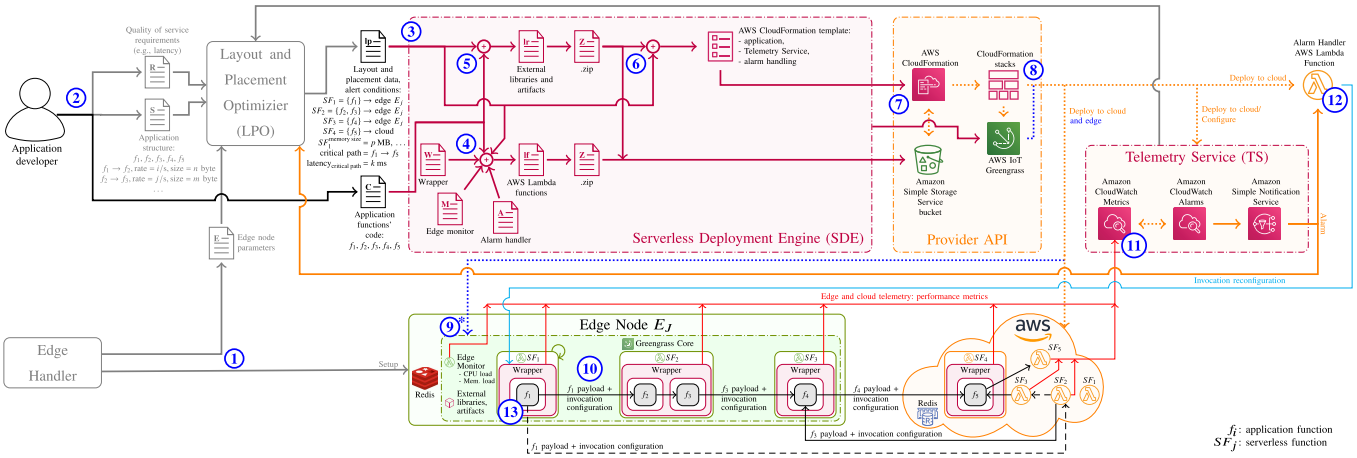

Fig. 3. Proposed system: overall architecture and details of SDE (Serverless Deployment Engine) and code deployment stages.

with a specific focus on the design and implementation of the SDE and TS main modules which expand upon our previous works for incorporating comprehensive telemetry management.

A. Setup Phase: the Edge Handler and the Layout and Placement Optimizer

During the bootstrap phase, the edge nodes are set up and their parameters are supplied to the LPO (see step 1 in Fig. 3). This is done by a dedicated module that synchronizes edge node configuration information between the physical devices and the AWS cloud. The setup of the application is initiated by the developer/operator who provides the input descriptions (step 2 in Fig. 3). The first description includes application functions, their expected average execution time and invocation hierarchy, as well as access to data stores, expected invocation rates, and transferred data sizes. The second part of the input provides QoS requirements: latency constraints on critical paths in the call hierarchy. The calculation of the optimal layout is based on the extended version of the algorithm proposed in our previous work [45]. The original method was able to handle only cloud deployment to central AWS services, while the extension makes it capable of leveraging hybrid edge cloud infrastructures as well. As our main focus in the current work is on the deployment and performance feedback phases, we give only a high-level summary of the relevant extensions. The detailed discussion of the updated LPO is part of a separate work.

The LPO handles two internal models. *i*) The *platform model* describes platform capabilities and pricing schemes. The capability model captures the performance characteristics of the main services of AWS which play role in the application composition, i.e., Lambda, Greengrass, data store, and invocation solutions. In the platform capability model, we introduce distinct resource flavors for the cloud and the edge domain, i.e., we create Lambda and Greengrass resource assignments augmented with the respective execution and invocation characteristics. These model parameters are set a priori based on reference measurements for the initial calculation and data coming from the Edge Handler in step 1. During application execution, these are updated on-the-fly based on live monitoring to help dynamic adaptation to changing circumstances on a longer time scale. In the platform pricing model, we also make distinctions between cloud and edge resources. The pricing model of cloud resources follows the real pricing scheme of AWS, while we define large virtual costs for Greengrass resources for controlling the placement decisions in a simple way. Therefore, edge resources are used only if the latency bounds require that, otherwise, central cloud resources are preferred. *ii*) The graph-based *application model* is initialized in step 2 of Fig. 3. It encompasses functions and data stores as nodes, while function invocations, data store read and write operations are represented by edges. Making use of these models, the LPO calculates a cost-optimal deployment layout that satisfies the given latency bounds. As a result, application function grouping (the group of functions to be packaged into the same serverless artifact), data store setup, their assignment to edge or cloud resources and invocation methods, as well as latency criteria are passed to the deployment engine.

B. Deployment Phase: the Serverless Deployment Engine

The two main tasks of the SDE are the following: to compose the deployable serverless artifacts and to deploy the whole application making use of the underlying Provider API. The composition task is realized in multiple steps. The SDE processes the input from the LPO and combines that with the application codes provided directly by the developer. Based on these pieces of information, the SDE assembles the serverless functions (deployable artifacts) including the selected groups of application functions (step 3), external libraries plus machine learning artifacts (step 5), and special-purpose runtime extensions (step 4). These latter extensions are central to our concept and realize core features required by the overall operation. The multi-purpose *Wrapper* i) enables the grouping of application functions by providing the communication layer among internal and external entities, ii) helps to adapt user code to different (edge or cloud) environments, iii) provides monitoring data, and iv) implements a dedicated interface for layout reconfiguration. The details are discussed in the sections III-C, III-D and III-F highlighting the main operation phases.

In addition, the *Edge monitor* and the *Alarm handler* functions are added as further artifacts. The former is responsible for monitoring CPU load and memory usage on edge nodes while the latter enables quick reactions to alarms raised by the Telemetry Service. The compressed serverless artifacts are finally uploaded to AWS.

In our implementation, the whole deployment is described by an AWS CloudFormation (CF) template (created in step 6), including the placement, resource flavor, and connection information, which can be added directly to the corresponding Provider API. CF deploys all cloud resources, while Greengrass sets up components on edge nodes. At step 9, this latter service finishes with application deployment and the application is ready for handling incoming requests as shown in the following.

C. Application Execution Phase: the Wrapper

In our proof-of-concept prototype, the Wrapper extends the capabilities of the default AWS Lambda Python runtime environment. More specifically, it enables the execution both in cloud and edge environments, supports different invocation options between serverless functions and implements access to different state stores transparently, i.e., the low level invocation and access technologies are hidden from the application. The bottom part of Fig. 3 shows a sample deployed application that has components on an edge node E_j as well as in the cloud according to the grouping and distribution shown in Fig. 2. In the default deployment, the single application function f_1 is put inside serverless function SF_1 while the Wrapper provides access for f_1 to the Redis data store on the edge, and call options to other functions located on the same edge node or in the cloud. In this setup, f_1 calls only one function, f_2, which is grouped together with f_3 in SF_3. As f_1 places a call to f_2 in step 10, f_1's Wrapper redirects it to SF_2 automatically while f_2's Wrapper captures the incoming call and forwards the payload, sent by f_1, to f_2.

D. In-Application Performance Monitoring: the Wrapper and the Alarm Handler

The Wrapper is also in charge of monitoring the runtime characteristics of all constituent functions. It is capable of measuring the key metrics describing application performance, such as invocation rate, latency, and size of the payload for both cases of accessing another function or a state store. For example, when f_1 calls f_2 in step 10, all these metrics are logged to our Telemetry Service (TS) in step 11. In case of edge deployment, edge node related metrics (CPU load and memory usage) are also sent to the TS by our *Edge Monitor* serverless function. We built our implementation of the TS on available AWS services and configure them together with the application deployment in step 8. More specifically, the TS leverages the capabilities of Amazon's general-purpose logging service, CloudWatch (CW), to store metrics and constantly monitor them, while CW Alarms is configured to watch out for alarm conditions specified by the LPO in step 3. We implemented a completely push-based alarming method for the TS: upon detecting limit violations,

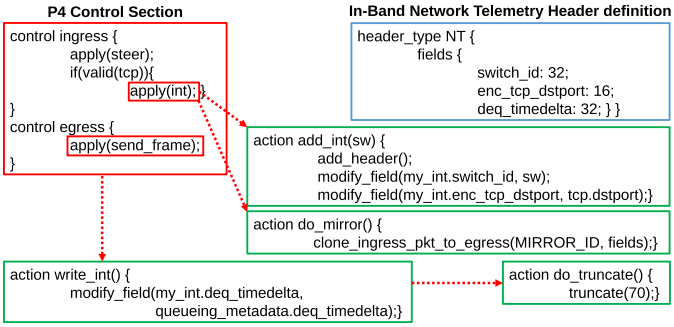

Fig. 4. P4 code employing telemetry system [7].

the alarm is pushed to two distinct system components, the LPO and the *Alarm Handler*. Although the former has more capabilities to adapt the application to changing conditions, new LPO layouts result in total application redeployment, which is a slow process, as shown later in Sec. IV-E.1. The Alarm Handler enables bidirectional communication between the TS and the application itself and provides a way for quick SF offloading from the edge to the cloud and reverting these changes. As this is not a complete redeployment, we name it reconfiguration.

E. External Performance Feedback: Network Telemetry

As our Telemetry Service utilizes AWS's general logging capabilities, we can extend the application and edge node-specific metrics with others as well that can enrich the reoptimization and reconfiguration capabilities of the LPO or the Alarm Handler. In our implementation, we provide network-related metrics that resort to telemetry points in the 5G transport network enabling fast detection of optical impairments and queue congestion on packet nodes due to dynamic traffic conditions (e.g., a heavy load of edge SF deployment connected to the same backhaul node). To this goal, the metro network connecting the Edge node is composed of a number of SDN P4-enabled switches performing in-band network telemetry (INT). Specifically, the edge-cloud service traffic flow is matched and tagged with an extra header providing network metadata information.

In this work, two kinds of network metadata are processed. *i*) The switch transmission queue latency (i.e., the time spent by each packet in the switch transmission queue) is embedded in the extra header, carried out by the packet traversing the metro network and extracted at the metro network gateway. This enables detailed monitoring of the metro network latency both on the granularity of switches and on an edge-to-edge network level, allowing detection of possible congestion or quasi-congestion points, supposing direct control by means of the SDN control system. *ii*) The round-trip time between the metro edge node and the AWS data center which allows monitoring of the core network latency towards the cloud infrastructure, supposing indirect or lack of control on such network segments.

Excerpts of P4 code related to metro nodes are shown in Fig. 4 to describe the INT operation performed within the switch. The P4 program includes a header parser description,

an ingress pipeline control description, and an egress pipeline control description. For each control section, a sequence of flow tables is defined, including a set of actions to be applied to the matched packet. In particular, each P4 switch, in the ingress section, defines and applies an extra header of type INT storing latency information computed by the P4 node itself. The header definition is custom and includes, in this work, a switch id (identifying the node performing INT), the layer-4 port identification over which the telemetry is applied (e.g., UDP port), and a 4-byte field carrying out the latency value expressed in microseconds. The action *add_int* inside table *int* triggers the creation, the encapsulation, and the update of the fields of the INT header. Moreover, dynamic cloning of selected service traffic headers is activated to provide INT header for the TS (action *do_mirror*).

At the egress section, while applying the *send_frame* table, the switch computes the queue time metadata and writes it into the header (action *write_int*). In addition, for mirrored traffic only, it truncates the packet to provide just the header stack (action *do_truncate*). Such packet treatment is repeated for all the crossed P4 nodes until the last metro network node (e.g., N_C) extracts INT and removes extra headers, thus avoiding telemetry to reach end-users.

Similarly, but in an end-to-end configuration, a different INT is applied between the metro network edge node and a P4 software switch instance deployed at the cloud. Packets reaching the cloud switch, before reaching the application virtual machine, are then mirrored and sent back to the edge node, which computes the round trip time. The TS NT module collects per-packet latencies related to the edge-metro segment and to the metro-cloud segment, respectively, and builds statistics to be provided to the LPO and the Alarm Handler.

F. Application Reconfiguration Phase: the Alarm Handler and the Wrapper

Fast reconfiguration of the already deployed and running application is a key feature of our proposed system, which is provided by the Wrapper and the Alarm Handler modules. As each serverless function that is deployed to the edge is also onboarded in the cloud, we can dynamically change the target of function calls depending on the current system state (e.g., system load, network utilization). Besides, Greengrass supports a special type of function on the edge that can run indefinitely, which provides a single entry point for (re-)configuration. Thus the Alarm Handler works together with long-running edge functions and reconfigures parts of the application appearing after such functions. In our example shown in Fig. 2, SF_1 is such a long-running function. New application configuration sent out by the Alarm Handler in step 12 arrives at a specific REST endpoint maintained by the Wrapper leveraging Flask [46]. In our example, after f_1's Wrapper receives the reconfiguration command, it changes subsequent calls of f_2 from edge to cloud in step 13. As SF_2 is configured the same way in the cloud as on the edge, by default, it will call SF_3 on the edge as well. Changing invocations of subsequent functions is realized by in-band communication between Wrappers in different serverless functions as complete invocation configuration is forwarded by each Wrapper together with function payload. This also provides the possibility of reconfiguring on-demand functions. We note that this method has the following limitations. Long-running functions are always required and only the part of the application after the long-running function can be reconfigured. When the first function of the application is such a function, this does not prove to be an issue. In certain typical edge scenarios, applications naturally use this structure, in others they can be adapted to it. A further limitation of the implementation is that long-running functions cannot be offloaded from the edge, as AWS Lambda provides only on-demand functions in the cloud. Finally, we note that in our current implementation, the Alarm Handler is application-specific and reconfiguration alternatives have to be provided by the application developer.

IV. Results and Evaluation

We evaluated the performance of our system by deploying an application that remotely controls a four-wheeled robot (later referred to as "rover", for simplicity) over an optical edge-metro network. We provide the details of the control application in Sec. IV-A and discuss network and edge equipment setup in Sec. IV-B. As for the rover, we used a SuperDroid Robots Programmable WiFi Mobile 4WD Robot that was built on the manufacturer's IG32 DM chassis having 6.1-inch wheels and equipped with DC motors capable of 53 RPM, making the rover able to reach 0.43 m/s maximum speed. The rover is controlled by an onboard Arduino Mega 2560 R3 microcontroller and the images, captured by a wide view onboard camera, can be accessed via a ASUS RT-N12 D1 Wireless-N300 router mounted on the chassis.[1]

Using our experimental setup, we investigated key aspects of deploying and running our use-case application with the SDE. Based on our observations gained by running every application component on our edge node, we determined metrics and limits for offloading the machine learning powered inference to the AWS cloud. In Sec. IV-C we present reference measurements concerning the Greengrass execution platform, while in Sec. IV-D we show reference metrics attained from the testbed. We conclude our evaluation in Sec. IV-E where we discuss application offloading characteristics using our two available options: complete (re)deployment and fast reconfiguration.

A. Experimental use-Case: Rover Remote Control

We built a Python 3.7 serverless application on a rover control use-case and split it into functions deployable by our Serverless Deployment Engine. The aim of the application is to detect objects that are in the rover's field of vision and stop or resume the rover's movement based on them. Object detection and classification is a field where machine learning algorithms excel, however, the rover is scarce in compute resources, thus

[1]See complete specifications at: https://www.superdroidrobots.com/shop/item.aspx?itemid=1320

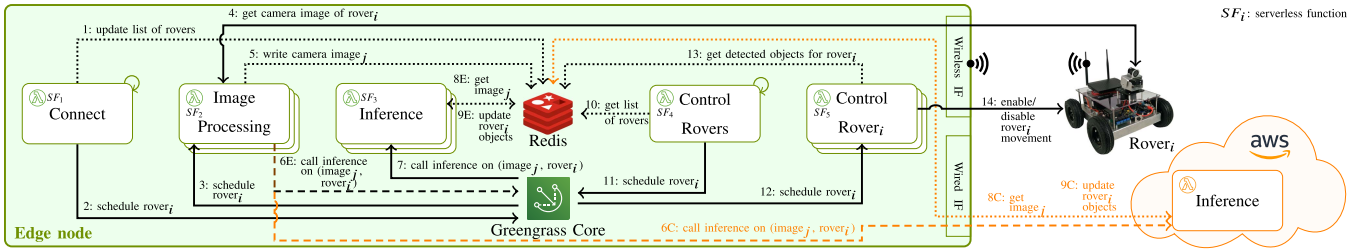

Fig. 5. Remote rover control use-case.

it cannot run such computation-intensive processes. As in most cases, such movement control tasks are highly latency-sensitive, thus processing needs to be run "close" to the rover. Placing the object detection on the edge provides a balance between having ample compute resources while still retaining low latency.

Fig. 5 shows our edge-cloud native layout of the application that is designed to be able to serve multiple rovers at the same time. The application is split into five different functions that will be deployed as AWS IoT Greengrass functions to an edge node located at our premises. Besides these, a non-AWS managed Redis in-memory cache is running on the edge node. The local Redis instance helps in exchanging information between the two loops and conserving bandwidth in function calls. To manage the dispatching task of handling multiple rovers, we leverage Greengrass's long-running Lambda functionality. Each of our two in-application loops is managed by dedicated long-running functions that serve as entry points. The first loop handles image processing and object detection tasks. Here, the *Connect* function checks for nearby rovers while managing a list of connected rovers in the cache (see step 1 in Fig. 5). This function triggers the *Image processing* function periodically by specifying the rover from which the image should be gathered. This call, as with every other call on the same edge node, goes through the AWS Greengrass Core. Image processing is an on-demand function that queries the specified rover's camera and saves it to the local Redis cache. The function then triggers a second on-demand function, *Inference*, by supplying the rover identifier and the grabbed image's location in the cache. The Inference function is responsible for detecting objects on the image, read from the cache, leveraging OpenCV [47] and its Deep Neural Networks module as well as the MobileNet-SSD network. As Image processing and Inference are both on-demand functions, we can handle object detection for multiple rovers at the same time leveraging the serverless execution platform's parallelization capabilities. The second loop has a similar design and it is responsible for controlling the movement of the rover. The long-running *Control Rovers* function checks the list of connected rovers in the local cache and passes connection information of a selected rover to an instance of the on-demand *Control Rover*$_i$ function. This latter function reads currently detected objects for the given rover from the cache and based on the recognized objects, sends control messages to the rover. The function stops the rover whenever an object falling in the *person* category is detected and resumes rover movement otherwise.

As noted above, handling multiple rovers and images is done by starting multiple instances of the respective functions. As Inference is computation heavy and has a comparatively long execution time, total application performance is heavily dependent on application trigger frequency and CPU resources. As an example, consider the following case of having an edge node with four CPU cores. Let's also assume that a single instance of the Inference function consumes one CPU core and with that it runs for 1 s, while the rest of the application components execute much faster and they require only a single CPU core together. In case of handling a single rover, we can sample the rover's camera with a rate of 3/s without overloading the edge node. Connecting a second rover or performing more frequent sampling thus would increase inference latency and consequently delay the movement control of the rover. However, by using a better-equipped edge node or offloading computation to the cloud, we can avoid this problem. Although the latter option (also shown in Fig. 5 with orange highlights) would certainly increase the latency in the rover control, it might still be a cost-effective solution for handling the issue. In this case, as noted in Sec. III-B, our system hides the differences between calling an edge function and a cloud function, while also providing access to the Redis instance running at the edge.

B. Experimental Testbed Setup

The proposed telemetry-driven serverless architecture has been implemented and validated in an edge-cloud network testbed (shown in Fig. 6) employing packet-over-optical metro networks including P4 switches and optical transport through transponders and reconfigurable optical add-drop multiplexers (ROADM). P4 switches are implemented over Linux server boxes equipped with 10 and 40 Gb/s optical interfaces and implementing the Behavioral Model version 2 (BMV2) with P4 code supporting custom INT and selected mirroring of packet headers. The switches are connected to the optical layer by means of commercial 100G optical cards and QoT telemetry is realized through NETCONF-based subscription to gRPC streaming of OSNR values received at the card (e.g., as in [5]). Additional dynamic traffic is injected in the backhaul network using a Spirent N4U generator, connected with the Edge P4 Switch through 10G optical interfaces. We used a Linux machine as our on-premise edge node, equipped with four Intel Xeon E5-2650 v3 vCPUs, 6 GB of memory, Ubuntu 18.04, Greengrass Core 1.9.2-RC4, P4 NICs with optical interfaces, and a wireless interface providing

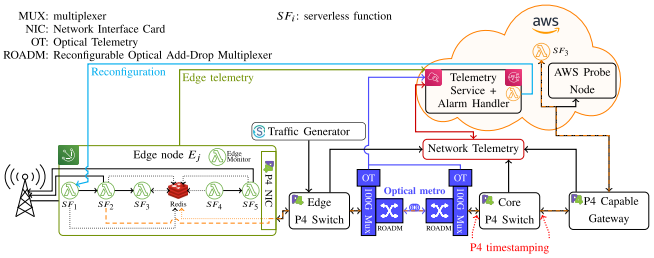

Fig. 6. Testbed setup and components of the telemetry system (extending on our previous work [7]).

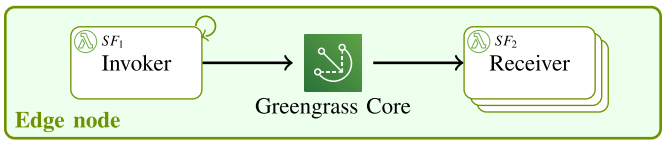

Fig. 7. Startup measurement setup with serverless functions SF_1 and SF_2.

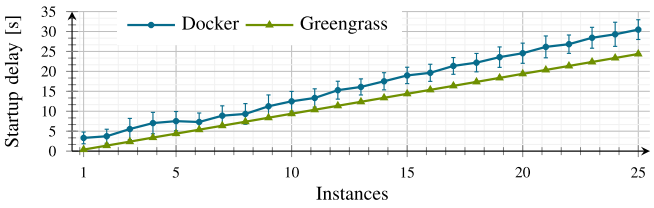

Fig. 8. Startup times of multiple concurrent instances.

Wi-Fi connection to the rover. For cloud execution, we used the *eu-west-1* (Ireland) AWS region.

C. AWS IoT Greengrass Scaling Behavior at the Edge

Since starting up, running, and stopping functions at the Greengrass edge presents significant differences compared to AWS Lambda cloud, we first investigated these aspects. Greengrass offers different containerization options for instantiating functions at the edge, among which the Greengrass container is the most versatile one. As the SDE specifies that as well, we investigated the performance of function cold startup and how it scales to multiple concurrent instances. In order to get a base-line, we used a simplification of our use-case application for running the tests, shown in Fig. 7. The long-lived *Invoker* (i.e., the Connect function in Fig. 5) is responsible for requesting new instances of the on-demand *Receiver* (i.e., Image Processing). For the test, we removed the delay caused by loading external packages and Wrapper overhead as well, both of which are highly dependent on the application at hand. We also removed any computation tasks from the functions, as well as average Greengrass function call latency for the specific payload size (2 ms), in order to get the raw performance data of the Greengrass container management. For comparison, we selected Docker [48], a vastly used containerization option, and measured its performance as well by using simple scripts for instantiation triggers. In our measurements, we simulated the simultaneous arrival of multiple requests. As functions serving these requests did not perform any computation tasks but waited for an extended period of time (60 s in our case), we were able to capture the cold start performance of multiple function instances. Fig. 8 shows the average startup latency and standard deviation in the case of 25 simultaneous requests gained from repeating the measurements 100 times. Data show that Greengrass is able to start up the first container in 387 ms. New instances are started up with a 1 s fixed delay. In each case, cold start delays produce small variations. Docker, on the other hand, proves to be significantly slower as it can achieve 1.9–6.1 s higher average cold start delays than Greengrass with much higher variance. Our measurements also revealed an undocumented limitation of Greengrass: the service does not start up more than 25 instances from the same function. In cases when more than 25 requests are waiting to be served, Greengrass waits for existing function instances to finish processing and further requests will be assigned to those.

Using the same setup, we also measured communication delay between two initialized function instances. Greengrass used its built-in invocation methods while we used HTTP communication leveraging Flask between Docker containers that were connected to the same network. Our measurements show that communication delays are similar in both cases. Greengrass provides 2.32–4.29 ms of average delay for packets of 100 bytes to 128 kB in size and Docker can achieve 2.41–3.15 ms with low variations in all cases.

Lessons Learned: Greengrass containerization provides adequate instantiation latency in case of a low number of simultaneous requests and loading small external packages or files. Although the addition of a 1 s delay between function instance startups seems to be a really simple built-in load balancing strategy of the Greengrass Core, paired with the low instantiation delay, it provides better performance than Docker. With the additional benefits of code, resource, and runtime management, as well as easily configurable and fast communication between function instances and integration with the AWS cloud, we find it to be a good solution for handling computation tasks at the edge. We also believe that the upper limit for the number of function instances is not problematic in our case as we assume that edge devices have access to limited computation power.

The aforementioned Greengrass cold start behavior is completely different from how scaling is handled in the cloud, where the initialization of subsequent instances is not delayed. With a respective measurement setup in the cloud, we measured 200 ms average cold startup latency irrespective of the number of concurrently invoked function instances. We also note that a higher level of performance degradation can appear at the edge node after the number of concurrent instances exceeds the number of available vCPUs. Because of two reasons, however, we leave further analysis of Greengrass startup behavior under edge CPU exhaustion conditions for future studies. *i*) It requires a deep investigation since it is highly dependent on the considered functions, CPU requirements of the executed compute task, its duration, and the rate of incoming requests. *ii*) The objective of this work is to focus on preventing the exhaustion of edge resources by offloading selected functions to the cloud in advance.

Other differences between Greengrass edge and AWS Lambda cloud occur for instance eviction. In Greengrass,

TABLE I
P4 SWITCH IN-BAND TELEMETRY RESULTS: PER-NODE LATENCY

Max TX rate	Congestion Threshold (Mb/s)			TX Queue latency (μsec)		
(kframe/s)	L_d = 128 byte	L_d = 1 Kbyte	iMIX	NO	UT	AT
5	5	40	15	70	800	12700
10	11	82	30	70	300	6350
20	22	164	60	70	150	3200
40	45	328	120	70	100	1550
80	90	655	242	70	100	750
120	160	-	-	70	100	500

it is dependent on either reaching a predefined timeout, throwing an exception (same as in the cloud), or running out of host memory which is a difference compared to the cloud. While Greengrass limits memory access of new function instances to user-defined values, it cannot guarantee that an instance will have access to the specified amount of memory. When a new parallel instance of an on-demand function SF_i is started up, and the edge node runs out of memory, the Greengrass core selects the "oldest" instance of SF_i and terminates it, irrespective of the health of the process.

Lessons Learned: Burst invocation of functions having a high memory footprint can cause serious performance degradation at the edge. Application component assignment to edge resources has to take into account this behavior as well.

If the edge node has enough memory to handle every concurrent function instance, the Greengrass Core keeps all of them alive and warmed up even when instances are idle. Our measurements showed that idle edge containers were still up after three hours, while in case of cloud Lambda functions, idle instances are cleared up after an undisclosed amount of time which, usually, can be measured in minutes.

Lessons Learned: As function instances at the edge are kept warm, moving functions back from the cloud to the edge spares "cold" start latency making such transitions more efficient. This behavior can be also leveraged for mitigating the initial instantiation delay mentioned above by triggering functions in advance.

In conclusion, we can say that resource management of serverless functions at the edge is still limited compared to AWS's cloud offer. Greengrass performs better than Docker. However, during application startup, we can expect issues if the amount of concurrent cold instances exceeds the available vCPUs. In our use-case, such burst calls could happen only when handling multiple rovers or using a high sample rate of the rover's video feed. In both cases, only the initial phase of the application's execution is affected since instances are kept warmed up indefinitely later on. In the case of applications requiring lower latency, either at startup or later on during their lifetime (e.g., when requiring more function instances), this might not be adequate. CPU and memory availability is a more crucial problem, as Greengrass does not perform checks on them before starting up new instances. This confirms that a viable trigger to offload functions from the edge to the cloud can be a high level of CPU or memory usage.

D. Networking Aspects of the Application and the Testbed

Tab. I shows the detailed latency values and behavior detected by the Telemetry System (TS) and experienced by the traffic flowing between the edge node and the AWS cloud under different network traffic scenarios at the intermediate P4 switch. To measure significant node latencies and profile latency behavior during bottleneck and congestion events, one P4 switch interface transmission rate has been limited to emulate real network congestion. The interface transmits the service traffic subject to INT as well as additional synthetic interfering traffic not subject to INT. Results show that in the case without synthetic traffic generation (NO) the latency experienced in the queue is always below 0.1 ms. In the case of synthetic traffic injection, latency behavior changes based on the value assumed by the disturbing traffic rate. If traffic is significantly below the Congestion Threshold, the latency remains low and flat as in the NO case. If the optical path is longer or external traffic increases, latency may reach warning conditions. In the under threshold condition (UT) queues are populated but not full, whereas for a rate range close to the threshold (approximately 5–10 Mb/s large) a slight latency increase occurs, in the range of 0.1–0.8 ms. This value increases as the max transmit (TX) rate decreases. Further congestion may occur, e.g., in case of failure in the optical network or traffic peaks, leading to additional latency. Above the threshold (AT), traffic experiences significant delays in the node queue, in the range of 0.5–12 ms, impacting edge-cloud QoS. In addition, the table reports the values assumed by the congestion threshold as a function of the max TX rate and the distribution of the disturbing traffic packet length. Three length value profiles are evaluated, two constant (set to 128 bytes and 1000 bytes, respectively) and one following the iMIX distribution. Results show that short length traffic decreases the threshold.

Lessons Learned: As having a congested metro network adds high latency to communication with cloud resources, TX queue latency can be a good indicator when to move compute jobs from the cloud to the edge.

Finally, the additional time required at the forwarding plane to perform the P4-based operations of INT insertion/extraction and packet truncation/mirroring has been assessed. Specific measurements have been conducted using a Spirent N4U traffic analyzer and generator. Results show an additional forwarding time to traverse the P4 node in the order of 0.1 ms in the considered BMV2 software implementation. This time becomes substantially negligible when hardware-based implementations are adopted. For example, a P4 code of similar

complexity implemented over FPGA led to an overall packet forwarding time of less than 5 μs from input to output switch port [11]. This time further reduces in ASIC-based implementation within switches or the emerging P4-based NICs.

E. Offloading Remote Rover Control Computation

With the application deployed to the environment shown in Fig. 6, we examined the effects of offloading the application's Inference function (SF_3) from the edge node and investigated its reversal as well. In accordance with our observations in Sec. IV-C and IV-D, we set up a combination of trigger events to perform the necessary tasks. When CPU usage at the edge reaches a critical level (above 80%), computation has to be offloaded from there. This would result in increased but still tolerable end-to-end application latency. Further on, when the TS finds that the TX queue latency reaches an AT level, computation has to be moved back to the edge node, in order to gain end-to-end latency.

A building block of TS, Amazon CloudWatch Alarms, specifies a minimum of 10 s of monitoring period which limits the reaction time of the module. Our measurements show that limit violation detection by the TS is 16.23 s on average (based on 100 measurements). This encompasses the time needed at the TS for finding that one of the metrics is above the specified level and notifying the appropriate endpoint via AWS Simple Notification Service (SNS).

Lessons Learned: In the rover application's case, such a time value represents a good trade-off between guaranteed performance and system stability, avoiding excessive reconfigurations between edge and cloud. In case of more dynamic scenarios at the edge, two main strategies can be adopted. The first strategy consists of lowering the CPU (and memory) load threshold for triggering the offload from the edge to the cloud. This would provide a higher margin to safely operate the functions at the edge before limit violations are successfully detected by the TS. The second strategy to follow can be the replacement of the CloudWatch tool within the TS with a different monitoring tool that enables the collection and querying of metrics on a much finer grain. In our case, multiple TX queue latency values are logged to the TS in a second and the CPU load can be collected once in each second as well, however, CloudWatch Alarms can process only aggregates of 10 s. Having the ability to use a smaller detection window would grant us the possibility to achieve better detection performance.

After a violation is detected, offloading can be done using two methods. On the one hand, function placement can be changed by redeploying the application. This method provides an opportunity for complete recalculation of the application layout (i.e., changing function grouping and assigned resource sizes) and total adaption to new usage patterns or infrastructural changes. Because of layout changes, this method is expected to be relatively slow. On the other hand, the application can be reconfigured as per Sec. III-F in a limited way by changing only the placement of components. In the following, we compare these two methods using application outage time as a basis. We define this metric as the time during which the application cannot serve incoming requests in time. For the sake of comparability, in both cases, we will investigate the same two function placements: one that runs every function at the edge while the other deploys the Inference function to the cloud (in the baseline case, it is not even deployed at the edge).

TABLE II
DEPLOYMENT INDUCED APPLICATION OUTAGE

Scenario	Mean	Standard Deviation	Minimum	Maximum
Limit violation detection	16.23 s	2.55	11.18 s	21.61 s
Full edge deployment	124.62 s	7.82	113.10 s	149.16 s
Offloaded deployment	114.83 s	23.88	97.43 s	223.03 s

TABLE III
MEAN DELAY OF DEPLOYMENT PHASES

Scenario	Input conversion	App code mgmt	CF deployment	Edge (re)deployment
Full edge	17 ms	40.15 s	102.37 s	18.57 s
Offloaded	15 ms	76.29 s	94.20 s	17.67 s

1) Baseline: Complete Redeployment: As noted above, in case of switching layouts this way, the application always has to be suspended as functions will be deployed to or removed from the edge node. As Tab. II shows, the outage of the application is around two minutes (based on 25 measurements). As depicted by Tab. III, collecting function code, external libraries, and uploading them to AWS (*app code mgmt*) and *CloudFormation (CF) deployment* have the highest contribution to total deployment latency using the automated method. Application code management is expected to be slow as handling the machine learning artifacts always results in big file uploads. The shorter phase delay in case of a full edge deployment is the result of the SDE deploying every external function artifact in a single resource to the edge. However, in the case of the offloaded scenario, it has to create two of them, one for the cloud (for the Inference function's external libraries) and one for the edge (for the external libraries of the rest of the functions). CF deployment shows a smaller difference between the two cases. Here, the offloaded scenario is quicker as AWS CF does not have to add the Inference function to the edge. Conversion from the input format to AWS CF format is negligible. Edge deployment does not show significant differences between the two layouts, which indicates that deploying the machine learning artifacts to the edge node does not impose a significant burden on either the edge node or the network.

Lessons Learned: As initial deployment always has to take this option, the test provides an estimation in that regard as well. Using the SDE with the high-level layout and placement input significantly simplifies serverless application setup in a hybrid edge cloud scenario compared to the manual case, where every artifact and the CF template have to be created separately. However, performing complete application layout updates this way, by fully relying on CF to calculate changes between deployments, is extremely slow, making this option

inapt for quickly offloading computation tasks from edge nodes. For certain applications and layouts, the outage can be shortened by moving part of the change calculation to the SDE from CF and using a more complex deployment mechanism. As CF deployment of a single function takes 40 s (plus additional time for code upload) deployment time cannot be reduced further than that level, however.

2) Automated Offloading With Application Reconfiguration: Our measurements with application reconfiguration show significant improvements compared to the redeployment mentioned above. Since there is no redeployment, the application is never suspended. Indeed, once it receives the reconfiguration commands from the Alarm Handler, it immediately swaps to the cloud or edge function according to the control realized by the Alarm Handler. The delay experienced by this control has two components, one originating from network/propagation delay between the Alarm Handler and the edge node, while the other is the result of the instrumentation overhead caused by the components interacting (the Alarm Handler and the Wrapper) during such communication. As Tab. IV attests, network delay between our on-premise edge node and the AWS region is estimated to be 20.4 ms (half of the round trip time [RTT]). We measured a total of 67.95 ms for total application control delay which contains network delay as well. Thus, in our case, instrumentation overhead is estimated to be around 47.5 ms, in which a huge constituent is HTTP, the protocol used for carrying the control messages.

After offloading computation to the cloud, the application's performance is affected by two factors: function instance cold start and edge-cloud data traversal delays. First, cloud functions need to start up, and thus they experience cold start delays which will induce an overshoot period in function invocation, execution, and consequently, application end-to-end latency. In our application's case, the approximately 1.1 s cloud cold start latency consists of three components. *i*) The default AWS Lambda cold start latency which is 200 ms as reported in Sec. IV-C. *ii*) The Wrapper overhead that is under 1 ms in our rover control application, thanks to our lightweight implementation. *iii*) The external package and resource loading takes the longest to execute as it has to load OpenCV and the machine learning models. In our case, it is approximately 0.9 s. Data traversal between the edge node and the cloud (comprising of propagation delay and AWS Lambda function call overhead) is expected to be significantly slower than in the edge-local case. As read and write operations accessing our Redis cache on the edge take less than 1 ms for data generated by our application, data traversal delay is approximately 21.5 ms. To summarize, 1.1 s after the alarm is received at the edge, the application can be considered totally reconfigured. After this point, it assumes normal operation, thus application outage in case of edge to cloud offload is also 1.1 s. The computation task can return instantaneously from the cloud to the edge. At the edge, functions are kept idle and they do not need to restart, as discussed in Sec. IV-C, which results in zero application outage time in case of moving back functions from the cloud to the edge.

Lessons learned: i) By skipping complete application redeployment, this method is able to reduce application outage significantly from around 2 minutes to a second (or even to zero). As the current reconfiguration mechanism is a simplification of switching between two different layouts, in a more general case, providing an option for joint deployment of multiple layouts (with different function groupings) can provide the benefit for higher-level fast reconfiguration. Such an extension, however, requires significant modifications in every framework modules except for the TS. *ii*) This reconfiguration delay is compatible with the rover application requirements. However, in case of more stringent requirements, several solutions may be considered to reduce this time, such as anticipating the cloud function startup before offloading the function from the edge by "warming up" instances with special invocations. *iii*) Finally, we note that inference in the cloud is expected to have a different execution time as on the edge because of different execution environments. The function will be delayed with a further 21.5 ms in normal operation because detected objects have to be written to the edge cache. Changing data store placement or using advanced data replication methods can be exploited to reduce this latency. The investigation of these options, however, is left for future studies since they are not needed by the considered rover application.

TABLE IV
APPLICATION RECONFIGURATION DELAY

Phase	Mean	Standard Deviation	Minimum	Maximum
Edge→Cloud RTT	40.83 ms	0.32	40.67 ms	42.59 ms
Limit violation detection	16.23 s	2.55	11.18 s	21.61 s
Application control	67.95 ms	6.21	63.21 ms	90.89 ms
Total	16.30 s	2.55	11.24 s	21.68 s

Fig. 9 shows the application performance during reconfiguration. We can observe that at application startup, CPU usage is around 10% while application end-to-end latency is around 1.2 s and the Inference function runs in 500–800 ms. After stressing the edge node with additional computation, we can see that the CPU load quickly rises. Then, the alarm condition is detected and the function offload to the cloud is successfully performed. CPU load is reduced accordingly at the edge and, at the same time, the invocation latency of the Inference function rises from 2 ms to around 200 ms with a short overshoot period caused by function cold start in the cloud. After function instances stabilize in the cloud, application latency assumes its steady state value around 2.3 s. The increase compared to the edge-local case is due to significantly slower execution of the Inference function in the cloud (1.2–1.3 s) as well as increased invocation latency. There are two factors affecting this function's execution time. On the one hand, images grabbed from the rover are still stored in the cache on the edge node, and the offloaded function has to read it from there. Writing inference results is also done using the same cache on the edge node. On the other hand, the difference between the edge node and cloud Lambda CPU resources contribute to worse performance. In our case, we limited the Inference function's memory usage to 1536 MB,

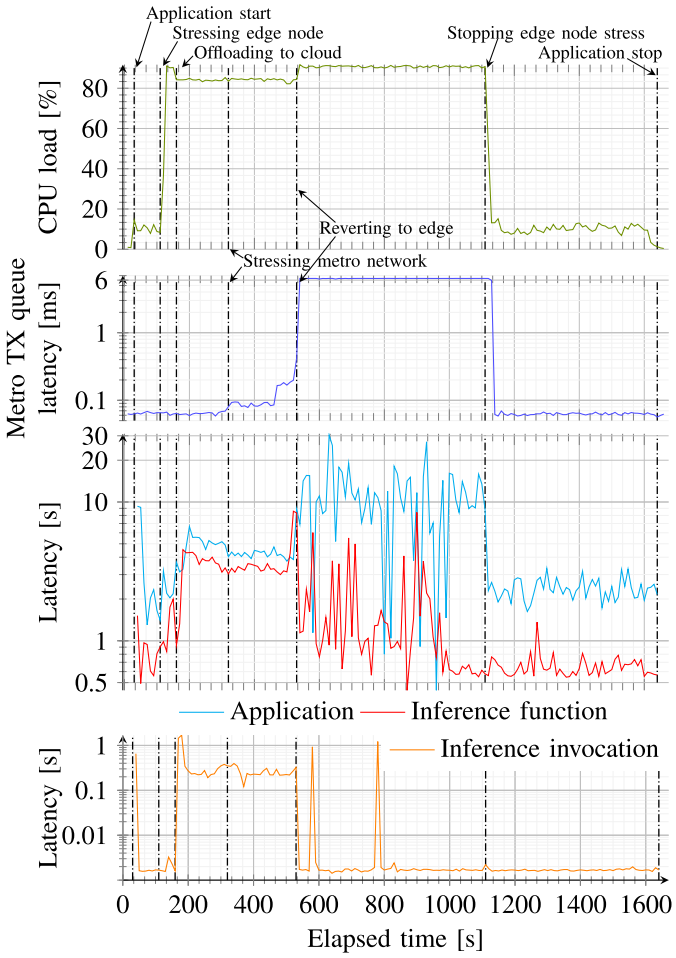

Fig. 9. Application component offloading behavior.

as it is adequate for satisfying the memory requirements of the inference process. According to our observations discussed in Sec. IV-C, the function can leverage the full potential of one CPU core at the edge. However, as reported by [41], AWS associates less than one CPU core to this setting. We note that increasing this setting might achieve better performance in the cloud. As per [45], used CPU types in the cloud are also different from the one used at the edge.

Lessons Learned: Although offloading computation to public cloud resources can be a cost-effective alternative, it has the drawback of increasing application latency, in certain cases, significantly. Better application performance might be achieved, albeit at a higher price, by offloading application components completely to another, closer, edge node or sharing requests among multiple edge nodes.

When metro network stress reaches critical congestion levels, executing the Inference function in the cloud becomes impossible any longer. When the alarm level is reached, reconfiguration to the edge takes place. Two cases may then occur. In the first case, shown in the figure, the CPU is still overloaded and no adequate computing resources are available at the edge. In this critical case, reloading the function at the edge node causes a significant decrease in application performance. We can observe that the edge node goes into complete overload and thus Greengrass cannot handle incoming requests. Although the performance of the Inference function is slightly increased, other application components cannot get as much CPU time as required, thus total end-to-end latency reaches levels as high as 30 s. In the second case, the CPU load at the edge is not overloaded when the network latency to the cloud reaches alarm conditions. In this case, the reconfiguration is successfully performed from the cloud to the edge, immediately restoring the good performance conditions originally experienced. This occurs also in the first case when turning off the stress at the edge node. As shown in Fig. 9, application performance returns to its original values with no function outage, and with low latency values successfully restored.

Lessons Learned: As our measurements attest, our rover control application was able to perform adequately when being deployed completely at the edge or when offloading Inference to the cloud. Although complete redeployment of the application introduces significant application outage, we were able to circumvent it using our fast reconfiguration option. Collecting application, infrastructure and network performance data using a dedicated framework module also proved to be a crucial aspect in proper application control.

V. Conclusion and Future Work

In this paper, we presented a framework for automated deployment and reconfiguration of latency-sensitive applications over serverless resources in a hybrid edge and AWS cloud scenario. The framework relies on three modules: the Layout and Placement Optimizer (LPO), the Serverless Deployment Engine (SDE), and the Telemetry Service (TS). In this work, we utilized a basic implementation of the LPO, while focusing on the other two modules, as main contributions. The SDE is a novel component performing serverless function deployment and enabling fast application reconfiguration as well. It fully automates a number of otherwise tedious manual operations and interacts with the provider API for fast instantiation at the edge or the AWS cloud. The TS is designed to retrieve telemetry information from the entire end-to-end computing and networking infrastructure, including CPU and memory usage at the edge, optical quality of transmission in the transport network, and queuing latency in NICs and packet switches. The TS also leverages P4-based in-band telemetry in collecting the latter. The module processes the retrieved monitoring data and provides inputs for optimal edge deployment. In addition, it detects alarm conditions and triggers fast application reconfiguration.

As further contributions, we provided detailed evaluations of AWS IoT Greengrass, the chosen edge serverless execution platform, and the SDE and TS framework modules using our machine learning-based rover driving application decomposed to latency-sensitive serverless subfunctions. We showed that Greengrass container instantiation outperforms that of Docker which justifies our selection of Greengrass as the edge execution platform. We also demonstrated the fully automated deployment of all application components to the edge resource and investigated automated offloading and reverting behavior

while putting the edge node and network under stress. While relocating the ML function with a complete redeployment can induce a minutes-long application outage, using our fast offloading mechanism can decrease this time significantly, even to zero when reusing "warmed up" function instances. These results enable the rover driving function, decomposed as FaaS, to successfully operate under extremely dynamic conditions of both edge computing and packet-optical network resources.

Our results show that the combined capabilities of the used platform and our framework are sufficient in our rover control application's case, despite the fact that AWS's FaaS services were not designed for guaranteeing good performance in low latency use-cases. However, these can prove to be inadequate for applications requiring stricter latency criteria. In our evaluation, we showed that invocation capabilities of the chosen platform can be adequate for stricter requirements in the edge-local case, whereas the performances of cloud invocation, instance startup, and alarm detection are lacking. Our main lessons learned arise exactly from these issues. First, although the FaaS execution platform promises low "cold" startup latency, using heavy libraries (machine learning in our case) can significantly increase this, resulting in slow application startup or underwhelming offloading performance. While this issue also affects other container and virtual machine services and FaaS can still outperform those when starting up instances in a reactive way, this is a significant drawback. However, preparing in advance for increased load, i.e., warming up instances in a proactive manner, can yield much better performance which can be leveraged in future works. Moreover, recent updates of AWS Lambda and Greengrass (unavailable at the time of our first evaluation) focus on ML use-cases and promise further improvements in invocation and execution performance, as well as they provide an ability to keep function instances warm even in the cloud. Our second takeaway is that collecting performance data in the cloud, specifically using Amazon CloudWatch, limits the framework's ability to detect performance limit violations, and consequently hinders proper management of applications requiring lower latency. Replacing CloudWatch in the TS module with a different option and placing it closer to the edge resource can significantly speed up the process of component offloading. As we demonstrated, complete application redeployment is out of the question when fast component offloading tasks are at hand, while in its current form, our reconfiguration option has limited capabilities. Extending our work with the ability to deploy multiple layouts together can take the best of both worlds and provide high reconfiguration flexibility with low application outage. However, this requires extensive modifications in most framework modules. In this work, we explored deployment to only a single edge node. Scenarios having multiple edge nodes with heterogeneous compute capabilities can pose interesting open questions, however. Moving application components between such resources is interesting on its own, as pointed out in Sec. II. Offloading to other edge nodes instead of utilizing public clouds as offload targets can improve application performance as well. In our case, high and low-level improvements would be necessary to handle edge node assignment and efficient communication between the edge nodes which is lacking in the current Greengrass implementation.

REFERENCES

[1] Open Networking Foundation. (2020). *P4. Open Networking Foundation*. Accessed: Jun. 4, 2020. [Online]. Available: https://p4.org/.https://p4.org/

[2] P. Bosshart *et al.*, "p4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014, doi: 10.1145/2656877.2656890.

[3] Y. Yan *et al.*, "P4-enabled smart NIC: Architecture and technology enabling sliceable optical DCS," in *Proc. 45th Eur. Conf. Opt. Commun. (ECOC)*, 2019, pp. 1–3.

[4] F. Cugini, P. Gunning, F. Paolucci, P. Castoldi, and A. Lord, "P4 in-band telemetry (INT) for latency-aware VNF in metro networks," in *Proc. Opt. Fiber Commun. Conf. (OFC)*, 2019, p. M3Z.6.

[5] F. Paolucci, A. Sgambelluri, F. Cugini, and P. Castoldi, "Network telemetry streaming services in SDN-based disaggregated optical networks," *J. Lightw. Technol.*, vol. 36, no. 15, pp. 3142–3149, Aug. 1, 2018.

[6] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *Proc. IEEE Int. Conf. Cloud Eng. (IC2E)*, Apr. 2018, pp. 159–169.

[7] I. Pelle, F. Paolucci, B. Sonkoly, and F. Cugini, "Telemetry-driven optical 5G serverless architecture for latency-sensitive edge computing," in *Proc. Opt. Fiber Commun. Conf. (OFC)*, 2020, pp. 1–3.

[8] X. Wang, Y. Ji, J. Zhang, L. Bai, and M. Zhang, "Low-latency oriented network planning for MEC-enabled WDM-PON based fiber-wireless access networks," *IEEE Access*, vol. 7, pp. 183383–183395, 2019.

[9] H. A. Alameddine, S. Sharafeddine, S. Sebbah, S. Ayoubi, and C. Assi, "Dynamic task offloading and scheduling for low-latency IoT services in multi-access edge computing," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 3, pp. 668–682, Mar. 2019.

[10] D. A. Chekired, M. A. Togou, L. Khoukhi, and A. Ksentini, "5G-slicing-enabled scalable SDN core network: Toward an ultra-low latency of autonomous driving service," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 8, pp. 1769–1782, Aug. 2019.

[11] F. Paolucci, F. Civerchia, A. Sgambelluri, A. Giorgetti, F. Cugini, and P. Castoldi, "P4 edge node enabling stateful traffic engineering and cyber security," *J. Opt. Commun. Netw.*, vol. 11, no. 1, pp. A84–A95, Jan. 2019.

[12] Z. Liu, J. Bi, Y. Zhou, Y. Wang, and Y. Lin, "NetVision: Towards network telemetry as a service," in *Proc. IEEE 26th Int. Conf. Netw. Protocols (ICNP)*, Sep. 2018, pp. 247–248.

[13] T. Pan *et al.*, "INT-path: Towards optimal path planning for in-band network-wide telemetry," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2019, pp. 487–495.

[14] R. Hohemberger *et al.*, "Orchestrating in-band data plane telemetry with machine learning," *IEEE Commun. Lett.*, vol. 23, no. 12, pp. 2247–2251, Dec. 2019.

[15] S. Xu *et al.*, "Emergency OPM recreation and telemetry for disaster recovery in optical networks," *J. Lightw. Technol.*, vol. 38, no. 9, pp. 2656–2668, May 1, 2020.

[16] The Apache Software Foundation, Apache OpenWhisk. (2016). *The Apache Software Foundation*. Accessed: Jun. 4, 2020. [Online]. Available: https://openwhisk.apache.org. https://openwhisk.apache.org

[17] The Knative Authors, Knative. (2020). *Knative*. Accessed: Jun. 4, 2020. [Online]. Available: https://knative.dev

[18] OpenFaaS Ltd. (2020). *OpenFaaS—Serverless Functions Made Simple*. Accessed: Jun. 4, 2020. [Online]. Available: https://www.openfaas.com/

[19] (2020). *Kubeless*. Accessed: Jun. 4, 2020. [Online]. Available: https://kubeless.io/

[20] A. Palade, A. Kazmi, and S. Clarke, "An evaluation of open source serverless computing frameworks support at the edge," in *Proc. IEEE World Congr. Services (SERVICES)*, Jul. 2019, pp. 206–211.

[21] L. Baresi and D. Filgueira Mendonca, "Towards a serverless platform for edge computing," in *Proc. IEEE Int. Conf. Fog Comput. (ICFC)*, Jun. 2019, pp. 1–10.

[22] W3C WebAssembly Community Group. (2020). *WebAssembly*. Accessed: Jun. 4, 2020. [Online]. Available: https://webassembly.org/

[23] P. K. Gadepalli, G. Peach, L. Cherkasova, R. Aitken, and G. Parmer, "Challenges and opportunities for efficient serverless computing at the edge," in *Proc. 38th Symp. Reliable Distrib. Syst. (SRDS)*, Oct. 2019, pp. 261–2615.

[24] NDN Consortium. (2020). *Named Data Networking*. Accessed: Jun. 4, 2020. [Online]. Available: https://named-data.net/

[25] M. Król and I. Psaras, "Nfaas: Named function as a service," in *Proc. 4th ACM Conf. Inf.-Centric Netw. (ICN)*. New York, NY, USA: Association for Computing Machinery, 2017, pp. 134–144, doi: 10.1145/3125719.3125727.

[26] D. Bermbach, S. Maghsudi, J. Hasenburg, and T. Pfandzelter, "Towards auction-based function placement in serverless fog platforms," in *Proc. IEEE Int. Conf. Fog Comput. (ICFC)*, Apr. 2020, pp. 25–31.

[27] A. Buzachis, M. Fazio, A. Celesti, and M. Villari, "Osmotic flow deployment leveraging faas capabilities," in *Internet and Distributed Computing Systems*, R. Montella, A. Ciaramella, G. Fortino, A. Guerrieri, and A. Liotta, Eds. Cham, Switzerland: Springer, 2019, pp. 391–401.

[28] C. Cicconetti, M. Conti, and A. Passarella, "Architecture and performance evaluation of distributed computation offloading in edge computing," *Simul. Model. Pract. Theory*, vol. 101, May 2020, Art. no. 102007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1569190X19301406

[29] Microsoft Corporation. (2020). *Azure Edge Zone*. Accessed: Jun. 4, 2020. [Online]. Available: https://docs.microsoft.com/en-us/azure/networking/edge-zonesoverview

[30] Amazon Web Services. (2020). *Lambda@Edge*. Accessed: Jun. 4, 2020. [Online]. Available: https://aws.amazon.com/lambda/edge/

[31] Amazon Web Services. (2020). *AWS Local Zones*. Accessed: Jun. 4, 2020. [Online]. Available: https://aws.amazon.com/about-aws/global-infrastructure/localzones/

[32] Amazon Web Services. (2020). *AWS Wavelength*. Accessed: Jun. 4, 2020. [Online]. Available: https://aws.amazon.com/wavelength/

[33] LLC Google. (2020). *Google Global Mobile Edge Cloud*. Accessed: Jun. 4, 2020. [Online]. Available: https://cloud.google.com/press-releases/2020/0305/google-cloud-telco-strategy

[34] Microsoft Corporation. (2020). *Azure Functions Serverless Compute*. Accessed: Jun. 4, 2020. [Online]. Available: https://azure.microsoft.com/en-us/services/functions/

[35] Amazon Web Services. (2020). *AWS Lambda*. Accessed: Jun. 4, 2020. [Online]. Available: https://aws.amazon.com/lambda/

[36] LLC Google. (2020). *Cloud Functions*. Accessed: Jun. 4, 2020. [Online]. Available: https://cloud.google.com/functions

[37] International Business Machines Corporation. (2020). *IBM Cloud Functions*. Accessed: Jun. 4, 2020. [Online]. Available: https://cloud.ibm.com/functions/

[38] Microsoft Corporation. (2016). Azure IoT Edge. The Apache Software Foundation. Accessed: Jun. 4, 2020. [Online]. Available: https://azure.microsoft.com/en-us/services/iot-edge/

[39] Moby Project. (2017). Moby. Moby Project. Accessed: Jun. 4, 2020. [Online]. Available: https://mobyproject.org/

[40] Amazon Web Services. (2018). *Firecracker*. Accessed: Jun. 4, 2020. [Online]. Available: https://firecrackermicrovm.github.io/

[41] I. Pelle, J. Czentye, J. Doka, and B. Sonkoly, "Towards latency sensitive cloud native applications: A performance study on AWS," in *Proc. IEEE 12th Int. Conf. Cloud Comput. (CLOUD)*, Jul. 2019, pp. 272–280.

[42] Amazon Web Services. (2020). *AWS IoT Greengrass*. Accessed: Jun. 4, 2020. [Online]. Available: https://aws.amazon.com/greengrass/

[43] Amazon Web Services. (2020). *AWS CloudFormation–Infrastructure as Code & AWS Resource Provisioning*. Accessed: Jun. 4, 2020. [Online]. Available: https://aws.amazon.com/cloudformation/

[44] H. Oomori, T. Matsui, Y. Tanaka, H. Tanaka, and E. Tsumura, "Compact optical transceiver 'CFP4' for 100 Gbit/s network systems," *SEI Tech. Rev.*, no. 82, pp. 112–116, Apr. 2016.

[45] J. Czentye, I. Pelle, A. Kern, B. P. Gero, L. Toka, and B. Sonkoly, "Optimizing latency sensitive applications for Amazon's public cloud platform," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2019, pp. 1–7.

[46] Pallets. (2010). *Flask*. Accessed: May 28, 2020. [Online]. Available: https://flask.palletsprojects.com

[47] OpenCV Team. (2020). *OpenCV*. Accessed: May 11, 2020. [Online]. Available: https://opencv.org. https://opencv.org

[48] (2020). *Docker*. Accessed: Jun. 4, 2020. [Online]. Available: https://www.docker.com/

István Pelle received the M.Sc. degree in computer science from the Budapest University of Technology and Economics (BME) in 2015, where he is currently pursuing the Ph.D. degree. He is currently a member of the MTA-BME Network Softwarization Research Group, where he works on cloud and edge computing with a special focus on services leveraging the serverless concept and Amazon Web Services.

Francesco Paolucci received the Laurea degree in telecommunications engineering from the University of Pisa in 2002 and the Ph.D. degree from Scuola Superiore Sant'Anna, Pisa, in 2009. He is currently a Senior Researcher with CNIT, Pisa, Italy. In 2008, he was granted a Research Merit Scholarship with the Istitut National de le Recherche Scientifique (INRS), Montreal, QC, Canada. He has been involved in many European research projects on next generation control networking (E-Photon/ONe+, BONE, NOBEL, STRONGEST, IDEALIST, PACE, 5GEx, 5GTRANSFORMER, METROHAUL, 5Growth, and BRAINE). He has coauthored two IETF Internet Drafts, more than 160 publications in international journals, conference proceedings, and book chapters, and filed four international patents. His main research interests include network control plane, orchestration for edge/cloud platforms, traffic engineering, network disaggregation, advanced network telemetry, and SDN data plane programmability.

Balázs Sonkoly received the M.Sc. and Ph.D. degrees in computer science from the Budapest University of Technology and Economics (BME), in 2002 and 2010, respectively. He is currently an Associate Professor with BME. He is also the Head of the MTA-BME Network Softwarization Research Group. He has participated in several EU projects (FP7 OpenLab, FP7 UNIFY, and H2020 5G Exchange) and national projects. His current research interests include cloud/edge/fog computing, NFV, SDN, and 5G. He was the Demo Co-Chair of ACM SIGCOMM 2018, EWSDN'15,'14, and IEEE HPSR'15.

Filippo Cugini is currently the Head of Research Area with CNIT, Pisa, Italy. He is the co-author of 14 patents and more than 250 international publications. His main research interests include theoretical and experimental studies in the field of communications and networking.