Received 26 August 2022, accepted 10 September 2022, date of publication 14 September 2022, date of current version 21 September 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3206433

RESEARCH ARTICLE

The Shape of Your Cloud: How to Design and Run Polylithic Cloud Applications

LASZLO TOKA, (Member, IEEE)
MTA-BME Information Systems Research Group, MTA-BME Network Softwarization Research Group, Faculty of Electrical Engineering and Informatics, Budapest University of Technology and Economics, H-1117 Budapest, Hungary

e-mail: toka.laszlo@vik.bme.hu

This work was supported in part by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund through Project 128233 under the FK_18 funding scheme, Project 135074 under the FK_20 funding scheme, and Project 2019-2.1.13-TÉT_IN-2020-00021 under the 2019-2.1.13-TÉT-IN funding scheme; and in part by the ÚNKP-22-5 New National Excellence Program of the Ministry for Innovation and Technology from the Source of the National Research, Development and Innovation Fund. The work of Laszlo Toka was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

ABSTRACT Nowadays the major trend in IT dictates deploying applications in the cloud, cutting the monolithic software into small, easily manageable and developable components, and running them in a microservice scheme. With these choices come the questions: which cloud service types to choose from the several available options, and how to distribute the monolith in order to best resonate with the selected cloud features. We propose a model that presents monolithic applications in a novel way and focuses on key properties that are crucial in the development of cloud-native applications. The model focuses on the organization of scaling units, and it accounts for the cost of provisioned resources in scale-out periods and invocation delays among the application components. We analyze dis-aggregated monolithic applications that are deployed in the cloud, offering both Container-as-a-Service (CaaS) and Function-as-a-Service (FaaS) platforms. We showcase the efficiency of our proposed optimization solution by presenting the reduction in operation costs as an illustrative example. We propose to group similarly low scale components together in CaaS, while running dynamically scaled components in FaaS. By doing so, the price is decreased as unnecessary memory provisioning is eliminated, while application response time does not show any degradation.

INDEX TERMS Cloud-native application, container-as-a-service, function-as-a-service, resource footprint, invocation latency, application scaling, microservice architecture.

I. INTRODUCTION

Cloud computing has transformed the scene of IT in less than two decades. The amazing technological evolution both in terms of computing and networking enabled several new applications and services running at extremely large scale on top of different cloud platforms. Public cloud platforms, such as Amazon Web Services [1], Google Cloud Platform [2] and Microsoft Azure [3] are capable of providing an ''arbitrary'' number of virtual resources on demand making use of virtualization techniques and resource management mechanisms. Well-designed data centers contain all the necessary physical assets, including thousands of blade servers and network devices, and the burden of operation is delegated to the cloud providers, ensuring high reliability and high performance. The cloud application owner has nothing else to do except for selecting the best-suited cloud service offering and deploying their application in the cloud to go live: this means i) zero initial investment as cloud services offer pay-as-you-go schemes, and ii) that there is no need to plan for maximum capacity as resource provisioning is adaptively flexible, usually automatically scalable, often depicted as being completely elastic.

Besides the non-existent capital expenditures into infrastructure on the cloud tenant's side, deploying applications into the cloud also has the benefit of low operating costs. The reason behind the effective operation is the economies of scale of compute infrastructure in data centers, and the

The associate editor coordinating the review of this manuscript and approving it for publication was Bijoy Chand Chand Chatterjee.

shared resources among a myriad of tenants, resulting in a time multiplexed usage of resources. On the other hand, the downside of the co-location and the relatively complex infrastructure is that extra delay might be introduced in the cloud application end-to-end latency due to several reasons. First, in case there are not enough resources strictly dedicated to the application, its runtime is prolonged. Second, distributed applications, often used in a cloud environment, have to count with an inter-component invocation delay. Third, the latter are exacerbated by the inter-node network latency of the data center, if application components are orchestrated to run on different compute nodes [4].

Web services and applications are now mostly deployed into cloud. These web applications typically follow the microservice architecture, where the monolithic software is broken down into smaller, independently managed components, usually realized by software containers that are separately orchestrated and scaled by the cloud platform enabling optimal resource utilization [5]. The cloud resource provisioning is continuously adapted to fit the demand for service, e.g., the time varying number of client requests hitting the application ingress. Allocating resources dynamically to constituent containers and scaling them properly on demand is a challenging task [6]. The scaling logic can be driven by various service management goals, e.g., either minimizing resource usage while sustaining a given service quality target, or minimizing Service Level Agreement (SLA) violations no matter the price paid for the provisioned resources. We argue that the application design should be inherently aligned with the operational features of the cloud setup: when scaling the application carefully to the point (not higher to avoid unnecessary resource costs, not lower to avoid rejected requests, i.e., QoS degradation), the cost-aware application designer should also account for the overhead that is introduced by the polylithic (opposed to monolithic) design in order to minimize the total footprint of a scaled out application. Similarly to microservices or service-oriented architecture, we make the case of polyliths that cover all the applications consisting of granular services. When software is packaged as a polylith, its modularity is high, both in terms of development and operation. In this work we focus on the latter: during scale out regimes separate software modules can be scaled independently of each other, resulting in an optimal cloud resource usage scheme. In contrast, if modules are packaged together and therefore scaled together, scale-out actions may lead to unnecessary resource, e.g., memory, consumption. However, the co-location of application components within the cloud results in lower operational delays, hence better QoS for the application user.

The cloud-native paradigm [7] aims to build and run applications exploiting all the benefits of cloud computing service models, but one must be aware of their choices in terms of application design and deployment options [8], these include several techniques and concepts, from microservices across DevOps to serverless architectures. 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 CaaS computing model or the FaaS paradigm, depending on the granularity level that the developer can consider when creating the software. Our contribution in this paper is two-fold: i) we propose a model to define the trade-off between response time performance (i.e., latency) vs. cloud resource footprint when it comes to the decision about designing, packaging and deploying a cloud native application, and ii) we evaluate the model in an illustrative example to provide insights to the extreme sides of this trade-off through a cost analysis of public CaaS and FaaS offerings.

This paper is organized as follows: in Section II we present the main differences between cloud services that offer various application deployment options for the cloud tenant and we give an overview of relevant research findings; building on those observations we propose an analytical cost model that accounts for deployment-related costs in Section III; afterwards we analyze illustrative examples with optimized model cost parameters and currently advertised cloud service fees in Section IV; finally, we conclude the paper in Section V.

II. RELATED WORK

Virtualization techniques have brought abrupt changes not only to web applications, but also to how telecommunications systems are designed. Network Function Virtualization (NFV) offers the opportunity to move the software running on traditionally expensive custom physical nodes into the cheap multi-purpose cloud, resulting in fast configuration and development cycles and cost-efficient scalability. The emergence of concepts like cloud computing, Software-defined Networking (SDN), and, ultimately, NFV, raises new possibilities for the management of telecommunications applications with a positive impact in terms of agility and cost. From a telecommunications viewpoint these concepts can help to both reduce operational expenditure and open the door to new business opportunities [7], [9]. As a specific example, let us take the IP Multimedia Subsystem (IMS) that enables various types of media services to be provided to end-users using common, IP-based protocols. To protect and hide vulnerable details of the mobile operator's core network, the Border Gateway Function (BGF) is placed between the access and core networks providing pinhole firewall and Network Address and Port Translation (NAPT) functionality. As such, it is responsible for filtering and transferring the RTP (Real-time Transport Protocol) based media streams exchanged by mobile subscribers. Traditional telecommunications nodes couple the states of the user sessions with the physical executors. Accordingly, if a physical entity fails, then the handled user sessions get lost. On the other hand, it is also common that each functionality is implemented on top of a dedicated hardware resource, e.g., board, DSP chip, that overall makes the system distributed and inherently more robust against hardware failures. In case of a failure, only those calls are affected that shared the same resources, which is an insignificant number of sessions. However, this does not apply to the cloud anymore where a virtual machine (VM) can serve tens of thousands of sessions, relying on a single hardware infrastructure. In a cloud deployment of BGF, each user session is tied to a particular instance, i.e., connected to a particular IP address/port of the BGF VM instance. In such a system, if a VM fails all the sessions get lost, which impacts a large number of subscribers; this is not acceptable [10]. However, the modern NFV ecosystem is fundamentally *stateless*; if the Virtual Network Functions (VNFs) do not maintain persistent state on their own, then scale-in/scale-out and other fail-over events are less complex to handle, improving overall elasticity, scalability, resiliency and performance.

Adopting the stateless design in its core concept, FaaS, often referred to as serverless computing, has recently become one of the most popular paradigms in cloud computing. The paradigm emerged not only as a pricing technique, but also as a programming model promising to simplify developing for the cloud. Using FaaS, developers do not need to care about resource allocation, scaling, or scheduling, since the platform handles these. However, the road of an effective transition from monolithic applications to the architecture most suited to FaaS platforms is by no means trivial. The architecture of the applications needs to be changed in order to take full advantage of the underlying platforms. Numerous projects managed by companies and academic institutions have built FaaS platforms, but the most widely used ones are underneath the FaaS services offered by IT giants, i.e., Amazon's AWS Lambda [1], Google Cloud Functions [2] and Microsoft Azure Functions [3]. Most of these platforms operate with container technologies; the user's executable code is packed into a container that is instantiated when the appropriate function call request first arrives. With this relatively lightweight technology it is easy to achieve process isolation and resource provisioning.

The evolution from IaaS (Infrastructure as a Service) to FaaS did not take too long, and there are pros and cons for all types of cloud services. As depicted in Figure 1, there is a diminishing burden on cloud tenants in terms of application management from IaaS through CaaS to FaaS, although this, of course, comes at an increased price of resource units; built-in features and their resource usage are incorporated into those prices. The higher level the service type, the fewer functionalities and assets the tenant has to provide for themselves, as depicted by the vertical labels in Figure 1. The author of [11] suggests an economic packaging of application modules in FaaS: function fusion has the disadvantage of making the application less modular and maintainable, however, it is an effective way to reduce the price tag, when transition cost, i.e., an extra cost in AWS Step Functions [1] for each state transition during the execution of the predefined workflow, dominates the function execution cost.

Besides the pricing aspects, resource footprint and latency overhead have also been in the focus of the research community; particularly the cold-start latency FaaS platforms suffer from when a new VM or container has to be launched to run the invoked task [12]. The size of the image to mount, the number of libraries and dependencies all have an impact on this latency [4]. Even though communication is significantly faster the closer the parties are (e.g., same data center, same rack, same server machine, same process), currently available platforms miss out on co-locating entities that often communicate with each other [4].

To the best of our knowledge, the trade-off of scale-out footprint and latency has never been addressed within the packaging options of cloud-native microservices. These two important aspects, which are at odds with each other, are tackled in this paper. On one hand, the co-location of application components within the cloud results in lower operational delays, hence better QoS for the application user. We consider the strictest affinity policy that can be expressed in public clouds today: packaging those components together within a container or a function that must be run on the same hardware. On the other hand, with more packaging comes less modularity, which results in superfluous resource consumption especially during scale-out regimes. Our focus is particular to CaaS and FaaS, therefore, IaaS, Platform- (PaaS), and Software-as-a-Service (SaaS) systems (a high level overview of those is depicted in Figure 1) are out-of-scope. The reason for this is that we are particularly interested in cloud services that offer application-agnostic, automatically scaled deployment options for the tenant. PaaS and SaaS are not well-suited to run proprietary code in the cloud, e.g., telecommunications core functions of a mobile operator.

Emerging from the agile practitioner communities, the microservice-oriented architecture emphasizes implementing and employing multiple small-scale and independently deployable microservices, rather than encapsulating all function capabilities into one monolithic application. Microservices architecture has become enormously popular because traditional monolithic architectures no longer meet the needs of scalability and rapid development cycle. However, performing the migration process is not trivial. Most systems acquire too many dependencies between their modules, and thus cannot be sensibly broken apart. It is for this reason that studies that provide information associated with the migration process to practitioners are necessary. A key challenge in this context is the extraction of microservices from existing monolithic code bases. While informal migration patterns and techniques exist, there is a lack of formal models and automated support tools in that area. Reference [13] tackles that challenge by presenting a formal microservice extraction model to allow algorithmic recommendation of microservice candidates in a refactoring and migration scenario. The results show that the produced microservice candidates lower the average development team size down to half of the original size or lower. Furthermore, the size of recommended microservice conforms with microservice sizing reported by empirical surveys and the domain-specific redundancy among different microservices is kept at a low rate. In [14] the authors address the same challenge: they propose a top-down analysis approach and develop a dataflow-driven

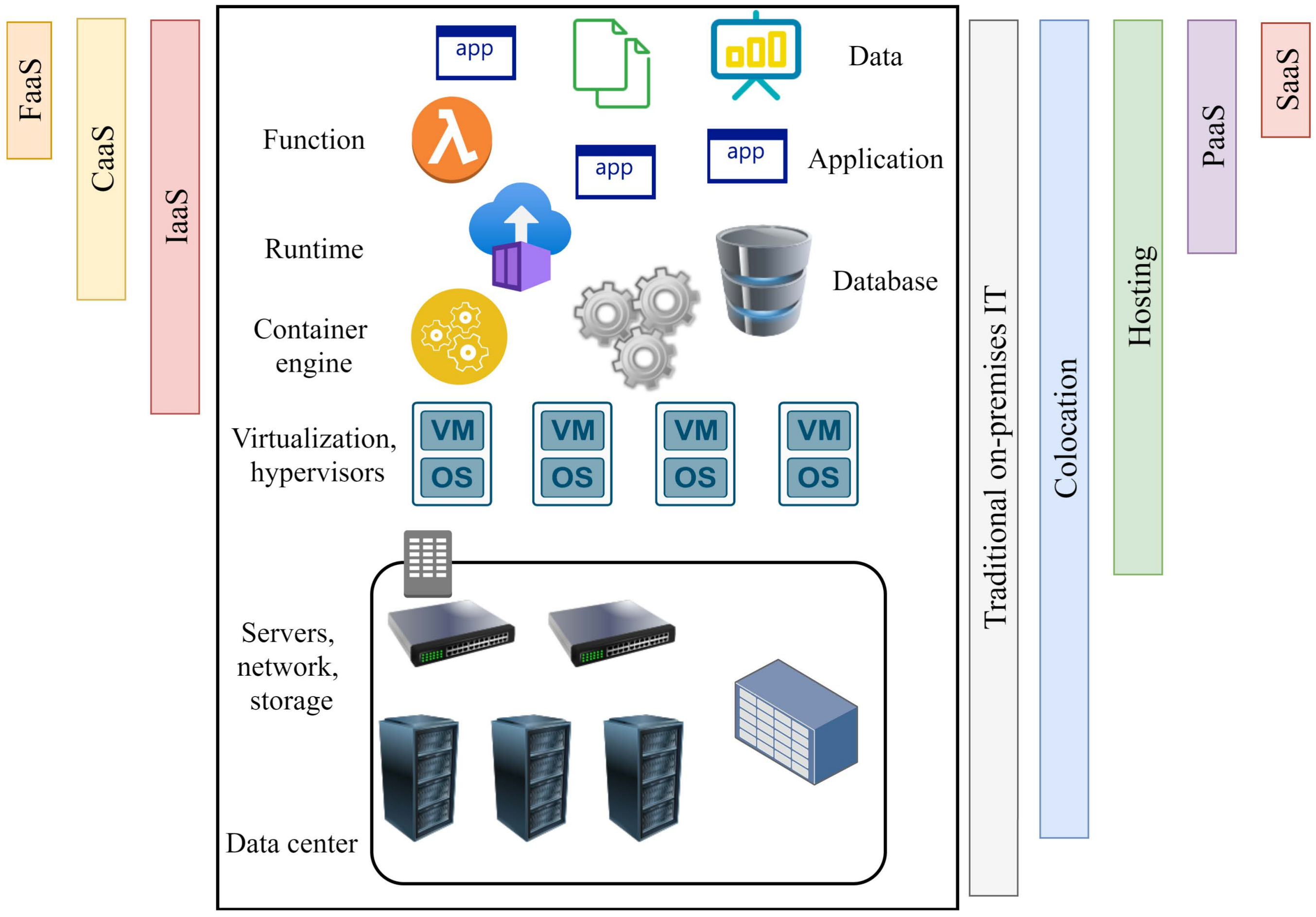

FIGURE 1. Relation between various cloud services (vertical labels) and the functionalities (horizontal labels) that are left for the tenant to take care of.

decomposition algorithm. In brief, a three-step process is defined: first, engineers together with users conduct business requirement analysis and construct a purified while detailed dataflow diagram of the business logic; then, their algorithm combines the same operations with the same type of output data into a virtual abstract dataflow; finally, the algorithm extracts individual modules of "operation and its output data" from the virtual abstract dataflow to represent the identified microservice candidates. Based on the ambiguity of determining the optimum size of a microservice, in [15], the authors propose a conceptual methodology to partition a microservice based on domain engineering technique. Domain engineering identifies the information needed by a microservice, services needed for microservice functionality and provides description for workflows in the service. In [16] the authors report on migration practices towards the adoption of microservices in industry, specifically on (i) the performed activities, and (ii) the challenges faced during the migration. Daoud *et al.* [17] proposes an approach combining different independent models that represent a business process's control dependencies, data dependencies, semantic dependencies, respectively. The approach is also based on collaborative clustering. Reference [18] analyzes 20 migration techniques proposed in the literature. Results show that most proposals use approaches based on design elements as input; 90% of the proposals were applied to object-oriented software (Java being the predominant programming language); and that the main challenge is to perform the database migration. Compared to this vast body of research, our work is novel in the sense that it addresses the repercussion of dissecting a monolith into too many microservices: the response time performance of the application potentially worsens due to added delay of inter-microservice communication. Therefore, we propose a model to take such aspects also into account when designing a microservices-based cloud-native application.

III. PROPOSED METHOD: THE COST MODEL OF RUNNING A POLYLITH

Cloud deployment enables easy scaling to the actual application load. The cost of scaling is greatly determined by the organization of application into scaling units. In this section, we propose an analytical model to reflect the resource footprint overhead at scaling, and the latency overhead of organizing application code into several deployment units. We show that these cost terms are opposing forces that steer the application designer towards organizing the application code in an optimal packaging setup for reaching the sweet spot in terms of operational expenses and application response time performance.

Let us start by modeling the application as a finite set of sub-application parts, called as modules, that can be grouped into separate deployment units, e.g., containers or functions, denoted as scaling units or groups from now on. We assume that any combination of these modules can be packaged together, i.e., in case all of them are grouped together, we arrive back to the monolithic application (from a cloud deployment perspective). We model the modules by their average memory consumption over a given time period, e.g., an average of 2396 MB memory footprint integrated over 1 hour. We denote those values by $r_1, r_2, \ldots, r_n$. Now, for each of these modules, let us further define a value s that reflects its time-weighted average scaling factor in a steady state operation of the deployed application over the given time period mentioned above, e.g., an average scaling factor of 2.67 due to being scaled to 3 instances for 40 minutes, and having 2 instances for the remaining 20 minutes. The value s_i can be interpreted as the number of invocations of the respective module i of the application which can be run in parallel, i.e., in multiple instances. Application codes that have to run sequentially, i.e., no possible way of parallelism of the module, cannot have a scaling factor s larger than 1.

The problem setup is now translated into dissecting a given application into modules such that each module is packaged separately, e.g., as one or more containers to be grouped in a pod under Kubernetes, the most widely used CaaS manager, or a function that will be executed in a FaaS platform. Along the process of dissecting a monolithic application into modules, then organizing them into scaling units, the application designer has to focus on the following operational aspects that appear consequently: i) memory footprint of scaling units, ii) overhead of invocation latency between scaling units.

Scaling cost is due to the amount of code that is scaled out unnecessarily in case of executing multiple instances of a scaling unit.

Communication cost An extra latency is added to the application execution time due to invocations across scaling units. Furthermore, when breaking modules and putting them in parallel resources, the increased network latency, as well as reduced reliability, requires careful reasoning about consistency.

We illustrate the scaling cost in Figure 2: we depict the modules of an application by 5 rectangles. On the *x-axis* we mark the memory footprint of one instance of each module. The ticks represent 200MB of memory, so the leftmost module's memory footprint is 600MB. We place the modules' memory footprint values right next to each other. On the *y-axis* we denote the number of instances that run in parallel for each module, i.e., the average scaling factor. If there is no parallelization for the modules of a given application, then the height of every rectangle is 1. In the illustrative case shown

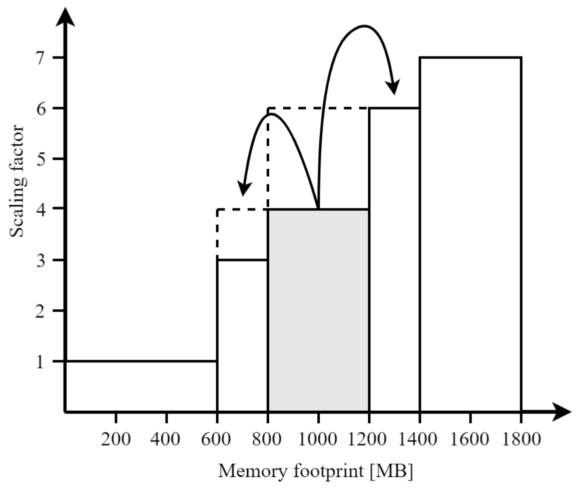

FIGURE 2. Illustration of the scaling overhead model through an example application consisting of 5 modules.

in Figure 2, the scaling factor is 1, 3, 4, 6, and 7 for the individual modules from left to right, respectively.

The overall memory footprint of the whole application is the area under the curve, i.e., the sum of the rectangles' areas. By separating the application into modules and scaling those modules with different factors, the end-to-end application execution time, e.g., response time for a web request, is greatly reduced, but the price to pay is the above-mentioned overhead: inter-module latency. Let us see how the memory footprint changes if some modules are merged into a joint scaling unit. For example, if the application designer decides to add the module in the middle of Figure 2 to another module which has either a lower or a higher scaling factor. We assume that the designer does not want to make any compromises on the execution speed at scale out regimes, so in the former case, the applied scaling factor will be the one dictated by the middle module; in the latter case it will be that of the other module. In both cases there will be modules to be scaled to an unnecessary extent, leading to an extra scaling cost. In the specific example of Figure 2 the extra cost is represented by the dashed rectangles: if the middle module is packaged together with its left neighbor, then the scaling factor of the merged module will be that of the middle module; similarly, merged with its right neighbor, this latter will dictate the scaling factor.

The overall operational cost is therefore increasing by merging different modules of the application that require diverse scaling factors. However, merging them might be necessary to meet the latency requirements dictated by application SLA. The questions naturally arise: how many scaling units should the application designer account for, and which modules should be packaged together into those? We make these statements in the following and provide hints on their proofs.

Lemma 1: For any given number of scaling units, the scaling cost is minimized by grouping the modules together into scaling units following the order of their scaling factors.

Proof: The proof is indirect. Let us assume an optimal arrangement of modules into scaling units in terms of minimal scaling cost. Without the loss of generality, let the scaling units operate at increasingly ordered $s_1^g, s_2^g, \dots, s_x^g$ scaling rates. Furthermore, let module i have s_i scaling rate and belong to scaling unit j with $s_i < s_{j-1}^g < s_j^g$. In this case the scaling cost can be decreased by re-arranging module i (that does not hold the largest scaling factor in its own group j) into another scaling unit for which the scaling factor is higher than its value, but lower than its original group's scaling factor, e.g., into group $j-1$, contradicting the initial assumption about optimal arrangement. ■

In Figure 3 we depict the modules of an illustrative example application, ordered by their scaling factors and grouped into scaling units along the ordering.

Lemma 2: For any given number of scaling units with minimized scaling cost, for the scaling factor s_b of the modules b on group borders

$$s_b \geq \frac{s_j^g r_b + s_{j-1}^g \sum_{i \in j-1} r_i}{r_b + \sum_{i \in j-1} r_i} \quad \forall j < x \tag{1}$$

holds.

Proof: In an increasing ordered setting of the modules as suggested by Lemma 1, the borders of scaling groups are left to such modules b for which the jump in scaling factor is larger than the scaling factor increment (relative to the module's) of the right hand side group multiplied by the width of the module, and divided by the width of the left hand side group. Specifically, assuming x scaling units, for any neighboring scaling unit pair $j-1, j$ for which $1 < j < x$, the following inequality must hold:

$$\left(s_b - s_{j-1}^g\right) \sum_{i \in j-1} r_i \geq \left(s_j^g - s_b\right) r_b. \tag{2}$$

It is straightforward to see that in case this inequality does not hold, then the area of the e.g., left hand side rectangle depicted by dashed lines in Figure 2, which is expressed by $\left(s_b - s_{j-1}^g\right) \sum_{i \in j-1} r_i$, would be greater than that of the right hand side rectangle, which is equal to $\left(s_j^g - s_b\right) r_b$, resulting in higher scaling cost, thus contradicting with the initial assumption of being at the border of optimal grouping. Equation 1 is then derived from Equation 2 by rearranging the terms. ■

In summary, there is a relatively large jump between the scaling groups' scaling factors in a setting that is optimized for scaling cost. The authors of [19] and [20] found that a small percentage of microservices are hot-spots in call graphs, specifically, about 5% of microservices are multiplexed by more than 90% of online services in Alibaba clusters, which creates such large differences between scaling factor values, ideal for marking the borders of scaling groups. For an analytically tractable model, in the next statement the modules are assumed to be infinitesimally small, and the scaling factor is interpreted as a differentiable continuous function over the variable that depicts the cumulative resource demand of the modules sorted in the increasing order of their scaling factors.

Theorem 1: In the continuous model of module resource demand ρ and the scaling factor as its function $\sigma(\rho)$,

$$\frac{d\sigma}{d\rho}\rho_L + \sigma - \sigma_R = 0 \tag{3}$$

must hold for the scaling group borders, i.e., the points on the x-axis that fall on the borders of neighboring scaling groups. ρ_L denotes the width of the scaling group to the left, σ is the scaling factor value that belongs to the scaling group on the left, and σ_R denotes the scaling factor of the scaling group to the right.

Proof: Based on Lemmas 1 and 2 and the assumption of modules being infinitesimally small in resource demand, $\sigma(\rho)$ is a monotone increasing function and similarly to Equation 2,

$$d\sigma\rho_L \geq (\sigma_R - \sigma)\, d\rho \tag{4}$$

also holds, since $\sum_{i \in j-1} r_i$ translates to ρ_L and s_{j-1}^g, s_j^g are denoted as σ, σ_R, respectively. Therefore, the solutions to the given differential equation provide the possible scaling group borders in the proposed continuous model. Equation 3 is then derived from Equation 4 by rearranging the terms and fixing equality. ■

Lemma 3: The scaling cost decreases monotonically when the modules are grouped into more scaling units.

Proof: The statement holds since any group that consists of at least 2 modules with different scaling factors can be split into 2 groups that have a lower overall scaling cost. As the superfluous resource footprint of the individual scaling units gets smaller, in case of scaling them out, the amount of memory consumption scaled out unnecessarily is also smaller. ■

In contrast to the statement in Lemma 3, the communication costs increase monotonically with the number of scaling units due to the resource overhead of virtualization and to the higher number of inter-module invocation delays. These opposing effects call for an optimization exercise in order to find the sweet spot in operational costs of polylithic applications. However, hindered by the complexity of the model, for a joint optimization of all the listed costs, i.e., scaling and communication, we propose a 2-step heuristic approach: in the first step the minimal scaling cost is calculated for a set of scaling group numbers, in the second step the communication costs are calculated for the same set of scaling group numbers, then the overall cost is minimized by summing those respectively, and seeking the group number with the lowest total cost. In the next section we present such a calculation for an illustrative example.

The limitation of the model is that it ignores the call graph among the modules: it groups those modules together that are close in scaling factor, not necessarily those that frequently invoke each other, or whose lifetime overlaps the most.

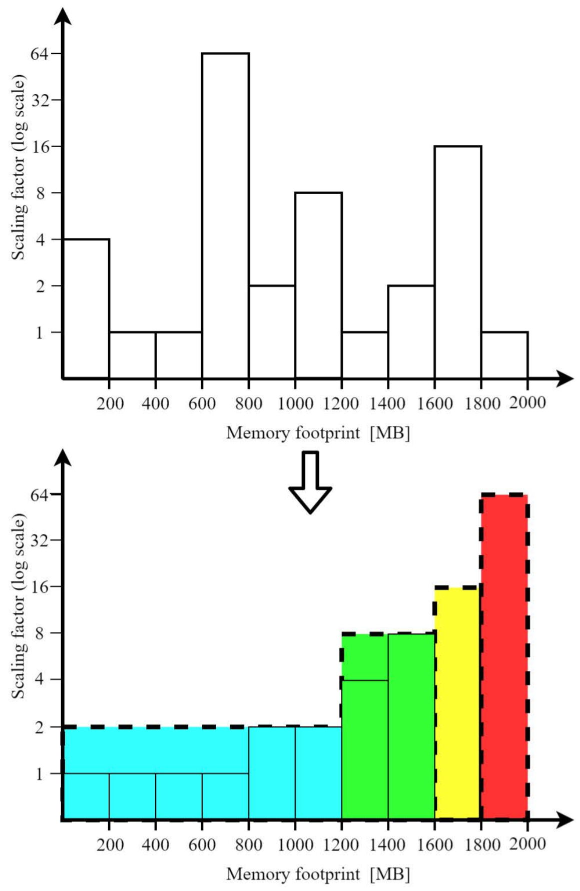

FIGURE 3. **Modules of an illustrative example application, ordered by their scaling factor, and grouped (denoted by various colors and dashed line contours).**

Therefore, workload affinity is not considered in consolidating software components into scaling units.

IV. RESULTS AND DISCUSSION: A COMPREHENSIVE COST ESTIMATION

We evaluate the cost model and the related optimization on typical polylithic applications recently collected from within the realms of Microsoft Azure. We take an illustrative case from the statistics reported in [21] and we optimize its packaging (scaling units) and deployment (CaaS vs. FaaS) depending on the scaling vs. communication cost interplay and the scale-out intensity of the operation. Also in this section, we apply the continuous approximation model proposed by Theorem 1 on a large-scale example inspired by reports on microservices in eBay and Alibaba clusters.

A. AN ILLUSTRATIVE EXAMPLE FROM MICROSOFT AZURE

Microsoft Research has recently published important statistics of Microsoft's cloud services [3]: VM allocation [22] and FaaS usage characteristics [21]. In the latter, the authors focus on the challenge of serverless platforms: the added latency due to cold starts. After a thorough analysis of usage data, the authors arrive at the conclusion that the resources the provider has to dedicate to each application are highly variable, and therefore the cost of keeping these applications warm, relative to their total execution (billable) time, can be prohibitively high, since the functions are very short lived compared to other cloud workloads, e.g., VMs. 50% of the functions run for less than 1 s on average, and 50% of the functions have maximum execution time shorter than 3 s; 90% of the functions take at most 60 s, and 96% of the functions take less than 60 s on average; while [22] shows that 63% of all VM allocations last longer than 15 minutes, and only $\approx$ 8% of the VMs last 5 minutes or less.

Building on the statistics published in [21], we provide a numerical evaluation of the model presented in Section III. We need the following types of data for our scaling vs. communication costs.

1) NUMBER OF FUNCTIONS PER APPLICATION

Reference [21] reports that half of the applications have a single function, i.e., monoliths. 5% and 0.04% of the applications have more than 10 and 100 functions, respectively.

2) DYNAMICS OF FUNCTION INVOCATIONS

The number of function invocations per day scatters over an 8 order of magnitude wide range. Half of the functions are invoked infrequently, i.e., once per hour or less; the fifth of them are invoked more than once per minute [21]. The interarrival times of invocations show an extremely high variation, i.e., a coefficient of variation higher than 5, for 20% of the functions, which means that the invocation rate is hectic for many functions [21].

3) MEMORY FOOTPRINT OF FUNCTIONS

The authors of [21] found that 90% of the functions consume less than 400MB of memory, and half of them consume less than 170MB.

We build an illustrative example on the measurement data set of [21] for showcasing our model's usability. First, we assume that all public cloud providers experience similar usage characteristics; second and more importantly, we suppose that modern and future applications will follow a similar design in terms of modularity and deployment. In the next cost calculations, we take an imaginary example application for which we draw the following attribute values from the empirical distributions of [21]. We make the case of an application consisting of 10 modules, each of them having the same memory footprint of $r_i = 200\text{MB}$ for $\forall i \in \{1, \dots, 10\}$. As per scaling dynamics, for simplicity, we distinguish off-peak hours and peak hours: during the former each module runs at a steady pace with a scaling factor of 1, during peak hours however we assume 4 modules at scaling factor 1, 2 modules at factor 2, 1 module at factor 4, 8, 16, and 64 each. With our model's notation: $s_1 = s_2 = s_3 = s_4 = 1, s_5 = s_6 = 2, s_7 = 4, s_8 = 8, s_9 = 16, s_{10} = 64$. The logarithmic steps in the modules' scaling factors are meant to reflect the high dynamics of invocations reported in [21].

B. OPTIMIZED MODULE FUSION FOR THE LOWEST COST

We evaluate the scaling and communication cost terms individually and in total. The scaling cost is a function of the memory allocated to the deployed software, as defined in Section III. Communication cost stands for the invocation delay. The communication rate is an inherent feature of deployment resources that strongly follows the actual interdependencies of software modules in the application. Whether modules that interact, hence produce high communication rate between each other, are packaged together has a large impact on the actual response time performance.

In order to make these cost terms comparable and summable, one must define the monetary value of the degradation in service quality caused by inter-module invocation latency. We simply use the added latency value in milliseconds for this purpose (to be compared with the memory footprint overhead in megabytes). We assume 10 ms added latency for the invocation of a scaling unit based on measurement studies [4], [23], [24]. We show the numerical results for 3 different call graph densities so that we try and cover a wide range of scenarios of inter-module communication. In the case when the call graph density is low, marked with Sparse in Table 1, the communication cost equals the average delay accessing a scaling unit multiplied by the square root of the number of scaling units. In the Medium and Dense cases in Table 1, however, we assume that the communication cost is equivalent to the access delay multiplied further with the square root of the number of scaling groups once and twice, respectively.

In Table 1 we organize the cost results by call graph density and fusion factor categories. The three labels in the latter cover the monolithic and fully polylithic scenarios under Low and High, respectively, while the Optimal label is assigned to the case in which the Total cost is minimized by selecting the most appropriate packaging of modules. The cost figures express the trade-off tackled in our model: the optimal setting is always between the two extremes, and as the delay becomes more critical, the more fusion takes place. The scaling groups (modules are depicted by their scaling factors, groups are defined by parenthesis) for Sparse, Medium and Dense call graph density cases are: (1,1,1,1)(2,2)(4)(8)(16)(64), (1,1,1,1,2,2)(4,8)(16)(64) as depicted in Figure 3, and (1,1,1,1,2,2)(4,8,16)(64). Hindered by the complexity of the model, the groups are determined by a brute-force search that consists of systematically enumerating all possible candidates for the solution and selecting the one with the lowest total cost. The enumeration is however shortlisted based on Lemmas 1 and 3.

The results in Table 1 show that when the fusion factor is high, then the scaling cost term becomes dominant, which is in line with the statement of Lemma 3. The optimal fusion factor counteracts this phenomenon, bringing down the scaling, and therefore the total cost into an acceptable range. On the other hand, the communication cost term decreases with the growth of the fusion factor. The optimal operation point depends on how we relate the two types of cost to each other: Table 1 depicts three specific scenarios in the rows of Sparse, Medium and Dense call graph density cases as mentioned above, e.g., in the Medium case with the optimal fusion factor the communication cost is 40, as there are 4 groups each imposing 10 ms of invocation delay. In the same scenario, the scaling cost is 8, because in the optimal grouping (1,1,1,1,2,2)(4,8)(16)(64) there is a 4×1 scaling overhead in the first group, and a 1×4 scaling overhead in the second group from the left.

TABLE 1. Cost terms (in abstract units) for different call graph density and module fusion scenarios.

Call graph density	Fusion factor	Cost terms		
		Scaling	Communication	Total
Sparse $\mathcal{O}(\sqrt{n})$	Low	0	32	32
	Optimal	0	25	25
	High	540	10	550
Medium $\mathcal{O}(n)$	Low	0	100	100
	Optimal	**8**	**40**	**48**
	High	540	10	550
Dense $\mathcal{O}(n\sqrt{n})$	Low	0	316	316
	Optimal	24	173	197
	High	540	10	550

C. LARGE SCALE EXAMPLE BASED ON eBay AND ALIBABA TRACES

In order to translate the presented cost values into a real-world example, let us take the microservice statistics of eBay, reported in [25], and that of Alibaba [19], [20]. The microservice system in eBay [25] includes around 3000 services (called as modules in this paper); these services work together to serve more than 10 business domains and form thousands of call paths. A regular application's call path may include up to 100 dependent services. The microservice system processes around 26 billion requests per day, i.e., 300 thousand requests per second. Luo *et al.* [19], [20] put more emphasis on the interconnection of the separate microservices, and presented an in-depth study of call graphs within the large-scale deployments of microservices at Alibaba clusters. Their main findings are that i) the size of microservice call graphs follows a heavy-tail distribution: around 10% of call graphs consist of more than 40 unique microservices (they have found that the largest call graph can even consist of 1500+ microservices), ii) a small percentage of microservices are hot-spots in call graphs, specifically, about 5% of microservices are multiplexed by more than 90% of online services and handle 95% of total invocations in Alibaba traces.

Based on the reported numbers, let us take a considerably large application, consisting of 100 microservices (the 95th percentile in the cumulative distribution of the number of microservices in each call graph reported in [20]). Furthermore, let us assume an exponential skewness in the invocation rate by combining the findings in [20] and [25]: a 400-fold difference between the most and least frequently called microservice, averaging to 100 calls per second. For simplicity, we assume a balanced memory footprint over the modules, each microservice instance taking 256MB memory. Following these assumptions, we approximate the scaling factor vs. memory footprint relationship by $\sigma(\rho) = a\exp(b\rho)$, such that $\frac{\sigma(100\times256)}{\sigma(0)} = 400$ and $\frac{\int_0^{100\times256}\sigma(\rho)\,d\rho}{100\times256} = 100$. Solving this simple system of

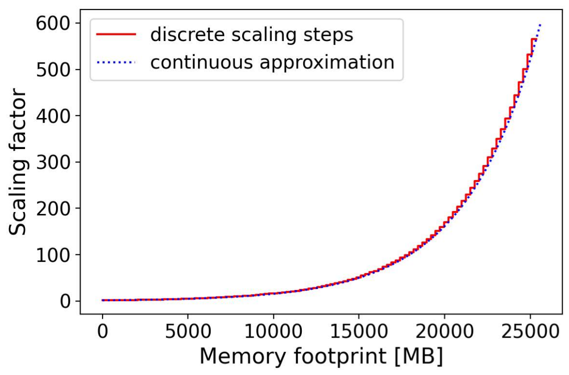

FIGURE 4. Scaling factor and memory footprint input to the large scale problem setup.

equations, the coefficients are $a = 1.5, b = 0.000234$. We depict discrete scaling factor values and the continuous approximation in Figure 4. In summary, we consider a microservices-based application, consisting of 100 components each taking 256MB of memory, and following an exponential curve in terms of scaling factor ranging from 1 to 600, averaging to 100.

As the next step we leverage Theorem 1, and solve the differential Equation 3 for various numbers of scaling groups. With the assumed exponential function, the equation that must stand for scaling group borders is

$$ab\exp(b\rho)\rho_L + a\exp(b\rho) - \sigma_R = 0. \tag{5}$$

In order to avoid an exhaustive search for the set of values of ρ_L and σ_R that satisfy the equation, we apply the following heuristic approach. We note, however, that other approaches might be applied as well, e.g., [26], [27], [28]. We iteratively look for the ρ values for which

$$\rho = \frac{\log(\sigma_R) - \log(ab\rho_L + a)}{b} \tag{6}$$

starting with σ_R being the largest scaling factor among the modules (e.g., the rightmost point on the curve of Figure 4) and ρ_L being the memory footprint (the whole range on the x-axis of Figure 4) divided by the number of groups we are striving to pinpoint. In the next round we re-set the value of σ_R to the resulting ρ, and we divide ρ with the decremented number of remaining scaling groups to get the new ρ_L. We repeat this step until the desired number of scaling groups are identified. The pseudo-code of the heuristics is shown in Algorithm 1.

The results are shown in Figure 5; the curves show the scaling (solid line on the left *y-axis*) and communication costs (dashed and dotted lines on the right *y-axis*) at various numbers of scaling groups found by Algorithm 1. The results validate Lemma 3, and show that with optimized packaging the achievable cost cut factor is significant in terms of scaling cost compared to a monolithic scenario. Contrarily, the rise of communication costs is observable as we move from a monolith towards a fully polylithic scenario in which software modules are all separately placed in the

Algorithm 1 Heuristic Algorithm for Finding Scaling Groups for Large Microservices-Based Applications

```
Require: $\sigma(\rho), n$    ▷ $n$: range of numbers of scaling groups
Ensure: $P \in R$ that satisfy $\frac{d\sigma}{d\rho}\rho_L + \sigma - \sigma_R = 0$
  for $i \in n$ do
      $R \leftarrow \varnothing$
      $\rho_L \leftarrow \frac{\rho_{\max}}{i}$
      $\sigma_R \leftarrow \sigma_{\max}$
      $j \leftarrow i$
      while $j > 1$ do
          $P \leftarrow \frac{\log(\sigma_R) - \log(ab\rho_L + a)}{b}$    ▷ assuming
  $\sigma(\rho) = a\exp(b\rho)$
          $R \leftarrow R \cup P$
          $j \leftarrow j - 1$
          $\rho_L \leftarrow \frac{\rho_{\max}}{j}$
          $\sigma_R \leftarrow \sigma(P)$
      end while
  end for
```

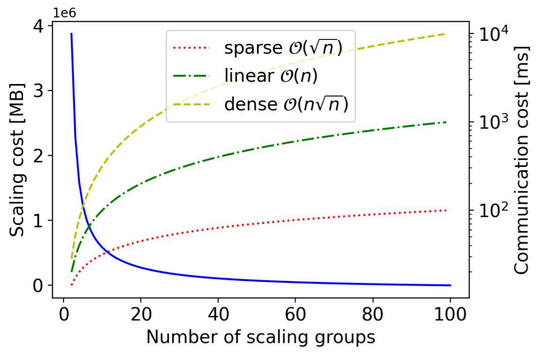

FIGURE 5. Scaling (solid line on the left *y-axis*) and communication costs (dashed and dotted lines on the right *y-axis*) in function of the number of scaling groups in the large scale problem setup. Communication costs are depicted for 3 different scenarios: sparse, medium (depicted as linear), and dense call graph densities determine the cost in function of the number of scaling groups.

cloud, potentially resulting in additional delays during service invocations. As [20] reports, end-to-end latency of online services increases linearly in the length of critical path, which is usually proportional to the number of microservices of the cloud-native application. The reason behind this phenomenon is that invocation between a pair of microservices is usually performed via HTTP REST API, RPC calls or Message Queues, and this can lead to a large communication overhead when many instances of these dependent microservices are located far away from each other. Indeed, various measurement studies [4], [23], [24] report non-negligible additional delays in end-to-end service response times due to invocation path of separately deployed virtualized components. In [20] the authors state that co-location of dependent microservices could improve response time performance by 22% on average. For demonstrative purposes we depict 3 different communication cost scenarios in Figure 5. The "sparse" scenario stands for the case in which the additional delay of inter-scaling group invocation increases with the square

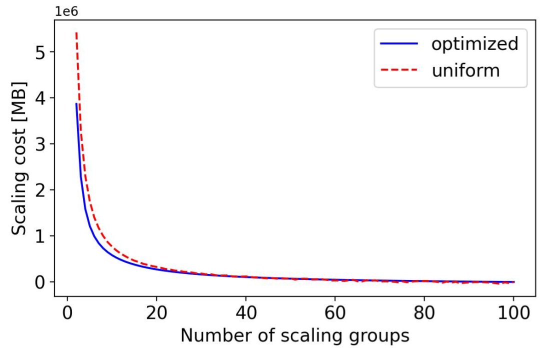

FIGURE 6. Scaling cost comparison between the optimized and the uniform grouping policies.

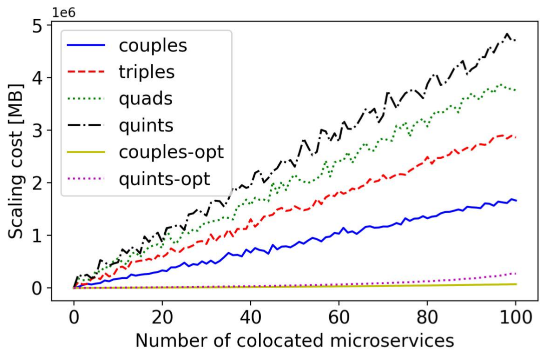

FIGURE 7. Scaling cost comparison between the co-locating policy and the optimized grouping.

root of the number of groups; in this case the added delay is not significant because calls are relatively infrequent. The "linear" shows the case in which this added delay is more prominent as such invocations happen frequently, i.e., as a linear function of the number of groups. Obviously in the latter case the overall minimum cost is achieved with fewer scaling groups. The third scenario, i.e., "dense", represents the case in which invocations represent almost a full mesh between scaling groups: in such a setup the overall added latency due to the invocations will grow super-linearly with the number of scaling groups, shifting the optimal number of scaling groups towards the low extreme. We calculated with 10 ms added delay per scaling group in each scenario.

In order to demonstrate how the scaling cost is affected by the chosen strategy, we show a comparison with uniform grouping: the modules are grouped in equally sized groups along the order of their scaling factors, e.g., if 5 groups are built, then 20 modules with the lowest scaling factors are grouped together, then the next 20 modules, and so on. In Figure 6 we depict the scaling costs for the optimal and uniform grouping policies in function of the number of groups. The results show that the "uniform" policy is only slightly overperformed by the optimal setup. This experiment demonstrates the message of Lemma 1, i.e., if the minimization of scaling cost drives the grouping, modules should be grouped together based on the order of their scaling factor. The optimized solution yields savings in terms of memory only in the case of a low number of scaling groups: the algorithm moves the scaling group borders away from the "uniform" cuts and the saved overhead is less likely to be high when there are many groups, i.e., when "uniform" cuts are dense.

The grouping strategy proposed in this paper focuses solely on the scaling cost, however one might consider a grouping strategy that builds the scaling groups with awareness to the call graph, further decreasing the communication costs. Several related work, e.g., [4], [19], [29], propose co-locating such microservices that frequently invoke one another in order to decrease the overall response time of the application. In a set of experiments, we co-locate couples, triples, quadruples and quintuples of microservices into scaling groups with no regard to their scaling factor, and look at the resulting costs in terms of scaling memory overhead. For comparison, we calculate the scaling cost yielded by our optimization algorithm. The results are depicted in Figure 7. On the *x-axis* we show the number of microservices that are assigned into groups, the remaining components are assumed to be deployed as singletons. We see an increasing scaling cost as the fraction of co-located microservices grow which is inline with Lemma 3. In fact the larger groups we create by co-locating components, the faster the memory overhead grows. Both phenomena are due to the fact that the scaling factors of the grouped microservices are not necessarily similar. For comparison, we depict the result of our proposed algorithm for the couples and quintuples cases, i.e., "couples-opt" and "quints-opt", respectively, in which the number and the sizes of scaling groups are the same as in the co-locating policy, but the grouping is performed along the scaling factor order of the microservices. The scaling cost is only a fraction of those of the co-locating setups. The reason for the significant difference is clear: while our proposed algorithm minimizes the scaling cost irrespective to its effects on the communications costs, the co-locating policy focuses only on the invocation graph, thus minimizing communications costs by grouping together modules with potentially diverse scaling factors. Then the more and larger groups of components with randomly picked scaling factors lead to increased scaling costs.

The drawback of co-locating microservices with similar scaling factors, and not those that are tightly connected in the call graph of the application, is visible in Figure 8. We take the couples and quintuples cases from the previous experiments, and calculate the communication cost by assuming medium density call graph among the groups after co-locating the microservices that are frequently invoking one another. As more microservices are grouped together and as larger those groups are, the communication cost is lower. The trend is exactly the opposite as with the scaling cost in Figure 7. However, when our algorithm sorts the microservices again into the same number of groups with the same sizes, those strongly coupled components might end up in separate scaling groups (assuming no correlation between the scaling factors and network positions in the call graph), making the call graph of groups dense. Therefore the communication costs resulted by our grouping algorithm are

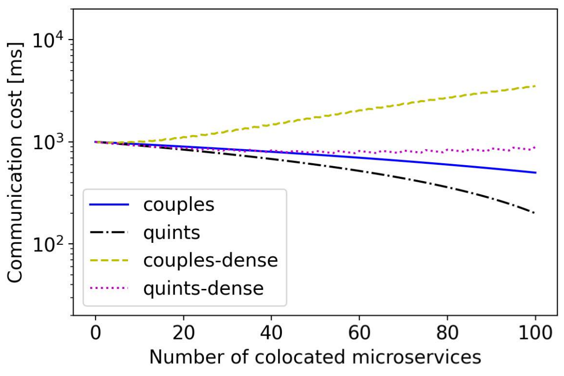

FIGURE 8. Communication cost comparison between the co-locating and the optimized grouping policies.

potentially higher, depicted by "couples-dense" and "quints-dense" in the chart.

One must ponder the relative importance of these cost terms, i.e., scaling and communication costs, in order to find the optimal arrangement of the microservices. As stated in Section IV-B, in order to make these cost terms comparable, one must define the monetary value of the degradation in service quality caused by inter-module invocation latency, as the price of memory allocation, i.e., the scaling cost, has already a definite price tag. Then the application designer may apply our proposed optimized grouping for the desired sweet spot in this trade-off dimension instead of option i) that considers co-locating microservice components frequently invoking each other for a low response time, or option ii) which is a fully polylithic scenario for the lowest scaling cost. In the next section we follow the cost conversion applied in Section IV-B and we calculate the operational costs of the illustrative example in various cloud deployments.

D. THE COST OF EXTENDED SCALE-OUT PERIODS

Finally, we put a real price tag on the deployment of the illustrative application in scale out episodes. We have collected the service fees of 3 major public cloud providers in both CaaS and FaaS. In Table 2 we calculate the hourly fee in those cloud services in the optimal fusion factor scenario under the medium call graph density regime (middle row of Table 1 in bold, as depicted in Figure 3). In the rightmost column we provide the price of a hybrid setup in which the grouped modules are being run in containers, i.e., CaaS, and the rest of the modules run in FaaS. As a reminder, each module has a 200MB memory footprint, and the number of instances is defined by the highest scaling factor in a group. So the grouping we consider is (1,1,1,1,2,2)(4,8)(16)(64) with scaling factors 2, 8, 16, and 64, respectively, as depicted in Figure 3.

We consider data center prices at the cheapest locations, and we take into account memory and, where it is applicable, processor fees. In order to keep the comparison fair, we do not consider edge computing scenarios with the involvement of accelerators including, but not limited to, GPUs and FPGAs. We consider the option of keeping functions

TABLE 2. Price (in USD) of running the example application in various public clouds for an hour in scale-out regime.

Provider	CaaS	FaaS	Hybrid
Amazon	4.14	9.72	8.04
Google	2.44	3.81	3.49
Microsoft	4.16	8.92	7.96

warm, a premium feature available at Amazon (called Provisioned Concurrency) and Microsoft (called Premium Plan). However, we omit invocation fees (negligible in this setting, less than 1% of total price assuming an invocation every 10s), free tiers (offered by Google), data traffic fees, and management fees of CaaS (charged by Google). In order to give a basis for the processor fee calculation, we assume that the code is continuously running for an hour, each module taking 1 vCPU. As Amazon does not allow to provision memory and CPU independently, we consider allocating the memory [30] that is necessary to reach 1 vCPU dedicated to it, i.e., 1800 MB instead of 200 MB for each module.

We show the total cost in Table 2 for each selected provider. Memory consumption is computed as the total memory footprint of all modules multiplied by their respective group's scaling factor. In contrast, CPU fee is calculated as the number of modules multiplied by their own scaling factor. Both memory and CPU unit prices have been collected from [1], [2], and [3].

Summarizing the figures of Table 2, we have 3 main observations. Comparing CaaS to FaaS, we can firmly state that deploying the application in CaaS is 2-fold cheaper, but it is widely known that reacting to hectic demand with scale out events is slower than doing the same with pre-warmed FaaS [30]. This aspect does not show in our analysis. Second, the FaaS offering of Amazon and Microsoft come with warm starts, hence the price difference compared to Google's service, which is cheaper but lacks the pre-warm feature. The cost is therefore expected to appear on the application QoS side when customers suffer from prolonged response times. Finally, the hybrid proposal, in which single modules (outliers regarding their scaling factors) are run in FaaS, while modules that are similar in scaling factor are packaged together in CaaS, yields a 10% saving compared to the FaaS-only scenario. The cost cut is due to efficient memory provisioning, and the hybrid solution does not compromise on fast scaling dynamics as low scaling factor modules are grouped for which scale-out rarely happens.

V. CONCLUSION

While the choice of cloud computing is unquestionable when it comes to deploying an application, as public cloud providers spoil the tenants with more and more service models, it has become a difficult question for application architects which service to use. The two major choices are CaaS and FaaS, the latter being originally tailored to running short-lived tasks serverless. We investigated this question from the perspective of the cost vs. latency trade-off, for which the stressful situations are scale out periods. We proposed an analytical model that incorporates the memory footprint during these episodes in case the application design does not allow for ample granularity of code scaling. On the other hand, the model also accounts for function invocation delays introduced by a highly distributed application design. Thus, we provided an analytical model for tackling the efficient scaling vs. communication cost trade-off, we evaluated several scenarios with an analysis built on real FaaS traces, and, finally, we proposed an optimal deployment setup for an illustrative example application with today's cloud pricing quotes. The analysis carried out on real CaaS and FaaS traces give some useful hints on how to distribute the application modules on cloud resources based on their scaling features and latency constraints. The proposed model can be useful for application designers that are building their cloud-based solutions for extremely tight delay constraints and hectic scaling events, e.g., telecommunications operators. The proposed approach can be fruitfully used in an actual design process once estimated or measured values for scaling factors and memory footprint have been obtained for the application under study in order to be able to take optimal decisions regarding its packing and deployment.

REFERENCES

[1] (2022). *Amazon Web Services (AWS): Elastic Container Service (ECS as CaaS) and Lambda (FaaS)*. Accessed: Sep. 11, 2022. [Online]. Available: https://aws.amazon.com/

[2] (2022). *Google Cloud: Google Kubernetes Engine (GKE as Caas) and Google Cloud Functions (FaaS)*. Accessed: Sep. 11, 2022. [Online]. Available: https://cloud.google.com/

[3] (2022). *Microsoft Azure: Azure Kubernetes Service (AKS as CaaS) and Azure Functions (FaaS)*. Accessed: Sep. 11, 2022. [Online]. Available: https://azure.microsoft.com/

[4] D. Haja, Z. R. Turanyi, and L. Toka, "Location, proximity, affinity—The key factors in FaaS," *Infocommun. J.*, vol. 12, no. 4, pp. 14–21, 2020.

[5] M. Alam, J. Rufino, J. Ferreira, S. H. Ahmed, N. Shah, and Y. Chen, "Orchestration of microservices for IoT using Docker and edge computing," *IEEE Commun. Mag.*, vol. 56, no. 9, pp. 118–123, Sep. 2018.

[6] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly, "Machine learning-based scaling management for kubernetes edge clusters," *IEEE Trans. Netw. Serv. Manag.*, vol. 18, no. 1, pp. 958–972, Mar. 2021.

[7] S. Sharma, R. Miller, and A. Francini, "A cloud-native approach to 5G network slicing," *IEEE Commun. Mag.*, vol. 55, no. 8, pp. 120–127, Aug. 2017.

[8] I. Pelle, J. Czentye, J. Dóka, A. Kern, B. P. Gero, and B. Sonkoly, "Operating latency sensitive applications on public serverless edge cloud platforms," *IEEE Internet Things J.*, vol. 8, no. 10, pp. 7954–7972, May 2021.

[9] J. Soares, C. Gonçalves, B. Parreira, P. Tavares, J. Carapinha, J. P. Barraca, R. L. Aguiar, and S. Sargento, "Toward a Telco cloud environment for service functions," *IEEE Commun. Mag.*, vol. 53, no. 2, pp. 98–106, Feb. 2015.

[10] M. Szalay, M. Nagy, D. Géhberger, Z. Kiss, P. Mátray, F. Németh, G. Pongrácz, G. Rétvári, and L. Toka, "Industrial-scale stateless network functions," in *Proc. IEEE Int. Conf. Cloud Comput. (CLOUD)*, Jul. 2019, pp. 383–390.

[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] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2020, pp. 419–433.

[13] G. Mazlami, J. Cito, and P. Leitner, "Extraction of microservices from monolithic software architectures," in *Proc. IEEE Int. Conf. Web Services (ICWS)*, Jun. 2017, pp. 524–531.

[14] R. Chen, S. Li, and Z. Li, "From monolith to microservices: A dataflow-driven approach," in *Proc. 24th Asia–Pacific Softw. Eng. Conf. (APSEC)*, 2017, pp. 466–475.

[15] I. J. Munezero, D.-T. Mukasa, B. Kanagwa, and J. Balikuddembe, "Partitioning microservices: A domain engineering approach," in *Proc. IEEE/ACM Symp. Softw. Eng. Afr. (SEiA)*, May/Jun. 2018, pp. 43–49.

[16] P. Di Francesco, P. Lago, and I. Malavolta, "Migrating towards microservice architectures: An industrial survey," in *Proc. IEEE Int. Conf. Softw. Archit. (ICSA)*, Apr./May 2018, pp. 2909–2929.

[17] M. Daoud, A. E. Mezouari, N. Faci, D. Benslimane, Z. Maamar, and A. E. Fazziki, "Towards an automatic identification of microservices from business processes," in *Proc. IEEE 29th Int. Conf. Enabling Technol., Infrastruct. Collaborative Enterprises (WETICE)*, Sep. 2020, pp. 42–47.

[18] F. Ponce, G. Márquez, and H. Astudillo, "Migrating from monolithic architecture to microservices: A rapid review," in *Proc. 38th Int. Conf. Chilean Comput. Sci. Soc. (SCCC)*, 2019, pp. 1–7.

[19] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proc. ACM Symp. Cloud Comput. (SoCC)*, 2021, pp. 412–426.

[20] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, J. He, and C.-Z. Xu, "An in-depth study of microservice call graph and runtime performance," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 12, pp. 3901–3914, Dec. 2022.

[21] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2020, pp. 205–218.

[22] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms," in *Proc. ACM 26th Symp. Oper. Syst. Princ. (SOSP)*, 2017, pp. 153–167.

[23] J. Li, S. G. Kulkarni, K. K. Ramakrishnan, and D. Li, "Understanding open source serverless platforms: Design considerations and performance," in *Proc. ACM 5th Int. Workshop Serverless Comput. (WOSC)*, 2019, pp. 37–42.

[24] M. Ganguli, S. Ranganath, S. Ravisundar, A. Layek, D. Ilangovan, and E. Verplanke, "Challenges and opportunities in performance benchmarking of service mesh for the edge," in *Proc. IEEE Int. Conf. Edge Comput. (EDGE)*, Sep. 2021, pp. 78–85.

[25] X. Guo, X. Peng, H. Wang, W. Li, H. Jiang, D. Ding, T. Xie, and L. Su, "Graph-based trace analysis for microservice architecture understanding and problem diagnosis," in *Proc. ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, 2020, pp. 1387–1397.

[26] O. A. Arqub and Z. Abo-Hammour, "Numerical solution of systems of second-order boundary value problems using continuous genetic algorithm," *Inf. Sci.*, vol. 279, pp. 396–415, Sep. 2014.

[27] Z. Abo-Hammour, P.-O. Alsmadi, S. Momani, and O. A. Arqub, "A genetic algorithm approach for prediction of linear dynamical systems," *Math. Problems Eng.*, vol. 2013, no. 1, 2013, Art. no. 831657.

[28] Z. Abo-Hammour, O. Abu Arqub, S. Momani, and N. Shawagfeh, "Optimization solution of Troesch's and Bratu's problems of ordinary type using novel continuous genetic algorithm," *Discrete Dyn. Nature Soc.*, vol. 2014, no. 2, 2014, Art. no. 401696.

[29] A. Saboor, A. K. Mahmood, M. F. Hassan, S. N. M. Shah, F. Hassan, and M. A. Siddiqui, "Design pattern based distribution of microservices in cloud computing environment," in *Proc. Int. Conf. Comput. Inf. Sci. (ICCOINS)*, 2021, pp. 396–400.

[30] 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.

LASZLO TOKA (Member, IEEE) received the Ph.D. degree from Telecom ParisTech, in 2011. He worked with Ericsson Research, from 2011 to 2014, then he joined the academia with research focus on software-defined networking, cloud computing, and artificial intelligence. He is currently an Associate Professor with the Budapest University of Technology and Economics, the Vice-Head of the HSN Laboratory, and a member of both the MTA-BME Network Softwarization and the MTA-BME Information Systems Research Groups.