MicroBlend: An Automated Service-Blending Framework for Microservice-Based Cloud Applications

Myungjun Son*, Shruti Mohanty*, Jashwant Raj Gunasekaran†, Mahmut Kandemir*
* The Pennsylvania State University, USA, † Adobe Research, USA
{mjson, sxm1743, mtk2}@psu.edu, jgunasekaran@adobe.com

***Abstract*—With the increased usage of public clouds for hosting applications, it becomes essential to choose the appropriate services from the public cloud offerings in order to achieve satisfactory performance while minimizing deployment expenses. Prior research has demonstrated that combining different services can be more cost-effective than solutions based on a single service type. However, automating the combination of resources for applications composed of large graphs of loosely-connected microservices has not yet been thoroughly explored, especially in the context of microservice-based cloud applications. Motivated by this, targeting microservice-based applications, we propose MicroBlend, an automated framework that mixes Infrastructure-as-a-Service (IaaS) and Function-as-a-Service (FaaS) cloud services in a way that is both cost-effective and performance-efficient. MicroBlend focuses on: (i) providing an automated approach for blending resources that takes microservice dependencies into account, (ii) generating FaaS-ready code using a compiler-based approach, and (iii) suggesting an optimization plan for combining microservices with user annotation. We implement MicroBlend on Amazon Web Services (AWS) and evaluate its performance using real-world traces from three different applications. Our findings demonstrate that by employing automated microservice-to-cloud service assignment, MicroBlend can significantly reduce Service Level Objective (SLO) violations by 9%, compared to traditional VM-based resource procurement schemes. Additionally, MicroBlend can decrease costs by 11%.**

***Index Terms*—automation, compiler, serverless, microservices, cloud computing, autoscaling.**

I. Introduction

A variety of tenants are utilizing the public cloud to host their applications, many of which have performance constraints (e.g., latency or throughput), known as service-level objectives (SLOs). The choice of cloud services acquired to host these applications has a crucial impact on satisfying such SLOs and determining costs, which can be termed as the *performance-cost problem.* In particular, one may want to find the minimum-cost solution that satisfies the specified SLO.

Public cloud resources are traditionally acquired through Infrastructure as a Service (IaaS), including virtual machines (also known as instances), containers, block storage devices, etc. Function as a Service (FaaS) and Software as a Service (SaaS) offerings have increasingly become more accessible. FaaS and SaaS are often favored over Infrastructure as a Service (IaaS) due to their finer-grained cost model and reduced administrative expenses.

Prior studies show that no single service type can satisfy all application requirements, and combining multiple service types can potentially help us better address the performance-cost problem [3], [46]. In particular, integrating Infrastructure-as-a-Service (IaaS) with Function-as-a-Service (FaaS) has been found to have numerous advantages over using IaaS alone. FaaS acts as an agile transition mechanism, enabling the creation of new virtual machines (VMs) in significantly less time than traditional VMs take to start, reducing the need for over-provisioning VMs and addressing performance-cost issues. Additionally, Splice [48] has demonstrated how the process of merging IaaS and FaaS can be automated via simple pragma type annotations in the source code.

Despite the automated method of leveraging diverse services, detailed performance analysis and modeling for service blending have been studied in detail. One specific area of interest that could benefit from service blending is microservices [1], [2], [14], [38], [39]. In particular, given a microservices architecture diagram, identifying the set of microservices to implement as FaaS, with performance and cost in mind, is an interesting research direction. This is a challenging problem in general due to the back-pressure and cascading SLO violations caused by dependencies across microservices. Automating this microservice-to-cloud service assignment can be of tremendous value to many microservice-based cloud applications.

Motivated by these observations, in this paper, we present **MicroBlend**, an automated framework for blending IaaS and FaaS services in a way that is both cost-effective and performance-aware for microservice-based workloads. The key features of MicroBlend include: (i) an automated strategy for blending cloud services targeting microservice-based applications considering microservice dependencies; (ii) FaaS-ready code generation using a compiler-based approach; and (iii) an optimization plan for merging microservices with user annotation. The main **contributions** of this paper can be summarized as follows:

- We introduce MicroBlend, which allows for the automated partitioning of microservices into IaaS and FaaS during cluster autoscaling.
- We demonstrate a heuristic service assignment algorithm that uses a performance model and a simple learning framework to identify optimal combinations of microservices for offloading to FaaS.
- We present an optimization plan that *merges* microser-

vices, if it is beneficial to do so, before offloading them to FaaS with the help of user annotations.

• We implement MicroBlend using various cloud computing services provided by AWS and the abstract syntax tree module. We evaluate the effectiveness of MicroBlend using real-world request arrival traces (WITS and Wikipedia) and three microservice benchmark suites (DeathStarBench [22] – SocialNetwork, MediaReservation, and Online Boutique [41]).

• We demonstrate that MicroBlend can reduce SLO violations by up to 9%, compared to using only VMs, and can save up to 11% on costs, compared to standard resource procurement methods.

II. Background and Motivational Experiment

A. Background

From the standpoint of application development, it has been proposed that cloud applications be designed utilizing a microservice-based architecture [37], [42]. Microservice-based architecture treats applications as a collection of loosely-coupled, lightweight, and modular software components and they are highly suited to the cloud paradigm because of their small size, modularity, agility, and flexibility [43], [53].

Fig. 1 provides a sample microservice architecture using the Social Network application. This particular architecture consists of multiple microservices that are containerized applications and communicate with each other through HTTP API or RPC calls. A user request is received at the front-end layer of the application and, depending on the type of request, a subset of microservices from the front-end tier, middle-tier (logic), and back-end tiers (storage) are involved in serving the request. The microservices are arranged in a directed acyclic graph (DAG) that represents their dependencies with respect to one another, with some microservices relying on others. This dependency graph highlights the complexity of microservice-based applications when compared to monolithic ones. The main objective of MicroBlend is to *automate* the selection and partitioning of microservices by identifying their hierarchical arrangement in a complex network. More specifically, MicroBlend automatically assigns each microservice in a DAG to either IaaS or FaaS.

Developers deploy each microservice on one or more containers while creating a microservice application. Kubernetes [30] is representative of a platform for orchestrating container deployment, scalability, and administration. A Kubernetes cluster consists of a collection of nodes that contain pods, and a pod is a shared storage and network environment that contains one or more containerized applications (usually one). Additionally, each pod has CPU and memory limitations that restrict how much of the node's resources it may use.

Kubernetes supports three application autoscaling mechanisms: *horizontal autoscaling*, *vertical autoscaling*, and *cluster autoscaling*. Horizontal autoscaling adjusts pod numbers in response to metric changes such as CPU or memory utilization. Vertical autoscaling modifies the CPU or memory constraints of pods, controlling minimum and maximum resource allocation. Finally, cluster autoscaling adds or removes nodes

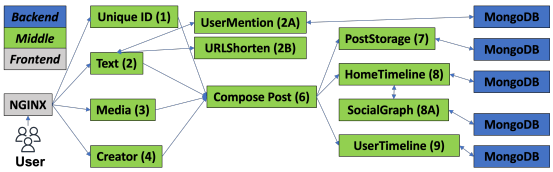


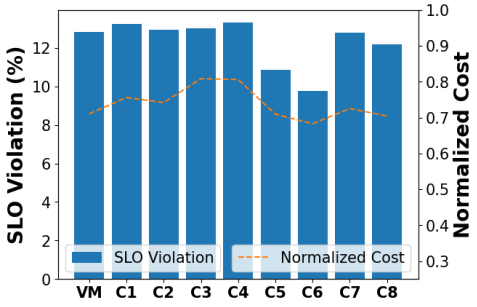
Fig. 1: Social Network as example of microservices architecture with numbered microservices for motivation experiment.

from a cluster. Note that cluster autoscaling is provided by the underlying infrastructure/cloud provider, such as Amazon Elastic Kubernetes Service (EKS) [5], Microsoft's Azure Kubernetes Service (AKS) [9], and Google Kubernetes Engine (GKE) [25], whereas horizontal and vertical autoscaling are provided by the Kubernetes system. Cluster autoscaling in the cloud modifies the size of the cluster and the number of resources leased from a cloud provider, which directly affects the cluster's cost. In this work, we tackle the challenges of cluster autoscaling in the cloud.

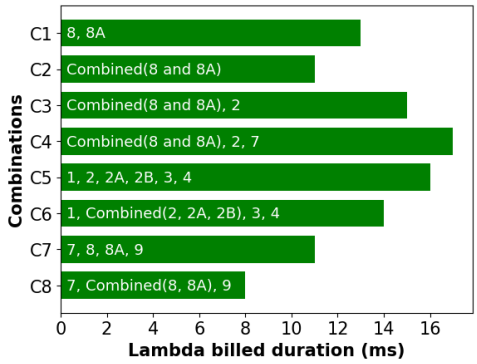
When spinning up new VMs (cluster autoscaling) in case of a surge in requests, VMs may take tens of seconds to minutes to start up, resulting in SLO violations and over-provisioning of VMs. Combining IaaS with FaaS has been found to reduce operational and runtime expenses, and AWS Lambdas have been used in a recent study on service blending. When scaling up IaaS resources and addressing traffic spikes, prior works [26], [29], [48], [52] generally employ FaaS as a "transition mechanism". However, to our knowledge, automated transition for microservice-based applications has not been explored.

B. Motivational Experiment

To examine the potential advantages of cloud service blending for microservices over the use of a single service type during cluster autoscaling, we have performed an experiment involving a microservice-based Social Network application [22]. Fig. 1 depicts the dependency graph of this application that comprises an Nginx front-end microservice, 11 functionality-based middle-tier microservices, and 5 MongoDB back-end microservices. We identify 11 functionality-based microservices that can be blended (using IaaS or FaaS for each microservice). These microservices are numbered in the diagram, where some of them are labeled with either "A" or "B" indicating that they are only called by the microservice labeled with the corresponding number; the "2" microservice only invokes microservices "2A" and "2B". We configure Lambdas to interact with one of MongoDB microservices in a distributed fashion. We use the WITS [50] trace to simulate the scenario where the web service handles different queries every second. We determine the number of initial servers required to maintain the average requests per second for the WITS trace by considering the number of requests each vCPU can handle simultaneously with a response time objective of 1000 ms. We assess the SLO violation and cost reduction by testing various combinations of blended resources during cluster autoscaling using the initial 30 minutes of the WITS trace. In particular, combination (1, 3) specifies that



(a) SLO (1000ms) violations and cost reductions for Social Network application under different combinations (more details about combinations available in Fig. 2b).



(b) Description of the microservice combinations presented in Fig. 2a with the billed duration on Lambda. The corresponding microservice numbers can be identified from Fig. 1.

Fig. 2: SLO violations and cost reduction under different microservice combinations.

the UniqueId (1) and Media (3) microservices are mapped to Lambda, whereas the remaining microservices are executed on the VM during cluster autoscaling.

Fig. 2a shows the comparison between SLO violation and cost savings of the Social Network application service, considering different service blending combinations. The horizontal axis represents different sets of microservices that are offloaded to Lambda – while (*VM*) denotes the first baseline where all microservices are executed on VM. The second y-axis indicates the monetary cost normalized to the second baseline, which represents a resource procurement strategy that solely relies on Lambda. Fig. 2b describes the specific microservice combination and its billed duration on Lambda.

The combination C1 (HomeTimeline and SocialGraph) involves offloading the microservice with the longest execution time, which is HomeTimelineService (8). It should be noted that, HomeTimelineService (8) calls SocialGraphService (8A), which involves having two separate Lambda functions. The social-network application consists of parallel microservices that execute in stages, and the latency of a stage depends on the microservice with the longest execution time. Even though executing the HomeTimelineService and SocialGraph microservices on Lambda is independent of the VM load surge, the other two microservices (7 and 9) that are involved in parallel invocations with HomeTimelineService (8) face performance degradation in the surge of requests, resulting in similar SLO violation and cost increases (1% and 6%, respectively). Note that the cost increase is due to the higher per-unit price of Lambda compared to VMs. Moreover, even when the subsequent microservices with the longest execution time are offloaded to Lambda (C3 and C4 combinations), the workload patterns and linear cost increase of Lambda, together, negatively impact the cost reduction without any change in SLO violation reduction.

In order to address the impact of blending resources on microservices with parallel invocation patterns, we assessed the impact of offloading these microservices to Lambda using two combinations: C5 and C7. These combinations have reduced SLO violations by 16% and 15%, respectively, when compared to the VM policy. This is due to the fact that Lambda can handle simultaneous invocations, which reduces the resource burden on VMs and allows unused VMs to be scaled-in early, resulting in cost savings of 1% and 2% for C5 and C7, respectively.

To further investigate the impact of blending resources while considering the dependencies of microservices, we have selected closely related microservice candidates. For instance, we have evaluated the combination of HomeTimeline and SocialGraph microservices as a single Lambda function (C2 combination) instead of having them on separate Lambdas (C1 combination). We emphasize that, grouping microservices have allowed microservices to communicate within a Lambda, instead of between two separate Lambdas, resulting in a shorter execution time. Similarly, combining the microservices involved in cascading functions into one (C6 and C8) instead of separate functions (C5 and C7) resulted in optimal resource utilization and reduced communication overhead, leading to a 5% reduction in SLO violation for C6 and C8, compared to C5 and C7, respectively.

Based on our findings, we propose that the automated identification of microservice dependencies and appropriate blending of services can lead to better resource utilization and reduced SLO violations in practice without much programmer involvement (i.e., increasing programmer's productivity).

III. MicroBlend Design

We begin by providing a general overview of MicroBlend's structure and components. Next, we provide a comprehensive guide on building a performance model for each microservice to determine their suitability for FaaS. We illustrate how MicroBlend uses a heuristic algorithm to produce FaaS executable code for microservices and an optimization plan that utilizes a compiler-based approach with user annotation.

A. Proposed Design

MicroBlend is a framework that can automate the integration of different cloud services to reduce the amount of human labor required for microservice applications. The high-level design specifications of MicroBlend are shown in Fig 3. MicroBlend has a "pragma-based" annotation system that the programmer may utilize to annotate the application

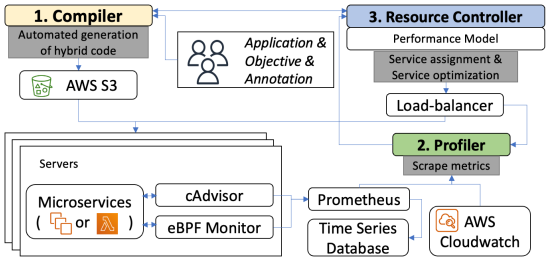


Fig. 3: High-level design of MicroBlend.

code. The programmer's purpose would be to impart domain knowledge about preferred so-called "basic execution units" (BEUs), an execution granularity (functionality in microservices), for blending selections that we would construct. MicroBlend employs annotations to specify specific BEU-to-service type translations. Annotated code is analyzed by MicroBlend, which creates an "intermediate representation" (IR) that our proposed compiler uses to make "cloud-ready" code automatically. The user defines the workload, service-level objectives (SLOs), and user annotations. If no annotations are provided, MicroBlend employs an empirical usage approach, utilizing performance model and a heuristic service assignment algorithm, which includes a simple learning framework (logistic regression), to determine the optimal microservice-to-cloud service assignment.

We begin with the initial servers linked to the load balancer that is responsible for (i) acquiring resources and (ii) cloud service assignment. Whenever the load balancer decides to scale up, it triggers the service assignment component of MicroBlend. MicroBlend interacts with its different components and instructs the load balancer to utilize blending services when required.

B. Component Details

1) ***Compiler****:* The Compiler is responsible for translating the source code into compatible executables for both Infrastructure-as-a-Service (IaaS) and Function-as-a-Service (FaaS). The process starts by searching for user-annotated pragmas on microservices, which are then used to transmit metrics to the Resource Controller. The Compiler automatically enables program analysis and the creation of cloud-ready code. Once the cloud-ready blended code is generated, the Resource Controller collects information and directs the Compiler to upload the executable code to all servers through remote storage, such as AWS S3 [8]. In cases where user annotation is absent, the Compiler interacts with the Resource Controller and generates blended code during autoscaling whenever blending resources becomes necessary.

2) ***Profiler****:* The Profiler is in charge of microservice observability. It periodically observes the Load-balancer, container services, and AWS CloudWatch [6][1]. It retrieves data from CloudWatch at regular intervals, including CPU and memory usage of running servers. Furthermore,

[1]AWS CloudWatch allows users to monitor and observe their applications and infrastructure resources on the AWS Cloud. It provides data and actionable insights by enabling users to track metrics, monitor log files, and set alarms.

Pragma Type	Description
BEU FaaS	Placement on a Lambda.
BEU FaaS [arrival_rate >R]	Placement on a Lambda with load balancing rule.
BEU FaaS Combine L1	Grouping microservices to the same Lambda.

TABLE I: Pragmas supported in MicroBlend.

Prometheus [44] is employed to gather various metrics from containers (further details in Sec. IV).

3) ***Resource Controller****:* The Resource Controller is responsible for autoscaling and cloud service assignment for microservices. It gathers data from the Compiler's user annotations, if present, and also uses the current request arrival rate, server metrics, and container metrics. It then determines the number of additional VM instances required for scaling out resources. To scale in resources, the Resource Controller examines VMs that have been idle for more than three minutes and terminates them to minimize the early termination of instances caused by short-term request rate fluctuations, as recommended by Gandhi et al [24].

The Resource Controller has several options based on the type of user annotation when it comes to scaling-out resources. These options include increasing VM resources through the standard auto-scaling policy, managing incoming requests with blending services while creating more VMs, or solely using blending services when user-defined metrics are met. For instance, it can operate in a FaaS-only mode when the request rate surpasses a particular value. When the specified metric is met, the Resource Controller notifies the Compiler to deploy the blended cloud-compatible code to all active servers.

In the absence of user annotation, the Resource Controller of MicroBlend creates a performance model based on the Profiler's metrics and uses a heuristic algorithm to determine the appropriate allocation of services to microservices. The Controller then instructs the Compiler to generate a cloud-ready, service-blended code, and the load balancer directs incoming requests to leverage the blended code to fulfill demand while obtaining more resources.

C. User Annotation Schema

In this work, the term "Basic Execution Unit" (BEU) refers to a code fragment for which MicroBlend decides whether to use serverless function or VM. Specifically, the code fragments delimited by programmer pragmas define a unit of functionality and are potential candidates for serverless execution, with related functions annotated by explicit pragmas considered as Lambda functions by developers; otherwise, the Compiler interprets the logic unit as a VM.

Our pragmas differentiate between single-function and multi-function specifications, as shown in Table I. Specifically, pragma BEU FaaS signifies that the function is suitable for inclusion in a Lambda function. When developers specify the pragma BEU FaaS [arrival rate $>$ R], they provide domain knowledge to the load balancer's service assignment rule, which MicroBlend should consider, and the cloud-ready code generated by the Compiler should contain a load-balancing rule that instructs requests above the user-defined threshold to

be routed to Lambda-based function invocations. Comparison of R to the current arrival rate is carried out by MicroBlend's Resource Controller. We want to emphasize that the rule mentioned above ([arrival rate $>$ R]) is just an example and MicroBlend can handle arbitrarily complex rules as well. On the other hand, the BEU FaaS Combine L1 pragma directs the Compiler to locate and combine functions with the "L1" prefix, which offers more flexibility to microservice-based systems. Combining many BEUs into a single execution unit (FaaS) may be the most cost-effective approach, especially if they form a "cascading workflow pattern" with high communication latency, as it reduces communication overheads and enables FaaS to complete within its allowed duration. We further discuss using annotations for grouping functions in service optimization in Sec. III-F.

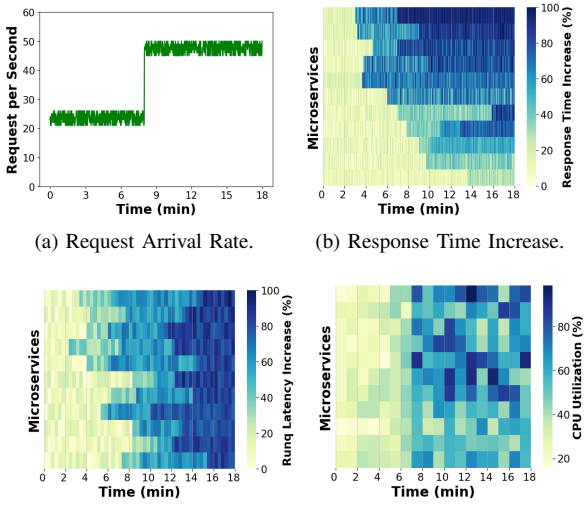
D. Performance Model

As stated in Sec. II-B, it is crucial to determine appropriate cloud service assignments for microservices, taking into account the dependencies and workflow patterns of microservice-based applications, to minimize SLO violations and maximize cost savings. To address these concerns, we have developed a performance model for each microservice, which is used to identify microservices suitable for FaaS.

Performance Model: Specifically, previous research has shown that CPU utilization is not the most effective metric for resource allocation [10], [16], [20]. As an alternative to CPU utilization, MicroBlend employs "extended Berkeley Packet Filtering" (eBPF), which is a safe and low-overhead tool for gathering precise metrics from kernel-level events [18]. Specifically, eBPF measures memory resource allocations, CPU scheduling decisions, and networking stack packet events. MicroBlend uses the eBPF Linux scheduler runq latency (runq latency) metric to determine the performance of each microservice. runq latency [19] is shown as a histogram depicting the latencies experienced by threads. runq latency has been extensively utilized for microservice observability for various goals, including performance improvements, profiling and tracing, and security [4], [13], [31]. Note that, runq latency is preferred over CPU utilization because it shows how application threads compete for CPU resources and the requirement for additional (or fewer) CPU resources.

Efficiency of runq latency metric: We conducted an experiment by deploying the Social Network application to investigate the correlation between the runq latency metric and the response time of different middle-tier microservices. In this experiment, we started with an average request rate of 24 requests per second and then doubled it to observe how the various microservices respond to changes.

The heatmap in Fig. 4b reveals that the middle-tier microservices, which interact with the back-end microservices, have experienced an increased response time delay relative to other microservices. It can be observed that as the number of requests increases during the experiment, the high response time spreads from middle-tier microservices that communicate with back-end microservices to those situated near the front-end microservice. The runq latency depicted in Fig. 4c also behaves similarly to the response time, gradually growing and propagating from middle-tier microservices situated near the back-end microservices and those near the front-end microservice. For each microservice, a high runq latency is *strongly correlated* with a high response time. However, as shown in Fig. 4d, the CPU utilization of most of the microservices increases as the number of requests increases. In contrast, the response times of middle-tier microservices located near the front-end microservice are not consistently affected by an increase in CPU utilization. While all microservices utilize a significant amount of the CPU, the response time of microservices situated near the front-end microservice remained constant. This is due to the fact that CPU utilization metric treats each microservice independently and does not account for their interdependence. To ensure precise monitoring, we have chosen the *runq latency* metric as the performance model for each microservice based on our experimentation.



(a) Request Arrival Rate. (b) Response Time Increase.

(c) Runq Latency Increase. (d) CPU Utilization.

Fig. 4: Heatmap depicting value increase as the request arrival rate doubles with different metrics. The middle-tier microservices are arranged on the primary y-axis, with those closely associated with back-end microservices at the top and those closely associated with front-end microservice at the bottom.

E. Service Assignment

Heuristic Service Assignment: To enable the identification and conversion of suitable microservices into FaaS for use during cluster autoscaling, we introduce a heuristic service assignment algorithm. The proposed algorithm is designed to allocate microservices to *either IaaS or FaaS*, taking into account the microservice performance and dependencies. The algorithm utilizes performance models to identify the most suitable resource combinations based on key metrics, such as runq latency, and a logistic regression-based learning framework. Algorithm 1 outlines the steps for heuristic service assignment. The algorithm takes a set of microservices and a target SLO as input and outputs a set of microservices that should be run on FaaS to meet the target SLO.

Algorithm 1: Heuristic Service Assignment

Input: M – Microservices;
S – SLO Target;
Output: Allocated microservices to *IaaS* or *FaaS*
P – Prediction Model;
while *True* **do**
  $L \leftarrow \emptyset$;
  $R \leftarrow 0$;
  for *each microservice* $m \in M$ **do**
    $runq\_latency \leftarrow$ runq_sample_histogram(m);
    $observed_m \leftarrow$ P95$(runq_latency)$;
    $L \leftarrow L \cup \{(m, observed_m)\}$;
    $R \leftarrow R + observed_m$;
  Sort L by $observed_m$ in descending order;
  $D \leftarrow \emptyset$;
  for *each* $m \in L$ **do**
    $D \leftarrow D \cup \{m\}$;
    Subtract $observed_m$ from the R;
    if R *reaches below the predicted runq latency based on* P **then**
      Break out of the loop;
  Assign D to FaaS;

To collect performance data on microservices, the algorithm obtains 60 instances of the runq latency metric histogram over the previous 60 seconds. From each histogram, the 95th percentile is selected, and the observed latency is obtained by calculating the average of the 60 data points. The microservices are then sorted in descending order based on their observed latency. The algorithm then employs a "Prediction Model", discussed in detail in Sec. III-E, which predicts the anticipated runq latency for the current request rate, aiming to achieve a response time that satisfies the SLO target, i.e., a response time that is less than or equal to the specified SLO.

Based on the *predicted* runq latency, the algorithm initially selects the microservice with the highest runq latency value and subtracts the corresponding runq latency from the sum of observed runq latency per microservice to determine if it reaches the predicted runq latency value. If it does not, the algorithm moves on to the next highest value and repeats the process until the total runq latency value falls below the predicted runq latency. Once this value is reached, the algorithm selects the set of microservices that were previously chosen and assigns them to Lambda. Note that, if the predicted runq latency is lower than the observed total runq latency, we do not assign microservices to FaaS as, in that case, there would be no benefit in reducing SLO violation (at the expense of additional costs associated with Lambda execution), as described in Sec. II-B. We want to emphasize that, by considering *both* the performance of microservices as well as their dependencies by utilizing runq latency, our heuristic service assignment enables offloading appropriate microservices to Lambda and reduces the CPU burden on VMs, and this eventually leads to better performance and a drop in SLO violations.

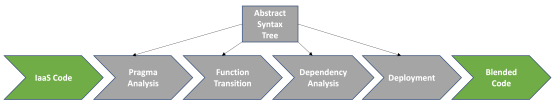


Fig. 5: MicroBlend compiler pipeline.

Learning Framework: MicroBlend utilizes logistic regression [27] – which is a statistical method that models the relationship between a binary outcome variable and predictor variables – to create a prediction model for microservice performance. Our model analyzes the relationship between the outcome variable, which is the runq latency metric in this case, and predictor variables, such as the median of request rate and end-to-end response time from the previous minute. MicroBlend collects these variables from the Profiler every minute. The prediction model is then utilized in the process of the service assignment. During scaling-out resources, the prediction model anticipates the total runq latency required to meet SLO target based on input of the median value of request rate observed in the previous minute. Based on the output, the service assignment assigns proper microservices to Lambda.

F. Service Optimization (with User Annotations)

As mentioned previously in Sec. II-B, properly scheduling and combining microservices into a single Lambda can be an effective strategy to enhance performance and decrease SLO violations. To this end, we present a service optimization approach that uses user annotation to create a more optimal service assignment for microservices-based cloud applications. Specifically, as described in Sec. III-C, developers leverage the "BEU FaaS Combine L1" pragma annotation to link multiple microservices that need to be deployed and executed together, thereby (i) reducing the number of network calls required between microservices, (ii) improving performance, and (iii) minimizing SLO violations. Note that, this approach assumes tenants have insights into the application structure and dependencies between different microservices.

The service optimization approach performs service assignments based on user annotations instead of using a heuristic service selection algorithm. During autoscaling, the pragma is detected by the Compiler and forwarded to the Resource Controller, which assigns FaaS to the user-annotated microservices. If there are annotations for function grouping, the service selection groups the microservices with the same prefix (such as "L1") into a single unit. This ensures that the user's preferred placements of microservices are carried out without the need of any manual code rewriting and improves performance utilization. We quantify the effectiveness of this service optimization approach in Sec. V-B.

IV. MicroBlend Implementation

We now elaborate on the implementation details of different components MicroBlend, on AWS.

Compiler: The MicroBlend system utilizes the Python AST [45] (Abstract Syntax Tree) package for automated code transformation, which is a commonly used data structure in compilers to describe the structure of computer code. The

Workload	# of M	Response time (ms)	Requests per vCPU
Social Network	36	31	16
Media Reservation	38	34	18
Online Boutique	11	16	24

TABLE II: Benchmarks. M in the second column means microservices. The third column shows the average response time, and the last column represents the maximum number of requests that are running in parallel per vCPU for VM.

structure provides a representation of the program that can be analyzed and altered in various ways. The MicroBlend's implementation pipeline, which is comprised of 2500 lines of Python code, is illustrated in Fig. 5. If the pragma includes parameters/metrics, it sends them to the Resource Controller and starts building the required Lambda code by examining the function name, parameters, libraries to pack, and global variables. MicroBlend *automatically* adds required library modules for the function's input and output based on the dependencies, such as function calls or data flow. It transforms function calls into Lambda-invoked function calls utilizing data acquired from the outcomes of dependencies. The Lambda code is then generated with the help of an AWS Command Line Interface [7] and a deployment zip file, which is stored in external storage (S3) and contains code for Lambda function as well as library modules and global variables.

Profiler: To gather performance and resource usage measurements for microservices, an advanced open-source monitoring tool, Prometheus [44], is employed. Prometheus can collect data from multiple sources, such as cAdvisor and custom eBPF Exporter. The cAdvisor [15] tool is used to analyze container resource utilization, such as CPU/Memory usage and network bandwidth, on each server. The custom eBPF exporter, using iovisor's bcc tools [12], is designed to collect runq latency histograms per container and export them to Prometheus. The Prometheus tool includes a time-series database to store metric values, and the Resource Controller in MicroBlend uses the *query language* included in the system to extract these values from the database.

Resource Controller: As discussed in detail in Sec. III-E, the heuristic service assignment process uses the runq latencies of microservices, which are obtained through a custom eBPF exporter and stored in Prometheus. Specifically, logistic regression is employed to predict the anticipated runq latency for the observed request rate, such that the resulting value would correspond to a response time that meets the SLO target. The scikit-learn Python ML library [47] is used to implement the logistic regression model.

V. Evaluation

This section provides a description of our evaluation setup and presents our results, which highlight the advantages of using MicroBlend, including the effects of cloud service assignment and service optimization, as well as the overheads associated with employing MicroBlend.

A. Experimental Setup

Workload Generator: To evaluate the benefits of MicroBlend, an accurate event-driven Workload Generator was created. This Workload Generator uses the input provided by the Request Generator (described in Sec. V-A) to generate request arrival times based on actual traces from the real world. Three microservice-based benchmarks were used as our workloads (Table II): (1) Social Network benchmark from DeathStarBench [22] where users read, upload posts, and follow others' posts; we use the workload of uploading new posts using 16 out of 36 microservices; (2) Media Reservation benchmark from DeathStarBench, which consists of 38 microservices and enables users to browse movie information, review (our experimental workload of 15 microservices), rate, and stream movies; and (3) Online Boutique from Google Cloud Platform [41], comprising of 11 microservices that allow users to add items to their online shopping cart and make purchases (our experiment workload of 6 microservices). For the current Python-based compilers, we modify the implementation of the involved microservices to Python-based microservices. [2]

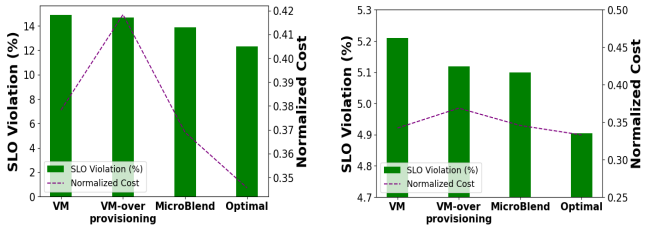
Request Generator: We use two different traces, WITS [50] and Wikipedia [49] (WIKI), as input for the Request Generator. The WIKI trace is a constantly updated record of how users engage with the Wikipedia website. It exhibits irregular bursts of activity over short periods of time, fluctuations in activity levels throughout the day, and a rate of user interactions that follows a Poisson distribution. The WITS trace, on the other hand, has more spikes than the WIKI trace. We scale both traces down to an average of 130 requests per second. To manage this rate, we determine the number of initial servers based on the number of requests each vCPU can process simultaneously per workload, aiming for a response time target of 1000ms (as shown in Table II). We use the first 30 minutes of both WITS and WIKI traces, which include a total of $216{,}377$ and $215{,}327$ requests, respectively.

Cloud configuration: To conduct a fair comparison, we adjust the memory configuration of Lambda to ensure an iso-performance comparison between VM-based and Lambda-based deployments of microservices. As microservices architecture treats a single application as a collection of small and short-lived services, the memory configuration for all microservices is set to 1344 MB. This value is chosen based on the experiments' characteristics, where the average response time for all microservices is approximately 17 ms.

To serve as a front-end for handling and distributing millions of concurrent requests, an Amazon EC2 C5.2xlarge (8 vCPUs, 16 GB RAM) instance is used, equipped with an Nginx load balancer [40]. We utilize the "Least Connection" load balancing algorithm, which assigns requests to the server with the fewest active connections. CloudWatch API is used regularly by the front-end to obtain metrics on the load balancer, CPU/memory usage of active servers, and AWS Lambda's billed duration. The pricing models for both VM and Lambda cover the front-end's running time as well as the number of requests made to the Amazon CloudWatch API.

To handle requests on the server side, we utilize AWS

[2]The current version of our implementation and workloads for evaluation are shared in https://github.com/mjaysonnn/MicroBlend.



(a) WITS (b) WIKI

Fig. 6: SLO (1000ms) violations and cost reductions for the Social Network application under different scaling policies.

EC2 C5.2xlarge instances that are equipped with a library for workload execution and FastAPI [21], a web framework that enables asynchronous programming and inter-communication. To ensure sufficient storage for both the microservices codes and MongoDB data, we utilized an EBS (elastic block store) with a size of 20GB, which was taken into account in the cost model. To avoid incurring high data transfer costs, all our experiments are conducted within the same AWS region. We use five replicas for each microservice in our experiments.

Cost Model: We determine the cost of virtual machines (VMs) by adding up the multiplication of the duration of the workload and the per-second cost of each VM. The cost of Lambda functions is determined based on four factors: the number of invocations, the amount of RAM allocated, the execution time, and the cost of 1 GB-s.

Baselines: For our experiments, we establish a maximum SLO target of 1000ms. We assess the performance of MicroBlend by analyzing its response time (as per SLO specifications) and cost in different scenarios of resource acquisition. We compare the results of MicroBlend in the following three scenarios: (i) Deploying entirely on Lambda (*All-Lambda*); (ii) Using virtual machines with auto-scaling (*VM*); and (iii) Using virtual machines with conservative over-provisioning (*VM-overprovision*), i.e., with 1.5 times the required number of resources. The VM over-provisioning method is commonly used in auto-scaling to prevent SLO violations. To minimize the impact of anomalous cloud noise, our workloads are executed five times.

B. Benefits of MicroBlend

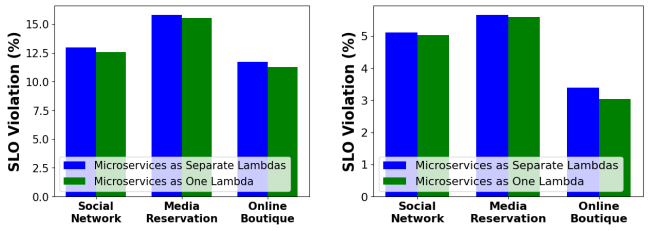
Service Assignment: To evaluate the effectiveness of MicroBlend, we compared its SLO violation count and normalized cost against different scaling policies, as shown in Fig. 6. Specifically, we considered the traditional VM scaling policy, the over-provisioning policy, as stated in subsection V-A, and the optimal solution obtained through an *exhaustive search*, which consisted of an optimal set of combinations offloaded to Lambda. We evaluated the performance of each approach using a set of workloads and measured the SLO violations and normalized costs for each approach. Due to page-count limitation, we present only the figure resulting from the Social Network application. The x-axis on the figure illustrates various methods for obtaining resources. The primary y-axis indicates the percentage of SLO violations (with response time set at 1000 ms), whereas the secondary y-axis displays the cost in monetary terms normalized to the *All-Lambda* resource procurement scheme, which only uses Lambda.

Our findings, presented in Fig. 6a, clearly demonstrate that MicroBlend outperforms both the VM and over-provisioning policies in terms of SLO violation and cost reduction for the WITS. When compared to the VM scaling policy, MicroBlend was able to reduce the SLO violations by up to 7% and achieve up to 2% cost reduction. Also, when compared to the over-provisioning policy, MicroBlend was able to reduce the SLO violations by up to 6% while achieving up to 10% cost reduction. *VM-overprovisioning* policy is mainly focused on acquiring more resources, which results in increased costs. In our experiments with the WITS, out of 216377 total requests, MicroBlend significantly reduced the number of requests that violated the SLO, achieving up to 2264 fewer violations compared to the VM scaling policy and up to 1823 fewer violations compared to the over-provisioning policy.

Fig. 6b presents the evaluation of the WIKI. It can be observed that the MicroBlend approach surpassed the VM scaling policy in terms of both SLO and cost by a small margin of only 1% and 2%, respectively. The total number of requests for the WIKI was 215.327 , and MicroBlend achieved a reduction of up to 194 SLO violations compared to the VM scaling policy. However, it is important to note that this reduction in cost and SLO violations may have a negative impact on the experiment's duration due to the higher per-unit price of Lambda in comparison to VMs and fewer peak requests of WIKI, as compared to WITS.

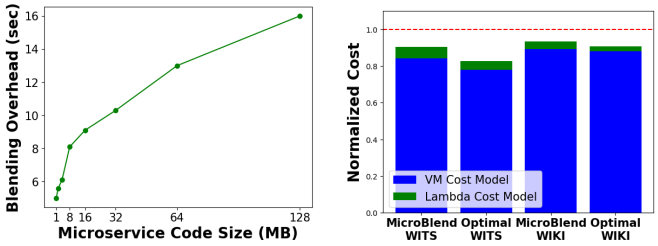
To evaluate the effectiveness of the MicroBlend, we also compared its performance against an optimal solution obtained through exhaustive research, as shown in Fig. 6. The experimental data collected indicates that the MicroBlend method achieves around 87% of the performance of the optimal solution for the WITS and up to 94% for the WIKI in terms of SLO violations. The difference between MicroBlend and the optimal solution is due to the overheads involved in finding the best combination of microservices through the service assignment process. We further discuss the overheads of MicroBlend in Sec. V-C. However, despite this difference, MicroBlend still significantly outperforms other traditional scaling policies in terms of *both* SLO violation and cost reduction.

We assess the effectiveness of MicroBlend's service allocation by conducting performance tests on two additional workloads, Media Reservation and Google Online Boutique, on WITS. MicroBlend results in up to 5% reduction in SLO violations and up to 8% cost savings for the Media Reservation workload. For the Google Online Boutique workload, MicroBlend achieves a reduction of up to 9% in SLO violations and cost savings of up to 11%. Our approach with the Google Online Boutique workload was able to achieve greater improvements in SLO and cost than the other two workloads. This is because, in contrast to the other two workloads, the Google Online Boutique workload uses a sequential workflow consisting of six microservices without any



(a) WITS (b) WIKI

Fig. 7: Comparing SLO violations between two user annotation types: (1) separating Lambdas for each microservice and (2) grouping microservices into one Lambda for three workloads.



(a) Relationship between microservice code size and MicroBlend's blending overhead in seconds.

(b) Cost model breakup of MicroBlend and optimal solution normalized to *VM-overprovisioning*.

Fig. 8: Benefit breakdown of MicroBlend.

involvement of database microservices. In this case, offloading the microservices to Lambda results in a significant reduction of the CPU burden. Although assigning microservices to Lambda on the other two workloads reduces the CPU burden, running microservices on Lambda still invokes MongoDB microservices, which consumes resources in VMs. The higher number of MongoDB microservices and microservices that interact with MongoDB in the Movie Reservation workload, when compared to the Social Network workload, may cause a more burden on the CPU, even when some microservices are offloaded to FaaS. Consequently, there is a lower decrease in the violation of the service level objective (SLO) by 1.7% for the Movie Reservation workload, as opposed to a reduction of 3.24% for the Social Network workload. These findings suggest that the effectiveness of MicroBlend may vary depending on the workflow of microservice-based applications, as well as the involvement of database microservices.

Service Optimization: In order to exhibit the benefits of utilizing user annotation with MicroBlend for complex microservice-based applications, we conducted an experiment aimed at exploring the effect of *grouping microservices* that form cascading workflows through user annotation. We identified particular combinations of microservices for FaaS via service selection and added user annotations to microservices exhibiting cascading patterns. Specifically, two types of user annotations were employed in this experiment for the identified microservices combinations: one to assign microservices as separate Lambda functions, and the other to group them into a single Lambda. For instance, in the Social Network shown in Fig. 1, we grouped microservices 8 and 8A. Similarly, we annotated two microservices, MovieId and Rating, for grouping microservices in the Media Reservation. Lastly, we grouped all microservices in the Online Boutique by annotating them since they all form sequential invocations. MicroBlend then offloads the chosen microservice combinations and assigns them to Lambda during autoscaling.

Fig. 7 provides a comparison of SLO violation and normalized cost for the microservice combinations we chose, based on WITS and WIKI, with two types of user annotations – one for having separate Lambdas for microservices and the other for grouping microservices into a single Lambda. The x-axis represents the different workloads used, while

Workload	WITS	WIKI
Social Network	72%	69%
Media Reservation	68%	67%
Online Boutique	81%	76%

TABLE III: Percentage of response time below SLO during autoscaling.

the y-axis shows the SLO violations. The graph illustrates that, using user annotation yields better results, with SLO reductions of up to 4% for the three workloads, respectively. This improvement is due to MicroBlend's ability to detect user annotations in microservice candidates that have inter-service communication latency and group them into one in an *automated fashion*, resulting in more reduction of SLO violations and cost. MicroBlend will provide a more effective automated blended resource plan by utilizing user annotation, particularly in scenarios with more complex microservices.

C. Breakdown of Benefits

Overheads of MicroBlend: The use of MicroBlend incurs overhead in both compilation and data transfer because it involves compiling and generating Lambda functions when blending resources are utilized. This is achieved by uploading the deployment zip file to an S3 storage service and transferring it to the Lambda computing service. It is critical to minimize the overhead of blending because the allocation of blending resources must be done promptly during autoscaling, which usually takes between 60 to 100 seconds [24], to reduce SLO violations. Otherwise, SLO violations can occur even *before* the new VMs are launched and become operational, similar to the issue with VM-scaling policies. To evaluate the overhead of MicroBlend's blending resources during autoscaling, we conducted experiments that gradually increased the code size (up to 128 MB) and the number of microservices (up to 8), to determine the overhead of compilation and data transfer for each combination of microservices. Note that Lambda functions deployed from S3 have a package size limit of around 250MB.

Fig. 8a illustrates how the blending overhead is related to the code size of each microservice, with the number of microservices set to 8. In this plot, the x-axis represents the microservice code size, ranging from 1MB to 128MB (exponentially), while the y-axis represents the overhead of compilation and data transfer in seconds. As expected, the

blending overhead increases as the code size increases since a larger code requires more processing time for blending. The minimum time was approximately 5 seconds, while the maximum time was around 32 seconds, corresponding to the largest code size of 230 MB. The graph indicates that the overhead of compilation and data transfer increases with code size and the number of microservices, which may limit the use of blending in larger-scale microservice-based applications. Nevertheless, given the small size of microservices (all the microservices' code sizes were less than 1MB), blending resources can be used effectively to reduce the overall cost and SLO violations during autoscaling.

Cost Model Breakdown: Fig. 8b plots the cost model breakdown for MicroBlend and the optimal solution, with the total cost normalized to the cost of executing the *VM-overprovisioning* policy. It can be observed from this figure that, compared to the *VM-overprovision* policy, MicroBlend reduces the cost of VMs by 17% and 11% for WITS and WIKI, respectively. However, when compared to the optimal solution, MicroBlend incurs higher costs for both VM and Lambda. This is because MicroBlend incurs an overhead due to mispredicting suitable microservice candidates for FaaS to reach the SLO target, which results in additional unnecessary scaling-out decisions and increased cost models for VM and Lambda. In contrast, the optimal solution can immediately use Lambda to reduce the queuing of queries to existing VMs, leading to an earlier scale-in of the unused VMs spawned during the scale-out period, resulting in higher cost reductions.

Overhead of Learning Framework: To gain a deeper understanding of the difference between MicroBlend's service assignment algorithm and the optimal solution regarding cost and SLO violation reduction, we performed an evaluation of the learning framework prediction model. We calculated the entire request count handled by our blended resources for each workload and computed the fraction of requests that met the SLO target of 1000ms (i.e., a response time that is less than or equal to 1000ms). The purpose of this evaluation is to determine whether the service assignment algorithm was able to effectively recommend suitable microservices for Lambda, based on the predicted runq latency metric from our learning framework, in order to meet the SLO target.

Table III shows the percentage of requests that were able to meet the SLO target of 1000ms during autoscaling for each workload under WITS and WIKI. We observed that the service assignment algorithm was able to reduce SLO violations by up to 81% and 76% for WITS and WIKI, respectively. As previously stated, the Google Online Boutique workload utilizes a sequential workflow that excludes database microservices. Deploying microservices to Lambda leads to a considerable decrease in CPU load, reducing response times and exhibiting superior predictive capabilities compared to other workloads. It is worth noting that the service assignment algorithm was not able to significantly reduce the SLO violations in all scenarios compared to the optimal solution. This suggests there may be potential for further improvement in the algorithm or other system aspects to further decrease the number of SLO violations during autoscaling, which we discuss in Sec. VII. Nonetheless, the blending process decreases the queuing of queries into current VMs, resulting in fewer SLO violations during autoscaling than the traditional VM policies.

VI. Related Work

The challenges of microservices' resource management also have recently begun to be studied [23], [36], [51]. Sage [23] utilizes a machine learning-driven, scalable root cause analysis for interactive cloud microservices. Inagaki et al. [28] proposed detecting the bottleneck of microservice-based applications by profiling the number and status of working threads as well as the dependency between microservices through network connections. Bajaj et al. [11] converts monolithic applications to microservices and serverless services. SHOWAR [10] offers an effective resource allocation strategy for microservice clusters based on past resource utilization and kernel-level metrics. In comparison to previous cloud scheduling studies, MicroBlend focuses on the resource challenges of microservice cluster-autoscaling.

Recent proposals aim to improve the cost- and latency-efficiency of serverless operations [17], [33]–[35]. WiseFuse [35] pinpoints the main serverless performance bottlenecks and suggests merging functions to lessen communication and computation skew. However, combining functions involves manual effort, identifying function names and calls, and adding remote storage and execution packages. MicroBlend provides a compiler-based automated creation of blending services, including grouping Lambdas as one.

VII. Concluding Remarks and Future Work

We propose MicroBlend, a platform that integrates IaaS and FaaS for microservice-based applications in the cloud. MicroBlend combines the two service offerings via a performance model and produces a service allocation plan that meets the user-specified latency service-level objectives (SLO) while keeping the cost at minimum. Our comprehensive experimental evaluation of three microservice-based applications reveals that MicroBlend reduces SLO violations by up to 9% while saving up to 11% on financial costs, compared to traditional cluster resource procurement approaches.

We plan to explore the potential benefits of other machine learning algorithms for service selection in microservices-based architecture. Specifically, we plan to employ a "black-box" approach, such as a single deep (large) neural network (DNN), in identifying the suitable microservices for FaaS; DNNs [32] are particularly useful in scenarios where the number of input features is large and complex, such as in microservices architectures with numerous dependencies. We believe they can extract and model high-level features and non-linear correlations between inputs, making them well-suited for microservice assignments.

Acknowledgement

We thank the reviewers for their insightful feedback. This work is supported in part by NSF grants 2211018, 2122155, 1931531, 1763681, and 1908793.

REFERENCES

[1] Microsoft Microservices Architecture Guide. https://www.slideshare.net/apigee/adapt-or-die-a-microservices-story-at-google.

[2] Adopting microservices at Netflix. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.

[3] O. Alipourfard et al. Cherrypick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *Proc. NSDI*, 2017.

[4] Marcelo Amaral et al. Microlens: A performance analysis framework for microservices using hidden metrics with bpf. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 230–240. IEEE, 2022.

[5] Amazon Elastic Kubernetes Service. https://aws.amazon.com/eks/.

[6] Amazon cloudwatch. https://aws.amazon.com/cloudwatch/.

[7] Aws command line interface. https://aws.amazon.com/cli/.

[8] AWS S3. https://aws.amazon.com/s3/.

[9] Azure Kubernetes Service. https://azure.microsoft.com/en-us/services/kubernetes-service/overview.

[10] Ataollah Fatahi Baarzi and George Kesidis. Showar: Right-sizing and efficient scheduling of microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 427–441, 2021.

[11] Deepali Bajaj et al. Partial migration for re-architecting a cloud native monolithic application into microservices and faas. In *International Conference on Information, Communication and Computing Technology*, pages 111–124. Springer, 2020.

[12] Bcc - tools for bpf-based linux. https://github.com/iovisor/bcc.

[13] bpftrace. https://bpftrace.org/.

[14] Brendan Burns. Designing distributed systems: Patterns and paradigms for scalable, reliable services. *O'Reilly Media, Inc.*, 49(4):127–144, 2018.

[15] google/cadvisor. https://github.com/google/cadvisor.

[16] CPU Utilization is Wrong. http://www.brendangregg.com/blog/2017-05-09/cpu-utilization-is-wrong.html.

[17] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. Photons: Lambdas on a diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 45–59, 2020.

[18] eBPF. https://ebpf.io/.

[19] ebp_runq_latency. http://www.brendangregg.com/blog/2016-10-08/linux-bcc-runqlat.html.

[20] Xiaobo Fan et al. Power provisioning for a warehouse-sized computer. In *ISCA*, pages 13–23, 2007.

[21] Fastapi framework. https://fastapi.tiangolo.com/.

[22] Yu Gan et al. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems. In *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2019.

[23] Yu Gan et al. Sage: Practical and scalable ml-driven performance debugging in microservices. *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, pages 135–151, 2021.

[24] A. Gandhi et al. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Trans. Comput. Syst.*, 2012.

[25] Google Kubernetes Engine. https://cloud.google.com/kubernetes-engine.

[26] J.R. Gunasekaran et al. Spock: Exploiting Serverless Functions for SLO and Cost Aware Resource Procurement in Public Cloud. In *IEEE CLOUD*, 2019.

[27] David W Hosmer Jr et al. Applied logistic regression. 2013.

[28] Tatsushi Inagaki et al. Detecting layered bottlenecks in microservices. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 385–396. IEEE, 2022.

[29] Aman Jain et al. Splitserve: Efficiently splitting apache spark jobs across faas and iaas. In *Proceedings of the 21st International Middleware Conference*, pages 236–250, 2020.

[30] Kubernetes: Production-grade Container Orchestration. http://kubernetes.io/.

[31] Joshua Levin and Theophilus A Benson. Viperprobe: Rethinking microservice observability with ebpf. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*, pages 1–8. IEEE, 2020.

[32] Huaxiong Li, Libo Zhang, Xianzhong Zhou, and Bing Huang. Cost-sensitive sequential three-way decision modeling using a deep neural network. *International Journal of Approximate Reasoning*, 85:68–78, 2017.

[33] Ashraf Mahgoub, Li Wang, Karthick Shankar, Yiming Zhang, Huangshi Tian, Subrata Mitra, Yuxing Peng, Hongqi Wang, Ana Klimovic, Haoran Yang, et al. {SONIC}: Application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 285–301, 2021.

[34] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. {ORION} and the three rights: Sizing, bundling, and prewarming for serverless {DAGs}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 303–320, 2022.

[35] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh Bagchi, and Somali Chaterji. Wisefuse: Workload characterization and dag transformation for serverless workflows. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(2):1–28, 2022.

[36] Jason Mars et al. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–259, 2011.

[37] Nabor C Mendonça et al. Developing self-adaptive microservice systems: Challenges and directions. *IEEE Software*, 38(2):70–79, 2019.

[38] Microservices Architecture on Google App Engine. https://cloud.google.com/appengine/docs/standard/python/microservices-on-app-engine.

[39] Microsoft Microservices Architecture Guide. https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices.

[40] Nginx Load-balancing methods. https://nginx.org/en/docs/http/load_balancing.html.

[41] Google Cloud Platform's Demo Microservice. https://github.com/GoogleCloudPlatform/microservices-demo.

[42] Claus Pahl and Pooyan Jamshidi. Software architecture for the cloud–a roadmap towards control-theoretic, model-based cloud architecture. In *European Conference on Software Architecture*, pages 212–220. Springer, 2015.

[43] Austin Parker et al. *Distributed tracing in practice: Instrumenting, analyzing, and debugging microservices*. O'Reilly Media, 2020.

[44] prometheus. https://prometheus.io/.

[45] python ast module. https://docs.python.org/3.6/library/ast.html.

[46] V.M. Do Rosario et al. Fast and low-cost search for efficient cloud configurations for hpc workloads. https://arxiv.org/abs/2006.15481, 2020.

[47] Scikit-learn: Machine Learning in Python. https://scikit-learn.org/.

[48] Myungjun Son et al. Splice: An automated framework for cost-and performance-aware blending of cloud services. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 119–128. IEEE, 2022.

[49] Wikimedia database dump. https://archive.org/details/enwiki-20180320.

[50] WITS: Waikato Internet Traffic Storage. https://wand.net.nz/wits/.

[51] Hailong Yang et al. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. *ACM SIGARCH Computer Architecture News*, 41(3):607–618, 2013.

[52] C. Zhang et al. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *Proc. ATC*, 2019.

[53] Hao Zhou et al. Overload control for scaling wechat microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 149–161, 2018.