Operating Latency Sensitive Applications on Public Serverless Edge Cloud Platforms

István Pelle, *Member, IEEE*, János Czentye, *Graduate Student Member, IEEE*, János Dóka, András Kern, Balázs P. Gerő, *Associate Member, IEEE*, and Balázs Sonkoly, *Associate Member, IEEE*

***Abstract*—Cloud native programming and serverless architectures provide a novel way of software development and operation. A new generation of applications can be realized with features never seen before while the burden on developers and operators will be reduced significantly. However, latency sensitive applications, such as various distributed IoT services, generally do not fit in well with the new concepts and today's platforms. In this article, we adapt the cloud native approach and related operating techniques for latency sensitive IoT applications operated on public serverless platforms. We argue that solely adding cloud resources to the edge is not enough and other mechanisms and operation layers are required to achieve the desired level of quality. Our contribution is threefold. First, we propose a novel system on top of a public serverless edge cloud platform, which can dynamically optimize and deploy the microservice-based software layout based on live performance measurements. We add two control loops and the corresponding mechanisms which are responsible for the online reoptimization at different timescales. The first one addresses the steady-state operation, while the second one provides fast latency control by directly reconfiguring the serverless runtime environments. Second, we apply our general concepts to one of today's most widely used and versatile public cloud platforms, namely, Amazon's AWS, and its edge extension for IoT applications, called Greengrass. Third, we characterize the main operation phases and evaluate the overall performance of the system. We analyze the performance characteristics of the two control loops and investigate different implementation options.**

***Index Terms*—Amazon Web Services (AWS), cloud, edge, greengrass, IoT, lambda, serverless.**

I. Introduction

CLOUD native programming, microservices and serverless architectures provide a novel way of software development and operation. A new generation of applications with features never seen before is promised, while the burden on developers and application providers is reduced or more exactly, shifted toward the cloud operators. On-demand vertical and horizontal resource scaling in an arbitrary scale, dependability, fault tolerant operation, controlled resiliency are just highlighted features provided inherently by cloud platforms. However, latency sensitive applications with strict delay constraints, such as several distributed IoT services, generally do not fit in well with the new concepts and today's platforms and pose additional challenges to the underlying systems. When strict delay bounds are defined between different components of a microservice-based software product, or between a software element and the end device, novel mechanisms and concepts are needed. A crucial first step toward the envisioned future services is to move compute resources closer to customers and end devices. Edge, fog, and mobile edge computing [30], [31], [37], [38] address this extension of traditional cloud computing. Nevertheless, solely adding cloud resources to the edge is not enough as the cloud platform itself could significantly contribute to the end-to-end delay depending on the internal operations, involved techniques and configurations.

In this article, we adapt some relevant aspects of the cloud native approach and related operating techniques for latency sensitive IoT applications operated on public cloud platforms extended with edge resources. Our general design concepts are applied to one of today's most widely used and versatile public cloud platforms, namely, Amazon Web Services (AWS) [1], and its serverless services. We identify the missing components, including novel mechanisms and operation layers, required to achieve the desired level of service quality. More precisely, we focus on serverless architectures and the Function as a Service (FaaS) cloud computing model where the microservice-based application is built from isolated functions which are deployed and scaled separately by the cloud platform. In our previous work [7], we proposed a novel mechanism to optimize the software "layout," i.e., to minimize the deployment costs, in central cloud environment, e.g., in a given AWS region, while meeting the average latency constraints defined on the application. A dedicated component is responsible for composing the service by selecting the preferred building blocks, such as runtime flavors (defining the amount of resources to be assigned) and data stores, and the optimal grouping of constituent functions and libraries which are packaged into respective FaaS platform artifacts. This approach can be extended to edge cloud infrastructures

Manuscript received July 10, 2020; revised September 30, 2020 and November 6, 2020; accepted November 23, 2020. Date of publication December 3, 2020; date of current version May 7, 2021. This work was supported in part by the AWS Cloud Credits for Research Program. *(Corresponding author: Balazs Sonkoly.)*

István Pelle, János Czentye, János Dóka, and Balázs Sonkoly are with the MTA-BME Network Softwarization Research Group, 1117 Budapest, Hungary, and also with the Department of Telecommunications and Media Informatics, Faculty of Electrical Engineering and Informatics, Budapest University of Technology and Economics, 1117 Budapest, Hungary (e-mail: pelle@tmit.bme.hu; czentye@tmit.bme.hu; janos.doka@tmit.bme.hu; sonkoly@tmit.bme.hu).

András Kern and Balázs P. Gero are with Ericsson Research, 1117 Budapest, Hungary (e-mail: andras.kern@ericsson.com; balazs.peter.gero@ericsson.com).

Digital Object Identifier 10.1109/JIOT.2020.3042428

but further considerations are necessary. More specifically, Amazon provides an edge extension for IoT services, called Greengrass, where the edge infrastructure nodes are owned and maintained by the user (or application provider) but managed by AWS. Obviously, the pricing scheme and the performance characteristics of serverless components in this realm is totally different from the regular billing policy and operation, therefore, our models should be adjusted accordingly. In this article, we aim to extend our basic model for edge cloud platforms and to enable dynamic and automated application (re-)deployment if an online platform monitoring module triggers that.

Our contribution is threefold.

1) We propose a novel system on top of public cloud platforms extended with edge resources which can dynamically optimize and deploy applications, following the microservice software architecture, based on live performance measurements. We add two different control loops and the corresponding mechanisms which are responsible for the online reoptimization of the software layout and constituent modules at different timescales. The first one addresses the control of the steady-state, long-term operation of given applications and it is suitable for following, e.g., the daily profiles, while the second one implements a more responsive control loop which can directly reconfigure the runtime environments of deployed functions if the monitoring system triggers that as a response to, e.g., SLA violation.
2) We provide a proof-of-concept prototype. In this article, we target AWS and its edge extension for IoT applications, called Greengrass, however, the concept is general and it can be applied to other public cloud environments as well. Our current solution supports geographically distributed edge cloud infrastructures under the low-level control of AWS. The system encompasses a layout and placement optimizer (LPO), a serverless deployment engine (SDE) and a live monitoring system with dedicated components and operation workflows.
3) We characterize the main operation phases and conduct several experiments and simulations to evaluate the overall performance of the system. We analyze the performance characteristics of the two control loops as well and investigate different implementation options. Finally, we reveal further challenges and open issues.

The remainder of this article is organized as follows. In Section II, the background is introduced and a brief summary on related works is provided. In Section III, an illustrative use case is defined which motivated our work. Section IV highlights the main principles driving our system design and presents the high level architecture of the system. In Section V, the proposed models related to the applications and the underlying platforms are presented, and the optimization problem is formulated. Section VI is devoted to the proposed system including the details of the relevant components. In Section VII, we evaluate the performance of the overall system and our main findings are discussed in detail. Finally, Section VIII concludes this article.

II. Background and Related Work

The cloud native paradigm aims to build and run applications exploiting all the benefits of the cloud computing service models. It includes several techniques and concepts, from microservices across DevOps to serverless and FaaS architectures, and everyone defines that in a slightly different way. According to the cloud native computing foundation (CNCF) [6], the ultimate goal is an open source, microservice-based software stack, where distinct containers are separately orchestrated and scaled by the cloud platform enabling the optimal resource utilization and agile development. The serverless approach allows to shift the focus from "where to deploy" to "how to create" the applications. It can be realized by following either the Container as a Service (CaaS) computing model or the FaaS paradigm, depending on the granularity level that the developer can consider when creating the software. In this article, we focus on the latter approach because it provides finer granularity in the organization of the application and more opportunities for optimization. There are several public cloud providers offering both services, such as Amazon [1], Google [13], Microsoft [24] or IBM [15], and a number of open source platforms are also available for private deployments, such as Kubernetes [18], Knative [17], OpenWhisk [2], or OpenFaaS [27]. This section provides a brief introduction on Amazon's serverless solutions over cloud and edge domains. Tools for automated deployment of serverless components fostering the development and operation of such applications are also highlighted together with open issues.

A. Serverless on Amazon Web Services

AWS [1], the platform of the market leader public cloud provider, offers a wide selection of services that can support building applications in the cloud. Among those, two options are adequate for executing serverless code: elastic container service (ECS) with the Fargate launch type, and Lambda which is a FaaS solution. Both can ease the task of deploying application components in different ways providing diverse configuration options and pricing models. Lambda offers fewer options for configuration but at the same time it simplifies automatic deployment and connecting other AWS services or Lambda functions. The service increases the assigned CPU power together with the only adjustable flavor parameter, available memory size. Instance startup, load balancing between the instances and networking configuration is taken care of by the Lambda framework without any need for developer interaction. There is also a select set of AWS services that have built-in triggers for Lambda, while other, third party services can invoke Lambda functions via the software development kit (SDK). Lambda defines methods for easy versioning and branching of deployed functions via Lambda versions and aliases. Compared to Lambda, ECS offers more options for setting up resources and networking, while also providing possibilities for quick invocations, however, it lacks the automatic load balancing options. Larger sized function code and related artifacts are better suited for deployment with ECS, since Lambda poses a 250-MB size limit of uncompressed packages.

AWS's CloudFormation service [3] provides possibilities for automating the deployment process of application components realized by either of these services. However, code deployment to edge nodes is only available via AWS IoT Greengrass.

B. Serverless at the Edge With AWS IoT Greengrass

AWS IoT Greengrass is a service that is part of AWS's IoT offerings and its main task is to make AWS Lambda functions available on edge devices. The service's basic building blocks are *Groups* that can be configured in the cloud and their deployment is managed by AWS. They are the collections of different entities serving different roles. The 1) *Core* is at the center of each group that has a two-pronged representation. It is present in the cloud as a link to the edge node while it is also a software instance running on the edge node and handles communication with the cloud. Every message flowing between the edge and the cloud is encoded using RSA keys for which an X.509 certificate is used. This also has to be set up in the cloud, and assigned to the Core, as well as transferred to the edge node before starting up the Core software; 2) *Devices and local resources* [e.g., devices connected via USB or machine learning (ML) artifacts] serve as inputs; for 3) *edge Lambda functions* which are linked to cloud Lambdas via AWS Lambda aliases. Configuration of edge and cloud functions is handled separately which enables extensions upon AWS Lambda functionality. Although code size limits are inherited from the cloud version, edge functions do not have lower or upper values in their memory settings and increments can be made in 1-kB steps, as opposed to the cloud version's 64-MB steps. A remarkable difference compared to on-demand cloud Lambdas is that edge functions can be long-lived (*pinned* in AWS terminology) and the single long-lived function instance can be kept running indefinitely. On-demand edge functions are handled by the Core similarly to cloud functions, multiple instances of a single function can run concurrently and they are stopped after reaching the configured timeout value. Three containerization method is offered for executing edge functions: a) Greengrass; b) Docker; or c) no containerization. The first option is the most versatile while the rest severely limit available functionality. Access to other edge and cloud functions is granted; and via 4) *subscriptions*.

C. Automated Serverless Deployment and Optimization

Deploying cloud applications across different platform services is a complex task. In order to ease this process, multiple tools exist that are able to set up required resources with different cloud service providers. For example, the *Serverless Framework* [35] uses a YAML configuration file to declare resources in a provider agnostic way and, with its own CLI, provides an interface for managing these resources. The service is able to cooperate with, e.g., AWS [1], Microsoft Azure [24], Google Cloud Platform [13], and Apache OpenWhisk [2]. *Terraform* [34] is a similar tool that enables setting up and managing cloud infrastructure spanning over multiple public cloud domains. The higher level, provider agnostic interface makes it easier to move the infrastructure from one provider to the next but it cannot fully hide provider specific parameters. These tools were designed to receive external parameters to be used at deployment from other services, e.g., for specifying resource types or memory size. One such external service is *Densify* [9] that, leveraging its separate optimization and monitoring components, makes cloud applications self-aware. It monitors AWS virtual machine (EC2) instances with a proprietary monitoring component and collects CPU, memory and network utilization data. Based on these, the optimization component uses ML to model the application's utilization patterns while also estimating the best fit of compute resources for current needs and predefined specifications. Such estimations can give recommendations on instance flavors to be used and on the number of such instances. These recommendations can be forwarded to application maintainers via different channels (e.g., Slack or email), or can be applied automatically. Such automatic redeployments can happen using templating tools that support dynamic parameter assignment or parameter stores, e.g., AWS CloudFormation, Terraform or Ansible. The service enhances change recommendations with a cost monitoring interface as well. As for AWS specific deployment options, the provider offers different services for managing resources. All of them use the same AWS API but they provide different levels of complexity. Low-level options, such as the Web console, the SDKs and the CLI have smaller granularity thus they make handling of applications containing multiple resources overly complex. *CloudFormation* [3] (that is also used by the AWS Cloud Development Kit and many third party options) can treat a whole deployment as a unit of workload. It can handle the setup, modification and deletion tasks of complex applications (called stacks or stack sets in CloudFormation terminology) using its own templating language. *Stackery* [33] is a set of development and operations tools accelerating serverless deployments on top of AWS. It supports the management of production serverless applications throughout their life cycle.

Albeit the availability of these versatile tools in deployment, they do not prove to be adequate for deploying applications to hybrid edge cloud scenarios when latency is of concern. While AWS tools offer edge node management, and the AWS Compute Optimizer [4] serves as a recommendation engine to help right-sizing EC2 instances, such an optimization engine for serverless applications is not available. AWS independent tools share a similarity in this regard, as they do not venture into the serverless domain and consider resource utilization but omit the investigation of application performance. Additionally, they usually lack the capability of handling edge resources altogether or have started to support this feature only recently thus not covering yet the full feature set made accessible by the cloud provider.

Besides the tools supporting deployment and orchestration over cloud platforms, there are only a few papers in the literature dealing with cloud native and cost-aware service modeling and composition. Eismann *et al.* [10], Fotouhi *et al.* [12], Leitner *et al.* [20], and Winzinger and Wirtz [36] provided pricing models for microservice-based application deployment over public clouds, but they focus only on supporting offline cost analysis for predefined deployment scenarios. Online cost tracing of a serverless application is a cumbersome task

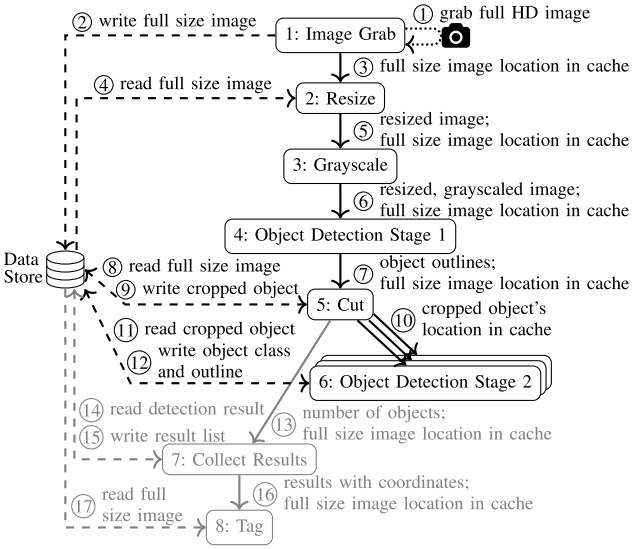


Fig. 1. Object detection use case.

due to limited billing information provided by the cloud platforms. To tackle this issue, *Costradamus* [19] realizes a per request cost tracing system using a fine-grained cost model for deployed cloud services, however, it lacks any optimization features. Researchers in [11] and [21] studied the optimization problem of cloud native service composition and provide offline solutions based on game theoretic formulation and a constrained shortest path problem. Other recent works in [5], [8], and [22] target similar problems of performance optimization of serverless applications leveraging public cloud resources, but only regarding the placement problem of the service components and missing any adaptive and automated service reoptimization task.

III. Targeted Use Case

In this section, we highlight an envisioned use case motivating our work. The application exploits cloud features and serverless tools in order to provide IoT services at large scale. The use case, presented in Fig. 1, addresses live object detection on Full HD video streams. As we target cloud native deployment and follow the serverless approach, we have stateless functions requiring all data as input. Therefore, making use of dedicated data stores is the reasonable (or the only feasible) way of data exchange. Here, we strive to decrease bandwidth requirements by preprocessing images before submitting them to elaboration and finally marking them with detailed object classification results. The preprocessing stage in steps ①–⑩ resizes and grayscales captured video frames and performs a preliminary object detection on the modified picture. At the end of the preprocessing stage, the full size image is cut into pieces based on the bounding boxes provided by the preliminary object detection. As a next step, the *Cut* function calls the second stage object detection function for each cropped image which performs the object classification task. Observe that the number of calls depends on how many objects we found during the preprocessing stage which we consider as an application specific metric. It depends on the software whether the calls are synchronous and invoked serially or asynchronous and handled in parallel. Consequently, the implementation of the next function, *Collect Results*, could be different for the two approaches. In any case, it awaits while each second stage detection function finishes and collects their individual results. Finally, this function calls the *Tag* function that marks detected objects on the full size image and annotates it with object classification results. For our use case, we interpret the end-to-end (E2E) latency as the average elapsed time between the arrival of a frame and the event when a recognized object's classification is written out into the data store.

In our implementation, we used Python and leveraged features of the OpenCV [26] library for image processing and object detection steps, relying on its deep neural networks module and the MobileNet-SSD network. In the remainder of this article, we focus on steps ①–⑫, the main parts of the application, and the components related to the rest of the steps are not deployed in our tests.

IV. System Design

This section is devoted to the main goals and principles driving our architecture design and the high level system description is also provided.

A. Design Goals

Our main goal is to foster the development and operation of latency sensitive IoT applications by adapting the cloud native paradigm. More specifically, we aim at improving latency control for serverless applications and allowing optimization of operation costs on public cloud platforms extended with privately owned edge infrastructures. We focus on the FaaS cloud computing model, however, the concepts are general and can be applied to container-based serverless solutions as well (such as Fargate containers or Kubernetes pods). Although, the finer granularity in the construction of the application provided by the FaaS approach yields more optimization options and requires more sophisticated solutions. Formally, the operation cost of the application is to be minimized by finding the cost optimal software layout required to meet the average latency bounds. To enable this optimization, we need to construct accurate application and platform models capturing the performance characteristics and operation prices. The first reasonable way of controlling latency is the careful placement of software components: the functions can be run in the central cloud or in available edge domains. Current APIs of today's systems typically do not provide sophisticated placement control based on delay information, therefore, we strive to explicitly select the domains to run the functions. We assume that edge resources are scarce, following different cost models as cloud resources, and the preferred deployment option is always the central cloud while edge resources are used only if the delay constraints require that. We argue that besides placement, the efficient grouping of constituent functions and libraries, which will be packaged into respective FaaS platform artifacts, and the selection of the runtime flavors are crucial tasks which significantly affect both the

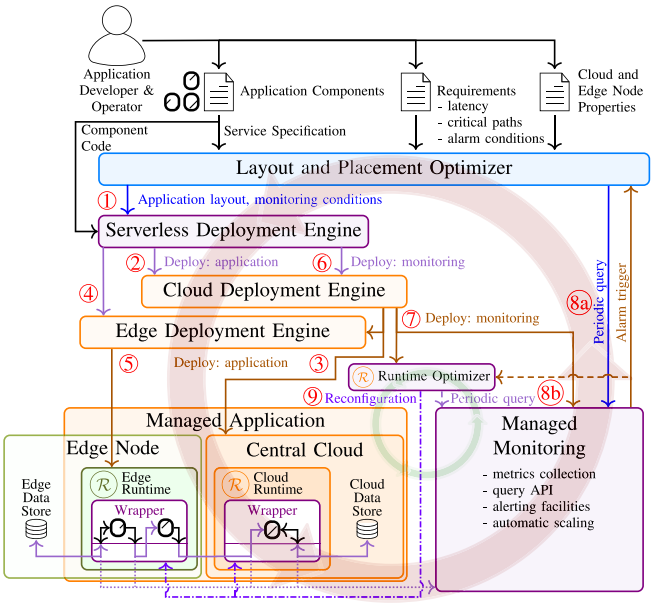


Fig. 2. High level system architecture.

performance (e.g., end-to-end latency) and the operation costs. A top level component is able to address all these targets and generate a software layout description including the function grouping with the selected flavors and placement information. Based on this general description, an adapter layer can directly deploy the application to the underlying cloud infrastructure while exploiting the exposed APIs and related cloud services.

As user demands, application characteristics and platform performance can vary in time, dynamic reoptimization is an essential feature which can be provided based on a versatile monitoring system. We target such a system making use of available cloud services and custom extensions. Two different approaches are considered to implement control loops. The first option is to realize a full reoptimization cycle starting with a model update gathered from live measurements, followed by the optimization task and the full redeployment of the application. Obviously, this yields a larger operation timescale. In order to ameliorate the response time, we address an alternative option as well, which realizes a shorter control loop. If different deployment options are onboarded in advance, the reconfiguration of the application can be executed much faster. However, we need to add a dedicated component to control the specific application based on monitored metrics, while a customized version of the FaaS runtime is also required in order to allow on-the-fly reconfiguration.

B. High Level Architecture and Operation

The high level architecture of the proposed system is depicted in Fig. 2. The system is capable of composing, deploying and dynamically reoptimizing IoT applications operated on serverless resources. We note, that the first and basic version of the system, without any support for the edge, was introduced in [7] and [29]. In the former, we focused on the optimization layer, while in the latter, we investigated deployment tasks. In the current work, we leverage their composition and extend upon it with support for edge deployment and a second option for inducing changes in the application layout.

At the top level, the Layout and Placement Optimizer (LPO) receives input data from the developer (or the operator in other scenarios). The data consists of the application model (Application Components with Requirements) and the platform model (Cloud and Edge Node Properties). The graph-based service model encompasses functions, data stores (as nodes) and invocations (function calls), read, write operations (as edges). Average function execution time, call rates, latency requirements on critical paths, etc. can also be defined for the service. The other input of the system is the platform model which describes the cloud platform's performance and pricing schemes and the list of available edge nodes with their properties. It can be given *a priori* based on previous measurements, however, the model parameters can be adjusted on-the-fly based on live monitoring. The LPO works with these service- and platform agnostic abstract models and constructs an optimal application layout by grouping the functions into deployable units (e.g., FaaS artifacts), defining the corresponding minimal flavors together with the hosting domains (central cloud versus edge) and determining the required data stores and invocation techniques (e.g., one for invoking functions on the edge, a different one for calling functions in the central cloud). The main objective is to minimize the operation costs while meeting the average latency bounds given by the developer or user. The application layout together with monitoring conditions is passed to the *Serverless Deployment Engine* (SDE) in step ① that transforms incoming data into platform specific API calls and adapts the application layout to the underlying edge or central cloud environments. In today's systems (such as AWS), the central cloud and edge domains are controlled via distinct deployment engines (*Cloud/Edge Deployment Engines* on Fig. 2) and APIs in separate calls (steps ② and ④).

As a result, the *Managed Application* can have parts running on edge nodes or in the central cloud launched in steps ③ and ⑤, respectively. We assume that the platform can run the same function artifacts in both runtime environments and in-memory data stores can be used for state management. In either case, the grouped application components are executed by our special-built *Wrapper* which is an essential extension to the platform's own runtime environment. The purpose of the Wrapper is threefold.

1) It enables grouping of functions into artifacts by handling both the internal interactions among the encompassed functions and the interactions with the outside world: state store access and invocation to other components.
2) The Wrapper logs measured metrics on these operations, including platform related and application specific ones, to the *managed monitoring system*.
3) The Wrapper grants on-the-fly reconfiguration access to the runtime environment via a novel API which is used by the runtime optimizer (RO), the controller of the shorter control loop (step ⑨). This reconfiguration allows to change the function calls (e.g., invoking

TABLE I
MATHEMATICAL NOTATIONS USED IN THIS ARTICLE

Notation	Description
$S = (V, A)$	Service model as a multidigraph
$F, D \subseteq V$	Function and data store nodes
$\wp \in V$	Dedicated node representing the platform
$R, W, I \subseteq A$	Read, write and invocation arcs
$S[F_\wp] = (F \cup \{\wp\}, I)$	Induced subdigraph of invocations
$u \xrightarrow{*} v : u, v \in F$	Directed path (chain) between nodes u, v
$\Pi : \{\langle \pi_s \xrightarrow{*} \pi_e \rangle\}$	Set of node-disjoint critical paths in $S[F]$
$\pi_s, \pi_e \in F$	First and last node of critical path $\pi \in \Pi$
$l_\pi : \Pi \mapsto \mathbb{R}^+$	Assigned latency limit of critical path $\pi \in \Pi$
$\Phi = \Phi_E \cup \Phi_\lambda$	Set of edge and Lambda flavors
$T_s \in \mathbb{R}^+$	Overall service running time
$\tau : F \mapsto \mathbb{R}^+$	Function execution time measured on one core
$\omega_r : A \mapsto \mathbb{R}^+$	Average request rate of an arc
$\ell_m : \Phi \mapsto \mathbb{R}^+$	Assigned memory (MB) of a flavor
$n_c : \Phi \mapsto (0, 1]$	Provided fraction of one vCPU core
$\delta_\Phi : A, \Phi \mapsto \mathbb{R}_0^+$	Flavor-related mean delay of an arc
$A^+ : P_F \mapsto A$	Egress arcs $\{uv \in A \mid u \in p, v \notin p, p \in P_F\}$
$P_F \subseteq \mathcal{P}(F)$	Function partitioning ($\mathcal{P}$ denotes the power set)
$i_f, i_p \in A$	Ingress arc of function $f \in F$, group $p \in P_F$
$t_p : P_F \times \mathcal{F} \mapsto \mathbb{R}^+$	Execution time of group $p \in P_F$
$r_p : P_F \mapsto \mathbb{R}_0^+$	Summed requests of group $p \in P_F$
$c_p, \bar{l}_p : P_F \times \Phi \mapsto \mathbb{R}^+$	Cost and latency function of group $p \in P_F$
$C_r, C_p, C_i \in \mathbb{R}_0^+$	Billing constants
$\varphi : P_F \mapsto \Phi$	Flavor assignment of partition groups
$\mathbf{C}, \mathbf{L}, \mathbf{K}, \mathbf{B}, \mathbf{F}, \mathbf{D}, \mathbf{T}, \mathbf{P}$	Structures for dynamic programming
$B \in \mathbb{N}$	Number of subcase latency bounds
$\mathbf{C}[u \xrightarrow{*} v]$, $\mathbf{L}[u \xrightarrow{*} v]$	Cost and latency value of the chain partitioning
$\mathcal{L}(v) = \{u \mid v \xrightarrow{*} u\}$	Reachable leaves $\{u \in F \mid deg^+(u) = 0\}$
$N^+(v), N_c^+(u \xrightarrow{*} v)$	Out-neighbourhood of node v, chain $u \xrightarrow{*} v$

the central cloud version of a function instead of the edge variant) or data store access in the artifact based on live monitoring without the need of redeployment. The monitoring infrastructure, consisting of the managed monitoring system and the RO, is deployed in steps ⑥ and ⑦ when the application has already been set up. The monitoring system aims at monitoring performance and application level metrics and it can send alarms to the LPO and the RO. In addition, a periodic query-based operation is also provided to support enhanced responsiveness (steps ⑧a and ⑧b).

V. OUR MODELS AND OPTIMIZATION PROBLEM

In this section, we define our service and platform models capturing the main performance and cost characteristics. The introduced notations are summarized in Table I. To establish accurate models, a comprehensive performance analysis of AWS Lambda and Greengrass is the essential first step.

A. Performance of AWS Lambda

In our previous works [7], [28], we provided a comprehensive performance study of delay characteristics of AWS FaaS and CaaS offerings, based on short- and long-term experiments. Here, we give a summary on them focusing on our main findings with regards to AWS Lambda.

Each AWS region operates using multiple CPU types with different capabilities, and the configured resource flavor (memory size) can have an impact on the selected CPU type. For single-threaded Python code, Lambda performance approximately doubles as assigned memory size is doubled until reaching the peak performance at around 1792 MB (one physical core is allocated). Our measurements indicate that execution time has no correlation with the time of the measurement but it is highly affected by the assigned CPU type and the selected Lambda resource flavor. We observed that AWS, time independently, assigns Lambda instances to different types of CPUs available in the chosen region in an undisclosed manner. At small flavors we measured significant differences among CPU types, but as higher flavors were selected, the differences diminished. Many different methods exist for invoking Lambda functions but most of them are inadequate for handling latency sensitive applications as they impose high delays with high variation, even for small transmitted data size. The quickest Lambda invocations are the SDK's and the API Gateway's synchronous calls, however, they have adverse effects on the execution time (thus the price) of the invoker function. Therefore, using asynchronous SDK calls can be a better fit for latency constrained applications. Long-term SDK asynchronous invocation tests showed no dependency on either the time of the call, the CPU type or the flavor of the instance. On average, we measured 103 ms when transmitting payloads with 130-kB size and 79 ms for 1 kB. Considering the asynchronous nature of the call, we measured surprisingly high blocking delay (the time while the invoker function gets blocked during an invocation) in the invoker function (52 and 44 ms, respectively). As Lambda is designed to serve stateless functions, whenever states should be stored we have to use an external service. In our previous work, we concluded that Amazon ElastiCache for Redis outperforms every other AWS offering for serving such purposes. It can handle both read and write operations under 1 ms for data smaller than 1 kB. Redis performance is among the best throughput-wise as well, and it handles increasing concurrent access notably well.

B. Performance of AWS Greengrass

Although Greengrass and cloud Lambda functions share many features, they differ in multiple aspects, as discussed in Section II-B, that significantly affect performance. In case of latency sensitive applications, the most important performance features to measure are flavor dependent computation proficiency and invocation latency. In order to investigate these aspects with AWS Greengrass, we repeated the respective benchmarks discussed in [28]. We used two different edge nodes to execute the tests: a local server with four Intel Xeon E5-2650 v3 CPU cores, 6 GiB of memory running Ubuntu 18.04 and an Amazon EC2 t2.micro instance with 1 vCPU and 1 GiB memory running the same OS in the *eu-west-1* (Ireland) AWS region. Each measurement was repeated 100 times to obtain average values and standard deviation.

1) Execution Time: As opposed to AWS Lambda behavior, Greengrass does not apply a memory size dependent access to compute resources. The service limits instance access to resources by using `cgroups`, however, it always provides

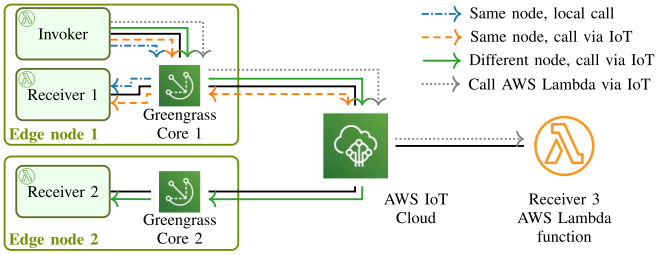


Fig. 3. Invocation measurement setup for AWS Greengrass.

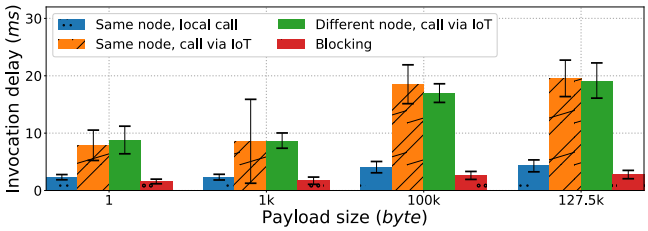


Fig. 4. Measured invocation and blocking delays.

access to unlimited processor time for each running function instance as their `cpu.share` parameter is set to 1024. Our measurements proved to be perfectly in line with this, as running multiple instances of the same function cause no significant increase in execution time until every core has been occupied by a function instance. However, when we start up twice as many function instances as the number of CPU cores, the execution times doubles. The behavior shows that the management jobs executed by the Greengrass core do not require significant CPU resources when no messaging is performed among the function instances.

2) Invocation Delay: As Fig. 3 depicts, there can be four different call paths among functions when AWS Greengrass is used depending on the location of the *Invoker* and *Receiver* functions.

1) When both functions are on the same edge node and local invocation is used.
2) The function locations are the same, but the call goes through the AWS IoT Cloud topic.
3) The two functions are on different edge nodes.
4) The Receiver function is an AWS Lambda function residing in the central cloud.

We benchmarked these scenarios on both of our edge nodes. In accordance with our previous measurements in [28] using the same methodology as here, invoking a function in the central cloud from the edge is the slowest, taking 125–231 ms to complete. In terms of latency, this invocation type is one of the slowest of available AWS Lambda calls and is 20–30-ms slower than asynchronous SDK calls between Lambda functions. Results for the rest of the cases measured on the t2.micro instance are shown in Fig. 4 together with the blocking delay caused by the invocations. (We opted to exclude the depiction of edge to central cloud calls from the figure in order to provide better visibility on invocation delay characteristics between edge functions). We can conclude that Greengrass local calls (calls between functions managed by the same Greengrass Core) are extremely fast compared to other Lambda function invocation options. As the local AWS Greengrass Core can handle the invocations, they last only 2.3–4.3 ms depending on payload size. Because of the Greengrass service's architecture, any other invocation has to interact with the AWS IoT Core, thus calls have to traverse the IoT Cloud topic. These invocations experience 7.8–19.5-ms delay when the Receiver function is found on a Greengrass node. When using our on-premise edge node, the increase in latency corresponded to the latency between our premises and the AWS region we used for the test. Blocking delay, the time while the Invoker function gets blocked during an invocation, is always small, ranging from 1.5–2.8 ms which is a fraction of those measured for the asynchronous cloud calls (50–70 ms).

Comparing the above results with those given by [28], we can conclude, that using AWS IoT Greengrass solutions result in relatively low latency only when the cloud functions are not involved. If an application requires low latency as well as edge and cloud functions, it is better to use SDK calls between them instead of relying on AWS IoT.

C. Service Model

The service model describes the user-defined service request including the software components and their interactions. Let S be the *service* structure description which is basically a directed multigraph. *Function* nodes F represent the simple, stateless and single-threaded basic building blocks, which use *invocations* I to call other functions and *read* R, *write* W arcs to perform I/O operations on *data store* nodes D. A dedicated platform node $\wp$, in the role of the API Gateway or the user, represents the main entry point of the service and designates the ingress service invocations. Recursive loops are modelled in their expanded form in which each iteration step is given with explicit invocations. This concludes the invocation subdigraph $S[F_\wp]$, unlike Control Flow Graphs, to be loopless, that is, a directed acyclic graph (DAG). Moreover, functions are considered to have only a single entry point which has a strict syntax typically predefined by the execution framework. The single-predecessor function characteristic further restricts $S[F_\wp]$ to be a directed rooted tree with $\wp$ as the root node.

Functions are characterized by the execution time τ measured on one vCPU core, while arcs have the average invocation rate ω_r attribute along with the explicit blocking delay δ_Φ introduced in the invoker function. Data stores can be described by their workload capacity in general. In addition to the graph-based description, the service model also keeps track of user-defined node-disjoint path(s) Π with associated latency limit l_π as the basic constraints for the layout optimization.

D. Platform Model

Our *platform* model captures the performance characteristics and cost models of function execution, invocation and data store access methods, respectively. For the runtime environment, we only consider single-threaded serverless functions. However, our models can be extended to use containers [7] or to support multithreaded functions by using explicit function execution profiles. Runtime flavors Φ are specified by their offered vCPU fraction n_c. To extend our previous model with

edge computation capabilities, we introduce edge nodes as standalone flavors. Thus, an assigned flavor implicitly carries basic placement information, that is, designating the specific edge node or the central cloud as required by the deployment engine. While Greengrass Lambdas always have access to one vCPU core on edge nodes, i.e., $n_c(\Phi_E) \triangleq 1$, the core fraction of cloud Lambdas can be derived from their assigned memory as $n_c(\Phi_\lambda) = \min\{([\ell_m(\Phi_\lambda)]/[\ell_m(\Phi^*_\lambda)]), 1\}$. The first Lambda flavor granting one core is $\Phi^*_\lambda = 1792$ MB as stated in Section V-A.

Regarding invocation types, we assume two different options relevant to latency sensitive applications. More specifically, *async SDK* invocation, depicted in Section V-B, and *local* invocation are considered. Local invocation is used when one function directly invokes another function in the same group and its blocking overhead is negligible in terms of latency.

E. Cost and Latency Models

Making use of our service and platform models, we can describe the end-to-end latency and the operation costs of the application. While serverless platforms support parallel function execution via autoscaling, internal parallelization (within a Lambda function) could also be realized by applying multithreading and internal asynchronous calls scheduled by the runtime. However, we consider single-threaded functions and runtime environments with a single core. Therefore, in order to calculate overall latency (and costs), we can model all functions as single-threaded components.

These functions can be composed together, where they can call each other directly in a synchronous manner, and executed in a single Lambda function. This way, the grouping of functions can reduce the overall latency by eliminating SDK invocation overheads in return for additional costs. In the same time, function grouping introduces serialized execution of the encompassed functions resulting in increased group execution time. The number of consecutive executions of a function is determined by its caller component's behavior. This can be modelled with a multiplier, i.e., the *serialization ratio*, which is the ratio of the caller and called component's invocation rates. This quotient is greater than 1 when the caller iteratively performs invocations, around 1 if it realizes one-to-one mapping and less than 1 if outgoing calls are filtered by conditional statements. First, let T_s define the overall service runtime. Then, let t_p denote the execution time of function group $p \in P_F$ on selected flavor $\phi_p \in \Phi$. In (1) we define t_p as the sum of the actual function execution times including flavor-related data and egress invocation overheads $A^+(p)$, and multiplied by the serialization ratio. Invocation i_f and i_p mark the ingress invocations of function f and belonging group p

$$t_p = \sum_{f \in p} \frac{\omega_r(i_f)}{\omega_r(i_p)} \left(\frac{\tau(f)}{n_c(\phi_p)} + \sum_{a \in A^+(p)} \delta_{\phi_p}(a) \right). \tag{1}$$

In accordance with AWS billing patterns, we use rounded up group execution time for the Lambda cost calculation. In addition, we define the summed number of received requests as $r_p \triangleq \omega_r(i_p) T_s$ for group $p \in P_F$. The flavor-dependent group cost function c_p is formulated in (2), where C_r, and C_p are the billing constants specified by the cloud provider for the total number of requests and rounded group execution time

$$c_p(p, \phi_p) = r_p\big(C_r + C_p \lceil t_p \rceil_{100\,\text{ms}}\big). \tag{2}$$

Although service cost calculation relies on the entire group execution time, the observed latency differs from t_p values. The end-to-end latency measured at a function can include different number of consecutive executions of the preceding functions based on their position in their serialization sequence. Thus, the number of distinct execution variations from which the measured latency value is computed is determined by the serialization ratios of the preceding functions. As these execution variations contribute evenly to the average latency value we define a modified formula $\bar{l}_p$ for the group latency calculation in

$$\bar{l}_p(p, \phi_p) = \sum_{f \in p} \frac{\frac{\omega_r(i_f)}{\omega_r(i_p)} - 1}{2} \left(\frac{\tau(f)}{n_c(\phi_p)} + \sum_{a \in A^+(p)} \delta_{\Phi_p}(a) \right). \tag{3}$$

With the same approach, we can formalize the cost function for data stores as well. As there are no outgoing data transfers, the data store cost only depends on the service runtime T_s and instance type C_i. Therefore, it can be expressed as a single layout-independent cost value $C_i T_s$.

F. Optimization Problem

The LPO's output is the service layout which defines the function partitioning P_F (equivalently called as clustering in the literature) along with the flavor assignment φ. Thus, the optimization task is to find the cost-efficient layout over the cloud/edge environment considering latency requirements.

Our problem, which falls under the topic of graph partitioning, is a complex task in general. For simplicity, we make the following assumptions without losing the original target.

1) We consider only one central cloud Lambda flavor and one edge flavor.
2) Since data stores $S[D]$ do not form a connected subdigraph and their cost is layout-agnostic depending on T_s solely, data store flavor assignment can be realized as a separated upper-bounded aggregation. Thus, we focus on the $S[F]$ partition problem in the following (as $\wp$ must not be part of any group).
3) We do not assume internal thread-based parallelization as functions represent simple software building groups. This means, P_F has to be technically a valid graph partitioning of $S[F]$ where partition groups are considered to be directed linear chains with no limit either on their number or size.

Summarizing the above, we define our objective function as to find the chain partitioning (P_F, φ) with the minimal cost

$$\min \sum_{p \in P_F} c_p(p, \varphi(p)) \tag{4}$$

such that the following constraints must be met.

1) Latency limit l_π of a given path π is not to be violated.
2) Function group $p \in P_F$ must contain exactly one chain.

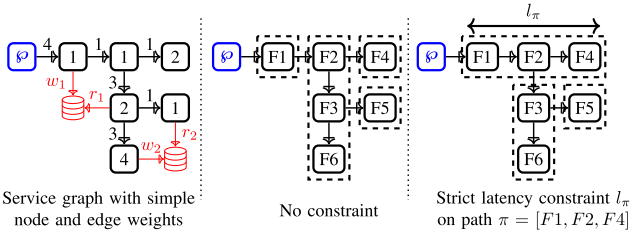


Fig. 5. Partitioning examples of a tree service (left) with no constraint (middle) and with strict latency constraint l_π (right).

Fig. 5 shows two illustrative partitioning of an example service. The latency constraint l_π leads to a substantially different grouping regarding the number and composition of the groups.

VI. Proposed System

We have applied our general design principles presented in Section IV to AWS Lambda and Greengrass and the complete system is shown in Fig. 6. Our prototype was implemented in Python3 making use of the AWS SDK and AWS IoT Greengrass SDK. In this section, the main components, algorithms and workflows are described in detail and we present how the exposed APIs of AWS are exploited by our system.

A. Layout and Placement Optimizer

The main task of the LPO is to solve the optimization problem defined in (4). Graph partitioning or clustering have been well-researched for decades. While partitioning is known to be $\mathcal{NP}$-complete for arbitrary directed graphs as well as weighted trees [14], several polynomial-time algorithms exist for sequential graph partitioning (SGP) which restricts the partition groups to contain only consecutive nodes [16], [25]. The available techniques for solving SGP assume either an upper bound for the group sizes or consider only fixed number of groups. However, our problem differs from the traditional variants of SGP in several aspects. Since we aim to split up trees explicitly into chains and the partition groups are bounded by the latency limits of service-wide critical paths in contrast to the locally verifiable group size or count limits, the aforementioned methods cannot be applied directly to our problem. By extending our prior algorithm designed for public clouds [7], we propose a heuristic approach for cost-efficient and latency-constrained partitioning of trees into chains on cloud and edge resources, called Chain-based Tree-Partitioning (CTP).

1) Chain Partitioning: First, we define the relaxed Chain-Partitioning (CP) algorithm utilized by CTP as a subproblem to solve tree partitioning. CP specifies chain partitioning as a variant of noncrossing sequence partitioning and leverages its related divide-and-conquer approach [23]. Suppose an n-length chain of functions f with their measured performance characteristics, the number of counted subcase latency bounds B and an optional path $[\pi_s, \pi_e]$ limited by l_π as the algorithm's input. Here, the cost-efficient partitioning along with the assigned flavors, overall cost and latency values can be derived by iteratively evaluating the recurrence relations in (5).

In the recursive formulas, the subcase of the first i nodes of the chain grouped into j groups is divided into two subparts: the previously calculated subcase of the first $k-1$ nodes into $j-1$ groups and the remaining last $k \xrightarrow{*} i$ nodes as a single group. Since the assigned flavor ϕ of the last group and its invoker group's flavor ν inherently predetermine the group execution times and the invocation delay between the two subparts, the selection of a minimal cost subcase cannot be guaranteed to be globally optimal regarding the overall latency constraint. Therefore, we use B precalculated latency bounds for each subcase and cache the related cost-optimal partitioning which enables tracking of more expensive subcase variants with better latencies that are optionally chosen during a subsequent iteration. These bounds are calculated evenly between the overall latency limit l_π and the smallest execution time of single function groups in descending order, keeping cheaper variants with lower bound indices b. The cost-optimal partitioning of a subcase is designated by the specific k^* value where the summed cost of the two subparts is minimal. In case of multiple minima, the subcase with the lowest index k, that is, the lowest group count, is chosen. During each iteration, all flavor combinations ν, ϕ are examined and only those prior subcases with feasible bounds $b^k_{\nu,\phi}$ are taken into account which meet the given bound b including the execution time of the last group and its invocation delay. To track the relevant subcases' values, dedicated matrices $\mathbf{C}$ and $\mathbf{L}$ are introduced for storing the summed cost and latency calculated with c_p and $\bar{l}_p$ from (2) and (3). Latency calculation formulated in $l(k, i, \nu, \phi)$ is performed only for the constrained path $[\pi_s, \pi_e]$ using the flavor-dependent invocation delays formed in matrix $\mathbf{D}$. To be able to reconstruct the partition groups, matrices $\mathbf{K}$, $\mathbf{B}$ and $\mathbf{F}$ are used for caching the barrier node k^*, by which the optimal subcase is divided, the opted latency bound b^* of the prior subcase and the last group's flavor ϕ^* opted for k^*, respectively

$$
\begin{aligned}
\mathbf{C}[i,j,b] &= \mathbf{C}[k^*-1, j-1, b^*] + c_p\Big(f\Big[k^* \xrightarrow{*} i\Big], \phi^*\Big)\\
\mathbf{L}[i,j,b] &= \mathbf{L}[k^*-1, j-1, b^*] + l\big(k^*, i, \nu^*, \phi^*\big)\\
\mathbf{K}[i,j,b] &= k^* \quad \mathbf{B}[i,j,b] = b^* \quad \mathbf{F}[i,j,b] = \phi^*\\
b^* &= \min_{\nu,\phi\in\Phi} b^{k^*}_{\nu\phi} \qquad \nu^*, \phi^* = \operatorname*{argmin}_{\nu,\phi\in\Phi} b^{k^*}_{\nu\phi}\\
k^* &= \operatorname*{argmin}_{j\le k\le i} \min_{\nu,\phi\in\Phi} \Big\{c(k,\nu,\phi) + c_p\Big(f\Big[k \xrightarrow{*} i\Big], \phi\Big)\Big\}\\
c(k,\nu,\phi) &= \begin{cases} \mathbf{C}\Big[k-1, j-1, \min b^k_{\nu\phi}\Big]\Big), & \text{if } b^k_{\nu\phi} \neq \emptyset\\ \infty, & \text{otherwise}\end{cases}\\
b^k_{\nu\phi} &= \big\{\beta \mid \mathbf{L}[k-1, j-1, \beta] + l(k,i,\nu,\phi) \le b\\
&\qquad \wedge \mathbf{F}[k-1, j-1, \beta] = \nu\big\}\\
l(k,i,\nu,\phi) &= \begin{cases} 0, & \text{if } [\pi_s,\pi_e]\cap[k,i] = \emptyset\\ \bar{l}_p\big(\widehat{f}_{ki}, \phi\big), & \text{if } \pi_s \ge k\\ \mathbf{D}[\nu,\phi] + \bar{l}_p\big(\widehat{f}_{ki}, \phi\big), & \text{otherwise}\end{cases}\\
\widehat{f}_{ki} &= f\Big[\max\{\pi_s, k\} \xrightarrow{*} \min\{\pi_e, i\}\Big]\\
i &= [2\ldots n]; \quad j = [2\ldots i]; \quad b = [1\ldots B].
\end{aligned}
\tag{5}
$$

The dynamic programming technique provides an efficient way to solve the recursive formulas in (5). Algorithm 1

Algorithm 1 Chain-Partitioning

procedure CHAINPARTITION($\ldots, \pi_s = 0, \pi_e = n, B = n, l_\pi = \infty$)
  Define $DP \leftarrow n \times n \times r$ matrix with 5-tuples as $\langle C, L, K, B, F\rangle$
  Apply memoization to functions $\bar{l}_p, c_p$ with cache size $n * |\Phi|$
  if $l_\pi = \infty$ **then** ▷ Calculate latency bounds
    $bounds \leftarrow [\infty]$
  else ▷ Decreasing bounds of evenly spaced r ranges
    $bounds \leftarrow$ LINSPACEBOUNDS($l_\pi, \min_{f\in F,\phi\in\Phi} \bar{l}_p(f,\phi), B$)
  for $i \leftarrow 1$ **to** n; $b \leftarrow 1$ **to** $|\Phi|$ **do** ▷ Precalculate trivial subcases
    $DP[i,1,b] \leftarrow \langle c_p(f_{1i}, \Phi_b), \bar{l}_p(f_{1i}, \Phi_b), 0, 0, \Phi_b\rangle$
    REVERSESORTBYLATENCY($DP[i,1]$)
  for $i \leftarrow 2$ **to** n; $j \leftarrow 2$ **to** i; $k \leftarrow j$ **to** i; $\nu, \phi \in \Phi$ **do**
    for $b \leftarrow 1$ **to** B **do**
      if $bounds[b] - D[\nu,\phi] - \bar{l}_p(f_{ki},\phi) \le 0$ **then** ▷ No residue
        break
      $b_{idx} \leftarrow$ GETFEASIBLEBOUNDINDEX($DP[k-1, j-1], b, \nu$)
      if $b_{idx} = NULL$ **then** ▷ No sufficient bound with flavor ν
        break
      $cost \leftarrow DP[k-1, j-1, b_{idx}].C + c_p(f_{ki}, \phi)\}$
      if $cost < DP[i,j,b].C$ **then** ▷ Subcase with lower cost
        $lat \leftarrow DP[k-1, j-1, b_{idx}].L + \bar{l}_p(f_{ki}, \nu, \phi)$
        $DP[i,j,b] \leftarrow \langle cost, lat, k, b_{idx}, \phi\rangle$
  $k^*,\ barr,\ flav,\ k_b,\ b_b \leftarrow \text{argmin}_k\, DP[n,k,1].C,\ [\,],\ [\,],\ n,\ 1$
  for $j_b \leftarrow k^*$ **to** 1 **do** ▷ Reconstruct groups
    $barr[j_b],\ flav[j_b] \leftarrow DP[k_b, j_b, b_b].K,\ DP[k_b, j_b, b_b].F$
    $k_b,\ b_b \leftarrow DP[k_b, j_b, b_b].K - 1,\ DP[k_b, j_b, b_b].B$
  return GETBLOCKS($barr, flav$), $DP[n, k^*, 1].C$

presents the pseudocode of our implementation which uses one matrix *DP* with 5-tuples. At the first step, the bound values are calculated based on B. Before calculating the subcases, *DP* is initialized with default values as grouping the first i nodes into a single group is trivial. During each iteration, the latency residue is calculated for the prior subcases and used for finding the feasible bound with the lowest index b and the subcase with the lowest summed cost is stored. As the last step, the partition groups are reconstructed. Groups are determined by the list of barrier nodes and the assigned flavors, which can be obtained recursively from the stored values. The barrier and bound values and their indices in the matrix designate the last group's first node and its assigned flavor, and inherently mark the next barrier's position. The cost-minimal partitioning values $\mathbf{C}[u \overset{*}{\rightarrow} v]$ and $\mathbf{L}[u \overset{*}{\rightarrow} v]$ can be obtained using indices $[n, k^*, 1]$ where $k^* = \text{argmin}_k\, \mathbf{C}[n,k,1]$. Additionally, function c_p and l_p are enhanced with memoization, where the calculated result for given function parameters is stored in a least recently used (LRU) cache to reduce the repeating computation steps.

Theorem 1: CP has time complexity of $\Theta(([n(n+1)]/2)B|\Phi|^2)$.

Proof: With memoization, each iteration step can be computed in $\mathcal{O}(1)$ time. Since i nodes can be grouped into maximum i groups, CP uses the left triangular part of the $n \times n$ matrices per bounds B, which results in $([n(n+1)]/2)B|\Phi|^2$ iterations. ∎

2) Tree Partitioning: Following an analogous formalization, CTP recursively calculates the partitioning of a subtree in S by leveraging the CP algorithm and previously calculated subtree groupings to enforce the node-disjoint critical paths Π. To accomplish this efficiently, CTP precalculates all the reachable leaves from each node by labeling the nodes using postorder DFS and the label definition in the following equation, where $N^+(v)$ is the out-neighborhood of node $v \in V(S[F])$

$$\mathcal{L}(v) = \begin{cases} \bigcup_{u\in N^+(v)} \mathcal{L}(u), & \text{if } \deg^+(v) > 0 \\ \{v\}, & \text{otherwise.} \end{cases} \tag{6}$$

In order to ensure CTP to inspect every candidate partitioning of a subtree, we define the *Subchain-Pruning* action which leverages *Node-Labeling* to track the chain from subtree root r to target leaf l, while it also fetches the chain-adjacent subtree roots $N_c^+(r \overset{*}{\rightarrow} l)$. It operates roughly as follows: Starting from the subtree root, *Subchain-Pruning* iteratively checks the labels of descendant nodes. The child node that has the target label is a member of the actual chain and marked as the next step, while the remaining successors belong to the chain neighbors.

We can get valid chain partitioning of an arbitrary subtree if we perform *Subchain-Pruning*, then apply *Chain-Partitioning* on the resulting root-leaf chain and take the partitioning of chain-adjacent subtrees. Consequently, we shall cover the cost-optimal subcase if we perform *Subtree-Pruning* on each leaf-ending chain designated by the subtree root's labels. To ensure the latency constraints, a separated chain traversal step is realized, similarly to *Subchain-Pruning*. Each critical path originating on the chain is checked during the traversal, while the related latency fragments of impacted paths are cached in $\mathbf{L}$. The latency limit that fits entirely on the chain is enforced by CP itself. This follows that CTP accepts constraints assigned for distinct node-leaf subchains solely, otherwise only the critical path originating the closest to $\wp$ can be guaranteed. Finally, we formulate our recursive CTP algorithm in

$$\mathbf{T}[n] = \min_{l\in\mathcal{L}(n)} \left\{ \sum_{m\in N_c^+\left(n \overset{*}{\rightarrow} l\right)} \mathbf{T}[m] + t(n,l) \right\}$$

$$t(n,l) = \begin{cases} \mathbf{C}\left[n \overset{*}{\rightarrow} l\right], & \text{if } \forall \pi : \mathbf{L}\left[\pi_s \overset{*}{\rightarrow} \pi_e\right] \le l_\pi \\ & \text{s.t. } \pi_s \overset{*}{\rightarrow} \pi_k \subset n \overset{*}{\rightarrow} l \\ & \quad \pi_{k+1} \in N_c^+\left(n \overset{*}{\rightarrow} l\right) \\ \infty, & \text{otherwise} \end{cases}$$

$$\mathbf{L}\left[\pi_s \overset{*}{\rightarrow} \pi_e\right] = \mathbf{L}\left[\pi_s \overset{*}{\rightarrow} \pi_k\right] + D\left[\phi_{\pi_k}, \phi_{\pi_{k+1}}\right] + \mathbf{L}\left[\pi_{k+1} \overset{*}{\rightarrow} \pi_e\right]. \tag{7}$$

To ensure the proper processing order of $S[F]$, CTP utilizes reversed BFS tree traversal. Evidently, the optimal partitioning cost of $S[F]$ is cached in $\mathbf{T}[r]$, where r is the root node of $S[F]$. Partition groups are tracked by the array $\mathbf{P}$ which caches the set of reconstructed groups as intervals for each subtree. The pseudocode in Algorithm 2 summarizes the CTP algorithm.

Theorem 2: If c is the longest chain size and Δ is the highest degree in $S[F]$ CTP has time complexity $\mathcal{O}(n(B|\Phi|^2 + \Delta)c^3)$.

Proof: First, *Node-Labeling* is a slightly modified DFS traversal where in each backtracking step the labels are collected from the node's successors. Since DFS requires $\mathcal{O}(|V| + |E|)$ steps and in trees $n \equiv |V| = |E| + 1$, node labeling has $\mathcal{O}(n)$ complexity. To track a chain in a subtree,

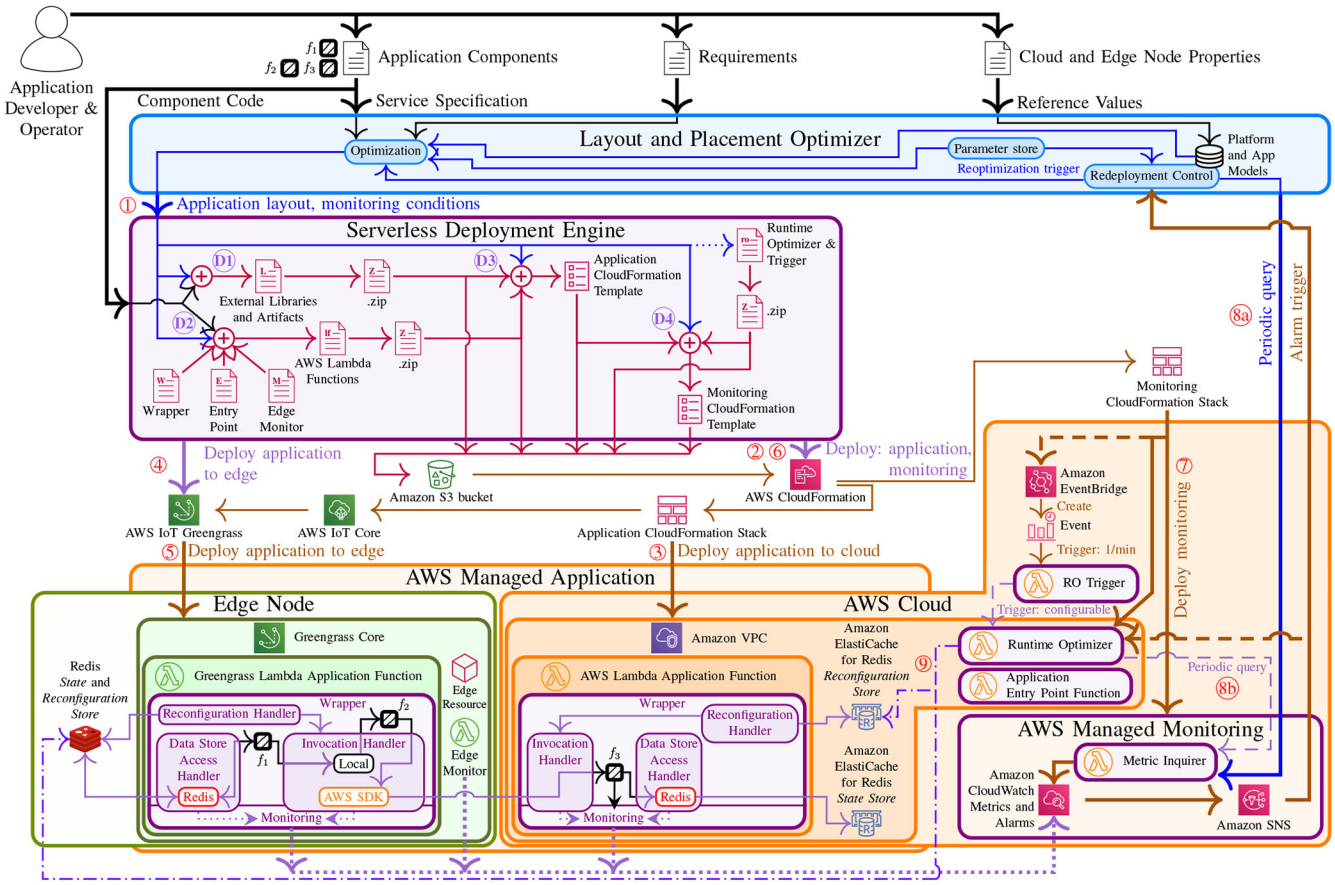


Fig. 6. Proposed system: overall architecture and details of our deployment, monitoring and reoptimization cycles.

Algorithm 2 Chain-Based Tree-Partitioning

procedure TREEPARTITION(*tree*, *root*, $\Pi = \{\}$, *B*)
  Define T, P as $|F|$-sized lists, L as $|F| \times |F|$ matrix with value ∞
  DONODELABELINGS(*tree*, *root*)
  for all *node* $\in$ REVERSEDBFS(*tree*, *root*) **do**
    for all *leaf* $\in \mathcal{L}$(*node*) **do**
      chain, *nghbrs* $\leftarrow$ SUBCHAINPRUNING(*tree*, *node*, *leaf*)
      ℓ_π, π_s, π_e $\leftarrow$ GETCRITICALPATH(*tree*, *chain*, Π)
      params $\leftarrow$ GETCHAINPARAMETERS(*chain*)
      part, *cost* $\leftarrow$ CHAINPARTITION(*params*, π_s, π_e, *B*, l_π)
      valid, *sub_lats* $\leftarrow$ CHECKCRITPATHS(*chain*, *part*, Π, L)
      sum_cost $\leftarrow$ *cost* $+ \sum_{m \in nghbrs}$ T[*m*]
      if *valid* **is true and** *sum_cost* $\leq$ T[*node*] **then**
        T[*node*] $\leftarrow$ *sum_cost*
        P[*node*] $\leftarrow$ GETPART(*tree*, *part*) $\cup \bigcup_{m \in nghbrs}$ P[*m*]
        L $\leftarrow$ L $\cup$ *sub_lats*
  return P[*root*], T[*root*]

Subchain-Pruning checks each node in the chain with all their neighbors, which can be upper bounded with the longest chain c and maximum degree Δ of $S[F]$. It leads to $\mathcal{O}(c\Delta)$ time complexity.

CTP visits each node and tests the partitioning subcases based on the node labels and *Subchain-Pruning*. It requires $\sum_v \mathcal{L}(v)$ checks in total which is exactly the sum of the sizes of all chains in $S[F]$. This value can be upper bounded by the longest chain size c multiplied by the maximum leaf count which is $n-1$ in trees. Each iteration needs $c\Delta$ steps from both the *Subchain-Pruning* and the latency segment testing, and $([c(c+1)]/2)B|\Phi|^2 < c^2B|\Phi|^2$ steps from *Chain-Partitioning* as we can overestimate the partitioned chain size with c. This follows that $c(n-1)[c^2B|\Phi^2 + 2c\Delta] \ll n(B|\Phi|^2 + \Delta)c^3$. ■

B. Serverless Deployment Engine

One level below the LPO, the SDE is responsible for translating application layout and monitoring conditions arriving in step ① (see Fig. 6) from the LPO to calls that AWS can process for setting up resources. On a high level, the SDE communicates directly with AWS accessing its CloudFormation (CF) and Greengrass services. The former is configured via its own templating language. CF processes incoming template requests describing what resources to set up, in which order and what connections will these resources have with each other, and creates individual CF stacks or stack sets from them. In our implementation, the SDE synthesizes templates specifying single CF stacks as a simplification. In step ②, the SDE passes a template to CF that defines all the components for the *AWS Managed Application* in the cloud as well as it configures Greengrass related resources to be deployed to edge nodes. When cloud resources have been set up by CF in ③, the SDE calls AWS Greengrass directly for deploying resources to edge nodes in step ④, since CF is incapable of deploying code to edge resources. After completing the whole application setup with edge deployment in step ⑤, the SDE configures elements required for the *AWS Managed Monitoring* of the deployed

application and the RO component. These are also exchanged with CF in step ⑥ and are set up in step ⑦. In steps ② and ⑥, application and monitoring code and other artifacts are shared between the SDE and CF in compressed format using AWS's own object storage service, *Amazon S3*.

In accordance with these, the SDE goes through four phases internally for creating the *Application* and *Monitoring CloudFormation templates* and artifacts for applications written in Python. (Of course, the concept can be applied for other programming languages supported by AWS Lambda as well). In phase (D1), external libraries and developer defined function resources required by the application components are collected. These are compressed depending on component placement and then uploaded to Amazon S3. In phase (D2), the actual code of application components gets processed. The code for every component group defined by the LPO is collected and purpose-built *Wrapper* code is added to them as well. Two special functions are added to the application. The *Entry point* function is able to divert incoming requests to the application's own entry point be it on an edge node or in the cloud. The *Edge monitor* function performs CPU and memory load measurements on edge nodes. The resulting *AWS Lambda functions* are compressed and uploaded to S3. In phases (D3) and (D4), the SDE formulates the application and monitoring CF templates, respectively. During application template creation, incoming layout and flavor specifications are used. These are complemented with code and artifact locations in S3 as well as additional AWS resources that are needed in order to set up the application properly. Such resources include but are not limited to AWS Lambda layers, versions, aliases, Amazon VPC, subnets, Internet Gateways, NAT Gateways, ElastiCache clusters, AWS IAM security policies and roles as well as Greengrass groups, cores, resources and subscriptions. The SDE's Python3 implementation contains around 2500 LoC.

The AWS Managed Application is ready to run as soon as CF finishes with step ③ if the application does not use edge resources or at step ⑤ otherwise. As depicted by the bottom side of Fig. 6, all interactions among application components with each other or with data stores, traverse our Wrapper. This lightweight runtime extension is capable of hiding (edge or cloud) placement differences, function invocation and data store access specifics from application components. It serves as the unique standardized entry point to functions that have been grouped together by the LPO, and it even relays function specific environment variables. Configuration of the Wrapper is also performed via environment variable assignment in the template at phase (D3) within the SDE. Here, the specific Lambda, IoT topic and Redis endpoints are supplied to the Wrapper. During normal application operation, our Wrapper implementation, comprising of 630 lines of Python code, adds negligible latency to application E2E latency as configuration parameters are cached (in Python dictionaries and objects) and the Wrapper's internal handler components are extremely lightweight. In case of cold start up, when configuration parameters need to be processed, Wrapper overhead is slightly greater but still remains under 2 ms.

C. Automated Monitoring

Since every communication attempt between application resources goes through the Wrapper, it proves to be ideal for handling monitoring related functionality as well. As these invocations and data store accesses traverse the Wrapper, it measures then logs call latency and rate, blocking delay as well as function execution time. An interface for logging custom application level metrics of application components is provided as well. Measured values are reported to the AWS Managed Monitoring component. This entity has three tasks: aggregating metrics, sending out alerts and providing a queryable interface. The first two tasks are handled by Amazon CloudWatch (CW). When logging monitoring data to CW, the Wrapper experiences significant, available CPU dependent delay. In case of the smallest Lambda flavor (128 MB), we experienced 125 ms on average with high variance using the highest available batching (20 metrics). However, thanks to implementation details, this does not contribute to application delay at all (metrics logging is running virtually in parallel with the application). It does, however, contribute to the price of maintaining the application. Data coming from the Wrapper goes to CW Metrics and limit violations are handled by CW Alarms. This latter AWS service is configured within the monitoring template in phase (D4) of the SDE for conditions coming directly from the Developer or the LPO. Alerts are sent out from the Monitoring component to the LPO and RO in steps (8a) and (8b), respectively, using the integration between CW and Amazon Simple Notification Service (SNS). For measurement data that does not trigger alarms, the component offers access via a *Metric Inquirer* function that is also deployed at steps ⑥–⑦.

D. Dynamic Reoptimization

The above discussed features of the Monitoring component serve as a basis for the closed loop reoptimization of the application. After deployment, the application starts to log usage metrics automatically that either trigger an alarm, or one of the optimization components discovers a nonalerting change in the application's behavior and initiates a change (see steps (8a) and (8b)). Depending on which component reacts, we define two control loop behaviors that differ in their reaction timescale as well as in their possibilities to make changes in the application.

1) Steady-State Control: The steady-state control loop strives to follow usage trends, daily profiles, or changes in the application users' behavior. As a default means to accomplish this, the LPO periodically queries the Managed Monitoring component via the Metric Inquirer facility of the latter (see step (8a)) and updates its own Platform and Application Models. Periodicity of the query is dependent on LPO configuration and certain use cases can require more frequent updates than others. For convenience, the Monitoring component is able to trigger the LPO directly as well, supplying notifications about changes in reported metrics out of regular query periods. In the current implementation, the SDE sets up such triggers as application E2E latency alarms in the

Monitoring component in step (D4) of the deployment, when a latency constraint is provided in the service specification. Both types of changes can induce service reoptimization in the LPO. In order to decide whether the deployed layout is worth replacing with a new one, a dedicated redeployment metric is applied. The LPO compares the user-given threshold value with the weighted sum of the following values to make the deployment decision: 1) costs of the relative change in the layout; 2) relative profit gain which is the difference of the deployed layout cost calculated with the updated service parameters and the new layout cost; 3) summed latency gain; 4) relative latency margin on critical paths; and 5) the number of avoided latency constraint violations. When the LPO deems a new layout better than the currently deployed based on this metric, it initiates a full redeployment incorporating steps ① through ⑥.

2) Dynamic Runtime Reconfiguration: In this case, the RO is the component making changes in the application. This has limited possibilities as it can switch between predeployed layouts by offloading functions from edge nodes or reverting these changes. The RO interacts with the Monitoring component in step (8b). With push-based alerting, the RO cannot get triggered sooner than 10 s after an alarm condition presents itself, because of CW Alarms limitations. Faster reaction time is achieved by placing periodic queries to CW Metrics, realizing poll-based execution. In order to perform such queries by the RO, we use a combination of an Amazon EventBridge event and an AWS Lambda function. Both of these are deployed at step ⑦, and EventBridge sets up a trigger event that gets fired every minute (which is the shortest time period for the service). The event trigger invokes our custom-made *RO Trigger* function that schedules the RO to run frequent periodic queries to the monitoring component. In either push or poll-based execution, the RO interacts directly with the Wrapper as shown by step ⑨ in Fig. 6. For on-demand functions in the cloud, the SDE configures a Redis instance at deployment, while for edge functions it uses the one available on the edge. The RO writes offloading information to these Redis instances and the Wrapper checks them before each function execution. As this data is small in size and Redis read operations have small latency, the average delay of this overhead is negligible (less than 1 ms) compared to the execution time of the application component. After reading a change request, the *Reconfiguration Handler* in the Wrapper changes subsequent invocations from edge local calls to cloud calls or vice versa.

VII. Evaluation

In this section, we evaluate the performance of our system investigating the use case presented in Section III, in varying operating regimes. First, the main operation phases of the overall system are characterized. Second, the performance of the steady-state control loop is analyzed, and finally, we evaluate the performance of the dynamic runtime reconfiguration loop. For describing our software deployment layouts, we introduce the following interval-based notation: $P = \{[i]_{C|E}, [j\text{–}k]_{C|E}, \dots\}$. Here, groups of single or multiple consecutive application functions denoted by their ordinals 1–n indices $i, j, k \in \mathbb{N}$ (see also in Fig. 1) are defined using square brackets and subscripts C or E identify the assigned cloud or edge flavors, respectively. E.g., in case of $P_{ECC} = \{[1\text{–}4]_E, [5]_C, [6]_C\}$, functions #1–#4 (Image Grab–Object Detection Stage 1 in Fig. 1) are placed within a group assigned to the edge, while functions #5 (Cut) and #6 (Object Detection Stage 2) are deployed in two distinct groups in the cloud. The experiments are conducted in Amazon's data centers located in the Ireland (eu-west-1), Frankfurt (eu-central-1) and Oregon (us-west-2) regions.

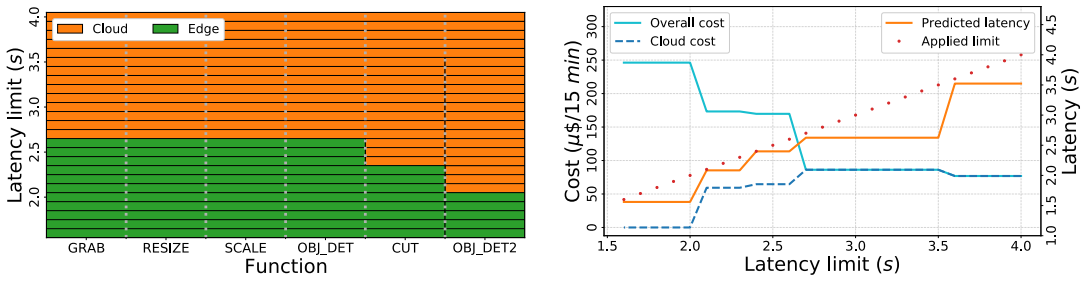
TABLE II
Mean Delay of Deployment Phases

Layout		LPO	SDE			
			LPO→CF translation	App code mgmt	CF deployment	Edge deployment
P_C	$= \{[1\text{–}6]_C\}$	3.5 ms	3.6 ms	20.1 s	52.9 s	N/A
P_{CC}	$= \{[1\text{–}5]_C, [6]_C\}$	4.3 ms	4.0 ms	35.2 s	64.7 s	N/A
P_{ECC}	$= \{[1\text{–}4]_E, [5]_C, [6]_C\}$	6.0 ms	5.4 ms	53.1 s	95.2 s	6.5 s
P_{EC}	$= \{[1\text{–}5]_E, [6]_C\}$	5.4 ms	5.1 ms	38.3 s	89.7 s	6.3 s
P_{EE}	$= \{[1\text{–}5]_E, [6]_E\}$	4.3 ms	5.4 ms	37.0 s	88.2 s	6.1 s
P_E	$= \{[1\text{–}6]_E\}$	N/A	5.6 ms	27.1 s	78.5 s	7.3 s
P_{6C}	$= \{[1]_C, [2]_C, [3]_C, [4]_C, [5]_C, [6]_C\}$	N/A	5.2 ms	81.3 s	132.0 s	N/A
P_{6E}	$= \{[1]_E, [2]_E, [3]_E, [4]_E, [5]_E, [6]_E\}$	N/A	6.9 ms	25.7 s	202.0 s	8.2 s

A. Overall System Performance

Table II illustrates the performance characteristics of the overall system when deploying select layouts. The first five options are generated by our system during normal operation. These show how the LPO changes the application's layout as it transitions from being completely cloud-based to completely deployed to the edge node, depending on different circumstances. (We discuss these cases and circumstances in more detail in Section VII-B.) The last three layouts in the table are corner cases created manually for comparison. The operation of the SDE is split into four distinct phases: translation from LPO to *AWS* CloudFormation (CF) format, application code management (application source code and external library collection, and upload), CF and edge deployment. For each layout, we executed 25 iterations where only application components were updated, state stores were not changed. Our system was executed on a *t3a.2xlarge* Amazon EC2 instance running in the same region chosen for deploying the application.

As shown in the table, LPO execution, and LPO→CF format translations have the lowest impact on deployment delay, both having subsecond values (under 7 ms) with our simple application. The LPO does not display high variance between different layouts as its execution time is dependent only on the number of used flavors which is now a fixed parameter. Using a fixed application, the translation's execution time depends on two factors: 1) number of groups in the layout and 2) the placement of these groups. As Table II shows, creating more groups naturally increases translation time, as setting up a function in AWS usually requires the specification



(a) (b)

Fig. 7. Simulation results of the LPO. (a) Calculated groupings of the use case application. (b) Predicted values of operational costs and E2E latency.

of multiple resources. Assigning functions to the edge node slows down the translation step for the same reason. Code management and CF deployment take significantly more time. The former requires 20–81 s to complete, as handling external libraries and ML models contribute heavily to phase latency. In case of the simplest P_C layout, a single artifact containing all the code, libraries and ML models is created and uploaded to AWS. In the worst case, P_{6C}, all these are packaged separately and sequentially for the six different functions, resulting in six comparatively big deployment packages. Phase delay is reduced when functions are mainly deployed to the edge, thanks to merging libraries and ML models in one single artifact on the edge. In case of P_{ECC}, however, the SDE still has to create deployment packages for functions #1–#4 and their shared libraries as well as separate ones for functions #5 and #6 that results in a comparatively high phase delay. CF deployment adds another 1–3.3 min to complete deployment time since connected Lambda functions are deployed sequentially instead of parallelly by CF and their update takes around 20 s each. As the difference between P_{6C} and P_{6E} (every function deployed separately in the cloud or on the edge, respectively) shows, edge related setup further adds to phase delay. The increase is due to the fact that for the edge, AWS needs to configure the complete Greengrass setup. Not only functions but the merged artifact containing libraries and ML models, as well as AWS IoT communication topics between the function groups. Edge deployment is comparatively quicker and less dependent on function grouping as external packages, shared among application functions, are deployed together in a common edge resource by AWS Greengrass. One or two function groups are deployed in 6.1–7.3 s, while assigning each function to a different group increases phase latency only with an additional 0.9 s. Overall, our measured complete deployment delay is 1.2–4 min depending on application layout. As the LPO's measurement update period is 15 min in our tests, delay for a complete reoptimization cycle via the steady state control loop can reach 19 min in total.

B. Reoptimization via the Steady State Control Loop

In order to design and conduct comprehensive test scenarios covering all cases for our proposed system, we perform preliminary simulations with the LPO module. The optimization algorithm is validated using a test request based on our use-case application described in Section III. The service description is constructed with reference parameters obtained from CloudWatch measurement logs and listed in Table III. The generated test cases are constrained with decreasing latency limits starting from 4.0 s. The tests are conducted until the service request is declined by the LPO due to unachievable E2E latency (at 1.56 s). The horizontal bar plot in Fig. 7(a) depicts the resulting groupings for the applied limits (horizontal) and the assigned flavour for each function component (vertical), while Fig. 7(b) shows the predicted values of E2E latency, overall application cost and partial cost required to be paid to the cloud provider. The results align with our expectation as stricter latency limits enforce the LPO to utilize compute resources at the edge, otherwise prefer the cheaper but, in terms of E2E latency, underperforming public cloud. It can be observed that the jumps in the overall cost at 2.1 and 2.6 s correlate with the increases in the aggregated function execution time assigned to the edge, while the predicted latency values give close approximation to the upper latency limits, but always fall below them.

TABLE III
CONFIGURATION PARAMETERS USED IN THE EXPERIMENTS

LPO configuration			
Invocation delay	**Value**	**System parameter**	**Value**
Intra-Cloud	95 ms	Measurement update period	15 min
Edge-Cloud	180 ms	Threshold of profit gain	5%
Intra-Edge	5 ms	Edge/Cloud cost ratio	3

Reference parameters of the use case application						
	Grab	**Resize**	**Scale**	**Obj_det**	**Cut**	**Obj_det2**
Exec. time $[ms]$	785	25	5	240	20	285
Inv. rate $[1/s]$	1	1	1	1	1	5
Read $[ms]$		40			40	5
Write $[ms]$	30				5	5
Blocking $[ms]$	25	40	40	25	25	

Regarding the different deployment scenarios, we can also notice that only five distinct and feasible software layouts are distinguished and generated by the LPO, out of the 132 possible grouping options. (Since the number of noncrossing partitions of an n-element set/chain is given by the nth Catalan number $\mathcal{C}_n$, where n equals to the number of functions in our case, our use case application has $\mathcal{C}_6 = 132$ distinct layouts [32]). These results show that the LPO can also be used to calculate feasible application layouts for a given latency limit in advance, thus, significantly reducing the state space

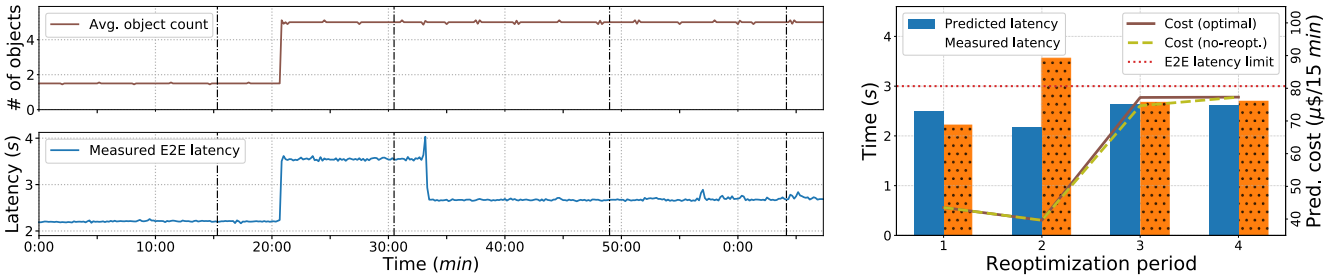


Fig. 8. Calculated and measured application metrics during *Phase 1* of our experiment. (a) CloudWatch logs with vertical markings at the end of each reoptimization period. (b) Predicted/measured values in the LPO.

of deployment options for additional layout reconfiguration features (see Section VII-C).

Based on the simulation outcomes, we construct a comprehensive and all-encompassing experiment to validate the behavior and performance of our system on AWS. Although, our proposed system implicitly manages the cloud-related performance fluctuations with the help of the control loops, there is no way to control the internal network characteristics and server workloads in a public cloud environment. For this reason, we select E2E latency and detected object count (an application specific metric) as the two input parameter which may vary in time and may affect the deployed application layout considerably. Therefore, our steady state control loop experiment is divided into two phases to observe the effect of these parameters' change separately, from a common initial state, while covering all the feasible deployment options.

For the experiment, we utilize dedicated requests generated during the previous simulations and apply two distinct input sources: a low (LO) and a high (HO) object count video stream resulting in 1 and 5 objects per frame in average, respectively. The detected object count directly influences the invocation rate between the last two functions, *Cut* and *Object detection stage 2*, as highlighted in Section III. The experiment is conducted in the Ireland region, whereas a dedicated VM with 8 vCPU in Frankfurt is set up as edge node. Each function is assigned to the runtime flavor with 1024-MB memory. The LPO is configured to apply a 15-min reoptimization period which is the time window used for periodically obtaining the measurement updates and for predicting the different layout costs, as well. The used system parameters are also summarized in Table III.

1) Phase 1: In the first phase of our experiment, we deploy our use case application using a reasonably permissive latency limit of 3.0 s and apply the LO video stream as test input. Then, we switch to the HO stream during reoptimization period 2, altering the application specific metric. At the initial deployment, the LPO decides to encompass all functions in a single group ($P_C = \{[1\text{–}6]_C\}$) resulting in 2.2 s measured E2E latency. Based on live measurements acquired directly from CloudWatch, shown in Fig. 8(a), we can state that both the deployment and detected object count metric remain unchanged after the first reoptimization period. As the input stream is altered from LO to HO, the detected object count, thus, the invocation rate of the last function rises. Consequently, the measured E2E latency of the active deployment layout exceeds the 3.0 s constraint, which is detected by the LPO at the end of the second period. At this point, the LPO initiates the service redeployment process. During the reoptimization, the LPO calculates a new optimal layout, while meeting the given latency constraint by moving the last component into a separate group ($P_{CC} = \{[1\text{–}5]_C, [6]_C\}$). The reason behind this decision is that the E2E latency can be reduced by eliminating the significant intragroup serializations and leveraging the platform-supported parallelization, in exchange of higher operational cost and additional intracloud invocation delay. Afterwards, the new layout remains optimal, keeping a steady state setup with an experienced 2.7 s E2E latency, and no other redeployment is performed in spite of the fluctuations in the measured values.

Fig. 8(b) sheds light on the decision process of the LPO from an internal point of view. It depicts the predicted cost in millionth dollar units (μ\$) and the E2E latency predicted at the beginning of the given periods along with the measured E2E latency acquired at the end of the periods for each step. It also visualizes the predicted cost of the non-reoptimization option, which is the recalculated cost of the layout in operation, but with the updated metrics, and used at the layout replacement decision. We can observe at period 2, when the measured value exceeds the limit and deviates from the predicted latency, that the LPO opts for a new layout, despite being 3.4% more expensive, in order to avoid the constraint violation. Fig. 8(b) also confirms that the predicted E2E latency aligns with the measured values in steady state, having only 0.8–2.6% difference.

2) Phase 2: In the second phase, we examine the effects of different E2E latency limits on the generated layouts, similarly to the simulation tests before. Continuing our experiment, we carry on with the HO video stream and set a 4.0 s latency limit to ensure the same initial cloud-only deployment as for *Phase 1*. After reaching the steady state (P_C), we deploy different layout options by iteratively sending new service requests with decreasing latency limits. The used arbitrary limits, which are 4.0, 3.0, 2.6, 2.1, and 1.7 s, are chosen from the simulations' results to cover all the generated deployment options. Between the deployments we leave enough time (at least 15 min) for our system to update the application metrics and confirm the steady state before proceeding to the next deployment.

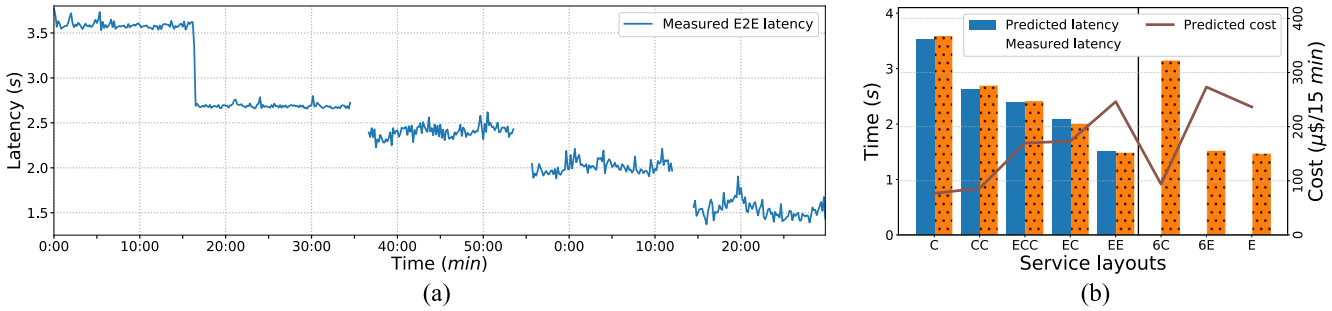


Fig. 9. Measured and predicted E2E latency and calculated cost values obtained during Phase 2 of our experiment. (a) CloudWatch log of the measured E2E latency. (b) Predicted and measured values in the LPO.

Fig. 9(a) presents the measured E2E latency acquired from CloudWatch for the entire duration of *Phase 2*. We can observe that the experienced latency values stepwisely follow the decrease of the applied limits, providing stricter E2E latency in each step. As it is examined in the previous phase, between the first two cloud-only deployment, P_C and P_{CC}, we can achieve around 0.9 s latency gain due to the platform-provided parallelization. By applying the next two constraints, we get mixed deployments of $P_{ECC} = \{[1–4]_E, [5]_C, [6]_C\}$ and $P_{EC} = \{[1–5]_E, [6]_C\}$, where the limits force the first several functions to be grouped together and assigned to the edge. With these layouts we can further reduce the E2E latency, despite introducing higher edge-cloud invocation latency. Utilizing edge resources moves processing closer to the video source, while keeping the last function in the cloud can still leverage its innate parallelization capabilities. Finally, applying the strictest latency limit results in a two-group, edge-only layout $P_{EE} = \{[1–5]_E, [6]_E\}$. Apart from the cloud-only scenarios, we can observe notable downtime during the layout replacement operation when the edge flavor is involved. The lack of support for seamless transition stems from the limitation of AWS CloudFormation, as described in Section VII-A. Although, supporting downtime-free replacement in the steady state control loop is matter of future work, our system offers rapid and seamless switching between layouts leveraging the runtime reconfiguration loop. Additionally, we also observed increased relative standard deviation (2.9–6.0%), which is calculated offline from exported CloudWatch logs, in the measured E2E latency compared to cloud-only layouts (0.8–1.1%). This stems from the presence of edge-cloud invocation in the deployments.

Fig. 9(b) depicts the predicted and measured latency values along with the predicted costs for the aforementioned layouts. For the sake of comparison, we also deploy and measure three manually assembled layouts, which represent the de facto, cloud-native deployment approaches of executing each code component separately (P_{6C}, P_{6E}), or encompassing them together (P_E). Applying these corner cases we can achieve similar E2E latency as with the corresponding cloud-only and edge-only layouts (P_C, P_{EE}) generated by the LPO, but at 2–2.4 times a higher cost (up to 22%). In addition, if we compare the LPO-calculated layouts to these manually crafted ones, while considering the associated latency limits, we can observe a significant 3.2 times cost increase in the worst case ($P_{EE} \leftrightarrow P_{CC}$). These differences in the layout costs confirm our argument, that is, an additional optimization mechanisms with precise models are required for operating serverless applications over public cloud in a cost-efficient manner. Moreover, it is worth highlighting that during *Phase 2* of the experiment, the predictions approximate the measured values well, including the mixed deployments, experiencing only 0.5–3.8% overestimation.

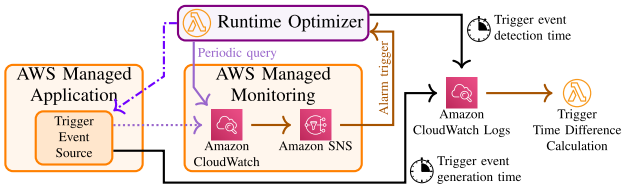


Fig. 10. Test scenario for measuring alert delay.

C. Dynamic Runtime Reconfiguration

As presented in Section VI-D2, two versions of the runtime reconfiguration loop are available: a push-based solution where Amazon CloudWatch (CW) sends out alarms to the RO, and a poll-based mechanism where the RO actively queries CW for limit violations. Both approaches are affected by the capabilities of CW. The former is limited by a 10 s, while the latter by a 1 s measurement window. CW also needs an undisclosed amount of time to consolidate metric data. As depicted in Fig. 10, we set up a test environment using our system to investigate detection time of limit violations. Our test application consisting of a single *Trigger Event Source* component is deployed in the cloud. Monitoring happens the same way as described by Section VI-C, via CW Metrics. The application component sends out trigger events that cause limit violations and logs their generation time. The RO also logs the time when it detects these violations. Time difference between the event generation and its detection is calculated by a separate Lambda function. Our measurements show that the effective feedback delay in case of the push-based option is 20.13 s, on average with 15.25 s minimum and 20.4 s maximum values based on our 100 tests. For the poll-based one, however, we can achieve 3.2 s average delay with 0.58 s minimum and 8.95 s maximum values. To determine total reconfiguration time of the application, we have to add another approximately 2 ms in both cases,

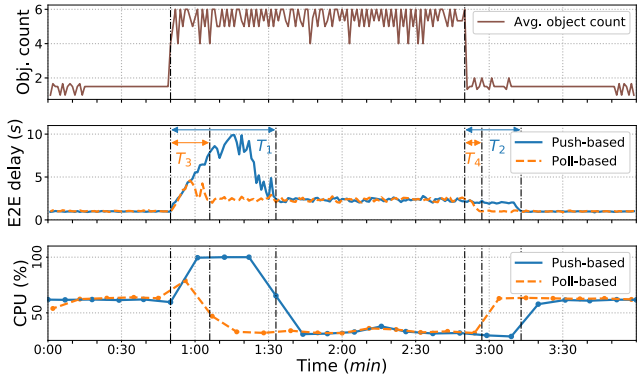


Fig. 11. Application and performance metrics logged into CloudWatch during the dynamic runtime reconfiguration experiment.

when communicating with cloud functions. This delay is due to data exchange between the RO and function Wrappers via Redis instances. Edge reconfiguration is slower, as the 2 ms exchange latency is increased by network delay between the RO's cloud region and the edge location.

We also investigated the performance of component offloading from edge to cloud in our object detection application with both available options. We set up our edge node having four CPU cores in the *eu-central-1* (Frankfurt) AWS region while *us-west-2* (Oregon) was chosen for cloud execution. The sample video was streamed from Budapest, outside of AWS, with a sample frame rate of 2/s. In this experiment, we use two layouts from those given by the LPO in Section VII-B: $P_{EE} = \{[1–5]_E, [6]_E\}$ as initial deployment, and $P_{EC} = \{[1–5]_E, [6]_C\}$ for offloading *Object Detection Stage 2* to the cloud. RO-driven offloading is triggered by the *object count* application level metric, supplied by the *Cut* function, surpassing the number of the edge node's CPU cores. In case of the application being triggered more frequently than the minimum execution time of the Object Detection Stage 2 function, the metric can signal an edge node overload condition. In such cases, concurrent instances of the function would consume more CPU resources than available.

Fig. 11 depicts the effect the different alarm detection options have on the application performance. Displayed metrics are taken from CW and in case of the object count and E2E delay, use a 1 s measurement window for aggregation. In case of CPU load, however, the Edge Monitor component logs the aggregated utilization metric less frequently. As expected, the poll-based mechanism outperforms the push-based in every regard. As the object count in the video stream increases, the push-based loop is slow to react and the edge CPU load reaches 100% while E2E latency tops at 9.87 s. The poll-based option experiences far lower rise in the E2E latency (with a maximum of 2.75 s) and manages to to keep the CPU load on the edge in check, with a maximum of 79% which is a 16% rise compared to normal behavior. After the end of application reconfiguration and function cold start latency, the E2E latency settles at 2.3 s (up from the original 1 s) and edge CPU usage at 33%. The 16 s transient time of the poll-based option is significantly shorter than the 43 s of the push-based (refer to the intervals T_3 and T_1, respectively, in the figure). The comparatively long transient time is caused by the increased execution time on the edge node as well as clod start delay for starting up functions in the cloud. After the object count decreases below four, the RO shifts Object Detection Stage 2 back to the edge. As both E2E latency and edge CPU usage return to their original values, we can observe that transition, in case of the poll-based reconfiguration, is unsurprisingly quicker again ($T_4 = 7$ s compared to the push-based version's reaction time of $T_2 = 16$ s).

Based on our tests, it is clear that although push-based application reconfiguration is cheaper to realize, it might not be sufficient for avoiding edge node overload. Depending on application characteristics, the poll-based option can improve performance, but with higher invocation rates that might fail as well. As our implementation reaches the limits of CW, if an even smaller reaction time is required, a different solution should be used for collecting application metrics.

VIII. Conclusion

In this article, we adapted the cloud native programming and serverless operating techniques for latency sensitive IoT applications. A novel system was proposed on top of public cloud platforms providing serverless solutions for central and edge domains. The general approach was applied to Amazon's AWS leveraging its FaaS offerings, Lambda and Greengrass. Our main findings are summarized as follows.

1) We argue that application latency and operational costs are significantly affected by the grouping of the constituent functions (how to group and package user functions into FaaS platform artifacts); the selected flavors providing the runtime for the functions; and the placement of the components (central cloud or edge domains). Developers or operators of latency sensitive applications can benefit from defining their expectations on latency and cost, while scaling to current workload is delegated to the cloud providers.
2) We propose to add an optimization component on top of public cloud stacks to optimize deployment costs while keeping soft latency boundaries. This component controls the deployment via available services and exposed APIs. Such a control loop allows supervising serverless deployments in the range of minutes or tens of minutes, which is sufficient to follow daily profiles and usage trends.
3) In order to support control on lower timescales, the platform and the FaaS runtime are required to provide direct configuration interfaces for swapping layouts. We presented an extension to a state-of-the-art FaaS platform implementation. As a result, control within a few seconds can also be realized if different deployment options are onboarded in advance.
4) Instrumentation is needed to implement the detailed monitoring required as input for optimization. Customization of cloud monitoring offers a simple implementation, which enables capturing the

performance characteristics of the deployed applications and the underlying platforms with acceptable accuracy. Therefore, adequate models of applications and platform components can be established, hence such a monitoring system fulfills all requirements to enable closed-loop control for latency sensitive serverless applications.

REFERENCES

[1] *Amazon Web Services*. Accessed: May 11, 2020. [Online]. Available: https://aws.amazon.com

[2] *Apache OpenWhisk*. Accessed: May 11, 2020. [Online]. Available: https://openwhisk.apache.org

[3] *AWS CloudFormation*. Accessed: May 11, 2020. [Online]. Available: https://aws.amazon.com/cloudformation/

[4] *AWS Compute Optimizer*. Accessed: May 11, 2020. [Online]. Available: https://aws.amazon.com/compute-optimizer/

[5] M. Bravetti, S. Giallorenzo, J. Mauro, I. Talevi, and G. Zavattaro, "Optimal and automated deployment for microservices," in *Fundamental Approaches to Software Engineering*. Cham, Switzerland: Springer, 2019, pp. 351–368.

[6] *CNCF: Cloud Native Computing Foundation*. Accessed: May 11, 2020. [Online]. Available: https://www.cncf.io

[7] 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.

[8] A. Das, S. Imai, M. P. Wittie, and S. Patterson, "Performance optimization for edge-cloud serverless platforms via dynamic task placement," 2020. [Online]. Available: arXiv.abs/2003.01310.

[9] *Densify: Enterprise-Class Resource Management, Optimization & Control*. Accessed: May 11, 2020. [Online]. Available: https://www.densify.com

[10] S. Eismann, J. Grohmann, E. van Eyk, N. Herbst, and S. Kounev, "Predicting the costs of serverless workflows," in *Proc. ACM/SPEC Int. Conf. Perform. Eng. (ICPE)*, Mar. 2020, pp. 265–276.

[11] T. Elgamal, "Costless: Optimizing cost of serverless computing through function fusion and placement," in *Proc. IEEE/ACM Symp. Edge Comput. (SEC)*, Oct. 2018, pp. 300–312.

[12] M. Fotouhi, D. Chen, and W. J. Lloyd, "Function-as-a-service application service composition: Implications for a natural language processing application," in *Proc. 5th Int. Workshop Serverless Comput. (WOSC)*, Dec. 2019, pp. 49–54.

[13] *Google Cloud Platform*. Accessed: May 11, 2020. [Online]. Available: https://cloud.google.com

[14] L. Hyafil and R. Rivest, *Graph Partitioning and Constructing Optimal Decision Trees Are Polynomial Complete Problems*. IRIA, New Delhi, India, 1973.

[15] *IBM Cloud*. Accessed: May 11, 2020. [Online]. Available: https://www.ibm.com/cloud

[16] B. W. Kernighan, "Optimal sequential partitions of graphs," *J. ACM*, vol. 18, no. 1, pp. 34–40, Jan. 1971.

[17] *Knative: Kubernetes-Based Serverless Platform*. Accessed: May 11, 2020. [Online]. Available: https://knative.dev

[18] *Kubernetes: Automated Container Deployment*. Accessed: Jan. 31, 2018. [Online]. Available: https://kubernetes.io

[19] J. Kuhlenkamp and M. Klems, "Costradamus: A cost-tracing system for cloud-based software services," in *Service-Oriented Computing*. Cham, Switzerland: Springer, 2017, pp. 657–672.

[20] P. Leitner, J. Cito, and E. Stöckli, "Modelling and managing deployment costs of microservice-based cloud applications," in *Proc. 9th Int. Conf. Utility Cloud Comput. (UCC)*, Dec. 2016, pp. 165–174.

[21] K. Mahajan, D. Figueiredo, V. Misra, and D. Rubenstein, "Optimal pricing for serverless computing," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2019, pp. 1–6.

[22] N. Mahmoudi, C. Lin, H. Khazaei, and M. Litoiu, "Optimizing serverless computing: Introducing an adaptive function placement algorithm," in *Proc. 29th Annu. Int. Conf. Comput. Sci. Softw. Eng. (CASCON)*, Nov. 2019, pp. 203–213.

[23] F. Manne and T. Sorevik, "Optimal partitioning of sequences," *J. Algorithms*, vol. 19, no. 2, pp. 235–249, Sep. 1995.

[24] *Microsoft Azure*. Accessed: May 11, 2020. [Online]. Available: https://azure.microsoft.com

[25] J. Misra and R. Tarjan, "Optimal chain partitions of trees," *Inf. Process. Lett.*, vol. 4, no. 1, pp. 24–26, Sep. 1975.

[26] *OpenCV*. Accessed: May 11, 2020. [Online]. Available: https://opencv.org

[27] *OpenFaaS*. Accessed: May 11, 2020. [Online]. Available: https://docs.openfaas.com

[28] I. Pelle, J. Czentye, J. Dóka, 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.

[29] 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. Exhibit. (OFC)*, Mar. 2020, pp. 1–3.

[30] P. Porambage, J. Okwuibe, M. Liyanage, M. Ylianttila, and T. Taleb, "Survey on multi-access edge computing for Internet of Things realization," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 4, pp. 2961–2991, 4th Quart., 2018.

[31] J. Ren, D. Zhang, S. He, Y. Zhang, and T. Li, "A survey on end-edge-cloud orchestrated network computing paradigms: Transparent computing, mobile edge computing, fog computing, and cloudlet," *ACM Comput. Surveys*, vol 52, no. 6, pp. 1–36, Oct. 2019.

[32] N. J. A. Sloane. *The On-Line Encyclopedia of Integer Sequences, Sequence A000108*. Accessed: Sep. 20, 2020. [Online]. Available: http://oeis.org/A000108

[33] *Stackery*. Accessed: May 11, 2020. [Online]. Available: https://www.stackery.io

[34] *Terraform by HashiCorp*. Accessed: May 11, 2020. [Online]. Available: https://www.terraform.io

[35] *The Serverless Application Framework*. Accessed: May 11, 2020. [Online]. Available: https://serverless.com/

[36] S. Winzinger and G. Wirtz, "Model-based analysis of serverless applications," in *Proc. 11th Int. Workshop Model. Softw. Eng. (MiSE)*, May 2019, pp. 82–88.

[37] A. Yousefpour *et al.*, "All one needs to know about fog computing and related edge computing paradigms: A complete survey. *J. Syst. Architect.*, vol. 98, pp. 289–330, Sep. 2019.

[38] W. Yu *et al.*, "A survey on the edge computing for the Internet of Things," *IEEE Access*, vol. 6, pp. 6900–6919, 2018.

István Pelle (Member, IEEE) received the M.Sc. degree in computer science from the Budapest University of Technology and Economics, Budapest, Hungary, 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.

János Czentye (Graduate Student Member, IEEE) received the M.Sc. degree (Highest Hons.) in networks and services from Budapest University of Technology and Economics, Budapest, Hungary, in 2014, where he is currently pursuing the Ph.D. degree.

Since then he worked with the Department of the Telecommunications and Media Informatics contributing in several research projects and gained wide knowledge about SDN, NFV, and microservice technologies. His current Ph.D. research focuses on cloud-native service modeling and provisioning.

János Dóka received the M.Sc. degree from the Budapest University of Technology, Budapest, Hungary, in 2020.

He is a Member of the High Speed Networks Laboratory, Department of Telecommunications and Media Informatics and the MTA-BME Network Sofwarization Research Group. His current research focuses on network function virtualization and serverless computing.

András Kern received the M.Sc. degree in computer science and the Ph.D. degree from the Budapest University of Technology and Economics, Budapest, Hungary, in 2003 and 2008, respectively.

During the academic years his key topics was optimization of Ethernet and optical transport networks. After his Ph.D., he joined Ericsson, Budapest, in 2007, as a Researcher. His initial research focus was network control planes, like GMPLS, and software defined networking. He was a Technical Leader in the FP7 SPARC European project focusing on OpenFlow implementations and prototyping. He has coauthored conference and journal papers and holds several patents. Since then, his research area has been extended with cloud infrastructures and orchestration.

Balázs P. Gerő (Associate Member, IEEE) received the M.Sc. degree in computer science from the Budapest University of Technology and Economics, Budapest, Hungary, in 1998.

He joined Ericsson Research, Budapest, in 2000. His research included a variety of fields, such as network management, transport network protocols, software defined networks, and data centers. He is currently active in orchestration system design of large scale distributed/edge cloud systems.

Balázs Sonkoly (Associate Member, IEEE) received the M.Sc. and Ph.D. degrees in computer science from the Budapest University of Technology and Economics (BME), Budapest, Hungary, 2002 and 2010, respectively.

He is an Associate Professor with BME, where he is the Head of MTA-BME Network Softwarization Research Group. He has participated in several EU projects (FP7 OpenLab, FP7 UNIFY, H2020 5G Exchange) and national projects. His current research activity focuses on cloud/edge/fog computing, NFV, SDN, and 5G.

Dr. Sonkoly was the demo Co-Chair of ACM SIGCOMM 2018, EWSDN'15,'14, and IEEE HPSR'15.