Performance Modeling and Optimization of Microservice-based Applications in Clouds

Liang Bao, *Member, IEEE,* Chase Wu, *Member, IEEE,* Nana Ren, Mengqing Shen and Xiaoxuan Bu

Abstract—Microservice has been increasingly recognized as a promising architectural style for constructing large-scale cloud-based applications within and across organizational boundaries. This microservice-based architecture greatly increases application scalability, but meanwhile incurs an expensive performance overhead, which calls for a careful design of performance modeling and task scheduling. However, these problems have thus far remained largely unexplored. In this paper, we develop a performance modeling and prediction method for independent microservices, design a three-layer performance model for microservice-based applications, formulate a Microservice-based Application Scheduling problem for minimum end-to-end delay under a user-specified Budget Constraint (MAS-BC), and propose a heuristic microservice scheduling algorithm. The performance modeling and prediction method are validated and justified by experimental results generated through a well-known microservice benchmark on disparate computing nodes, and the performance superiority of the proposed scheduling solution is illustrated by extensive simulation results in comparison with existing algorithms.

Index Terms—Microservice, performance modeling and prediction, task scheduling, cloud computing.

1 Introduction

The emerging microservice architecture has been increasingly adopted in application design and development [1]. A microservice-based application running in the cloud typically involves the execution and interoperation of multiple microservices, each of which can be implemented, deployed, and updated independently without compromising the application's integrity. Such independence in microservice management greatly increases the application's scalability, portability, updatability, and availability, but at the cost of expensive performance overhead [2] and complex resource use dynamics [3]. Therefore, performance modeling and task scheduling are among the most critical design concerns in microservice-based applications in container-based public clouds [4].

Different from traditional monolithic applications [1], microservice-based applications bring two new challenges to performance modeling and task scheduling [3]: i) characterizing the performance of a microservice must consider some new properties, such as application feature, workload interference, and resource contention; and ii) as public clouds make computing a paid utility, application providers are facing the challenge of reducing financial cost in addition to meeting the traditional goal of performance optimization in dynamic environments shared by multiple users with many runtime uncertainties. For the first challenge, benefiting from the containerized environment and the popular "service instance per container" deployment pattern [5], such interference and contention could be encapsulated at the container level. More specifically, a container may be imposed with certain limits on the CPU and memory resources consumed by a service instance, and hence the service instance is isolated and has an exclusive access to some virtual resources. As a result, it is not necessary to address the interference and contention problem at the microservice level. For the second challenge, however, the existing performance monitoring and modeling frameworks, such as Nagios [6], Ganglia [7], and Heapster [8], collect measurements and calculate statistics for CPU, memory, filesystem, and network usages, but not for microservice-level performance metrics. Also, many widely adopted resource scheduling frameworks, such as Kubernates [9] and Mesos [10], only offer simple rule-based scheduling strategies. So far, very limited efforts have been directly devoted to these microservice-specific performance modeling and resource scheduling problems.

In this paper, we develop a performance modeling and prediction method for independent microservices, design a three-layer performance model for microservice-based applications, formulate a Microservice-based Application Scheduling problem for minimum end-to-end delay under a user-specified Budget Constraint in public clouds, referred to as MAS-BC, and propose a heuristic scheduling solution to MAS-BC for one single application consisting of multiple microservices. Specifically, our work makes the following contributions to the field:

1) The proposed method for performance modeling and prediction is based on generic computer system and program models, which provide an accurate account of microservice-based applications and make it directly applicable to different microservices running on various cloud computing platforms.
2) The MAS-BC problem is formulated based on rigorous, comprehensive, and practical models, shown to be NP-complete, and solved effectively and efficiently by the proposed heuristic scheduling approach that employs a greedy recursive critical path strategy.
3) The proposed performance modeling and prediction method is validated and justified by experimental results using a well-known microservice benchmark on disparate computing nodes, and the performance superiority of the proposed heuristic scheduling approach is illustrated by extensive simulation results in comparison with existing algorithms.

The remainder of this paper is organized as follows. Section 2

- *L. Bao, N. Ren, M. Shen, X, Bu are with the School of Software, XiDian University, Xi'an, ShaanXi, 710071.*
  E-mail: baoliang@mail.xidian.edu.cn.
- *C. Wu is with Department of Computer Science, New Jersey Institute of Technology, Newark, NJ 07102.*
  E-mail: chase.wu@njit.edu.

Manuscript received June 22, 2018; revised XX XX, 2018.

surveys related work and Section 3 introduces microservice-based applications in clouds. Section 4 constructs the model of a microservice for performance estimation. Section 5 presents the analytical models of a microservice-based application and formulates the MAS-BC problem. Section 6 designs a resource scheduling algorithm. Section 7 describes the experimental setup and evaluates the performance model and scheduling algorithm using both experiments and simulations.

2 Related Work

We discuss existing efforts on three related topics, i.e., performance modeling and prediction, workflow-based application scheduling, and some early studies on the microservice architecture.

2.1 Performance Modeling and Prediction

Modeling and prediction of system performance have been investigated in-depth for decades. The existing efforts for execution time estimation can be broadly divided into three categories, namely static, dynamic, and hybrid [11].

In a static scheme, a complete and concrete performance model must be constructed *a priori* in the current execution environment without considering the dynamically changing state of the resources. Chen *et al.* [12] proposed an empirical approach to predict the performance of enterprise applications based on component technologies such as CORBA and J2EE through benchmarking and profiling. Aguiler *et al.* [13] designed tools to isolate performance bottlenecks in distributed systems composed of black-box nodes.

A dynamic scheme attempts to predict the performance in the future by using dynamic information together with some prediction results. Jang *et al.* [14] predicted the execution performance of computing modules on a target machine based on historical application-level performance data generated by the instrumentations of applications. Sadjadi *et al.* [15] proposed a regression-based dynamic execution time prediction method, which achieved an accurate prediction for computational processes with an increasing number of nodes. Barnes *et al.* [16] used multivariate regression analysis to predict the running time of executables on high-end machines by taking low-end processor configurations as a training set.

In a hybrid scheme, researchers have taken into consideration both the static and dynamic properties of an executable and a computational node to predict the execution time. Goldsmith *et al.* [17] proposed a method for measuring a program's empirical computational complexity by running the program on workloads spanning several orders of magnitude in size, measuring their performance, and fitting these observations to a model that predicts the performance as a function of workload size. Thereska *et al.* [18] constructed practical models to predict the performance of popular client applications based on the understanding of performance consequences of a change in software, hardware or configuration. Wu *et al.* [11] proposed a statistical learning approach to model scientific computations and predict their execution time based on a combination of both hardware and software properties.

The proposed method follows the line of the hybrid methodology and differs from the existing efforts in two aspects:

i) We consider request caching, business processing, and database transaction for a microservice in a systematical manner.
ii) To the best of our knowledge, our work is among the first to model and predict the performance of microservice-based applications in clouds.

2.2 Workflow Scheduling in Clouds

Task scheduling for workflow-based applications in clouds has been extensively studied in the literature in the past decade. Abrishami *et al.* [19] designed a QoS-based workflow scheduling algorithm based on Partial Critical Paths (PCP) in SaaS clouds to minimize the cost of workflow execution within a user-defined deadline. Mao *et al.* [20], [21] investigated the automatic scaling of clouds with budget and deadline constraints and proposed Scaling-Consolidation-Scheduling (SCS) with VMs as basic computing elements. Hacker *et al.* [22] proposed a combination of four scheduling policies based on an on-line estimation of physical resource usage. Zeng *et al.* [23] proposed a budget-conscious scheduler to minimize many-task workflow execution time within a certain budget. Zheng *et al.* [24] extended the classic Heterogeneous Earliest Finish Time (HEFT) algorithm [25] and proposed BHEFT to include budget constraint. Arabnejad *et al.* [26] proposed HBCS to minimize workflow makespan under a cost constraint. Wu *et al.* [27] constructed performance analytical models for scientific workflows using cloud-based computing resources, and proposed a critical-greedy algorithm to achieve minimal end-to-end delay for scientific workflows under budget constraints in clouds.

In this paper, we construct analytical models to quantify the performance of workflow-structured, microservice-based applications in IaaS clouds with VMs as computing elements, and incorporate the practical instance billing model into cost calculation. These models facilitate a rigorous formulation of a budget-constrained delay minimization problem for microservice-based applications in clouds.

2.3 Studies on Microservice Architecture

Proposed in 2014 [28], microservice has recently emerged as a promising enterprise application architectural style. Ever since, researchers have conducted many studies and shared their experiences in applying such a style to the implementation of different software systems, such as workflow engine [29], [30], cloud platform [31], [32], IoT middleware [33], and science gateway [34].

There also exist a number of efforts on the performance of microservice-based applications such as [35], [36], [37], [38], [38], [2], [39], [40], [41], [42], [43], [44], [45]. In [36], Hasselbring discussed how microservices support scalability for both runtime performance and development performance, via polyglott persistence [46], eventual consistency, loose coupling, open source frameworks, and continuous monitoring for elastic capacity management. In [37], Villamizar *et al.* presented a cost comparison of a Web application developed and deployed using the same scalable scenarios with the monolithic and microservice architecture, respectively. Their results show that microservices can help reduce infrastructure cost in comparison with standard monolithic architectures. In [38], Knoche *et al.* presented a simulation-based approach for sustaining runtime performance during incremental modernizations towards microservices. In [2], Ueda *et al.* used an open-source Web service benchmark named Acme Air to

compare the performance of benchmark applications with both microservice and monolithic architectures. They observed that the microservice architecture incurs a significant overhead and its performance could be 79.2% lower than the monolithic version. Based on these findings, they pointed out the necessity of developing optimization techniques for microservice workloads. In [39], Heinrich *et al.* argued why new solutions to performance engineering for microservices are needed, and identified some open issues and outline possible research directions with regard to performance-aware testing, monitoring, and modeling of microservices. In [40], Zhou *et al.* identified the gap between existing benchmark systems and industrial microservice systems, and then proposed a medium-size benchmark system of microservice architecture. In [41], Lloyd *et al.* presented results from their investigation into the factors which influence microservice performance afforded by serverless computing. They identified four states of serverless infrastructure including: *provider cold*, *VM cold*, *container cold*, and *warm* and demonstrate how microservice performance varies up to 15x based on these states. In [42], Gotin *et al.* presented a case study to investigate which performance metrics to be used by a threshold-based auto-scaler for scaling consuming microservices of a message queue in order to prevent overloaded queues and to avoid SLA violations. In [43], Engel *et al.* proposed an evaluation approach for microservice architectures based on identified architecture principles from research and practice like a small size of the services, a domain-driven design or loose coupling. In [44], Saman *et al.* used Kieker framework and Kieker trace analysis tools to monitor and analyze the performance of an existing microservice-based application. In [45], Gribaudo *et al.* provided a simulation based approach to explore the impact of microservice-based software architectures in terms of performances and dependability, given a desired configuration.

In sum, there have been a number of efforts devoted to modeling and predicting application performance in different computing environments, and many practices in constructing new systems and modernizing existing systems using the microservice architecture. However, very limited efforts have been made to construct a general performance model for microservice-based applications and determine optimal resource scheduling to deploy such applications in public clouds. Our work makes an attempt to fill up the gap in this area.

3 A System Overview

As shown in Figure 1, the microservice architecture [47] defines a framework that structures the application as a set of loosely coupled, collaborating, and interconnected services, each of which implements a set of related functions. Services are developed and deployed independently at different locations, and use either synchronous protocols such as HTTP/REST or asynchronous protocols such as AMQP [48] for communication. Typically, each service is associated with its own database in order to be decoupled from others.

These distributed microservices can be composed by an aggregator [49] to provide a certain application functionality. As shown in Figure 2, if there are multiple clients that need to access Services A, B, and C, it is recommended to abstract such logic as one computing unit and aggregate them into a single composite microservice. One advantage of abstracting at this level is that the individual services, i.e., Services A, B, and C, can evolve

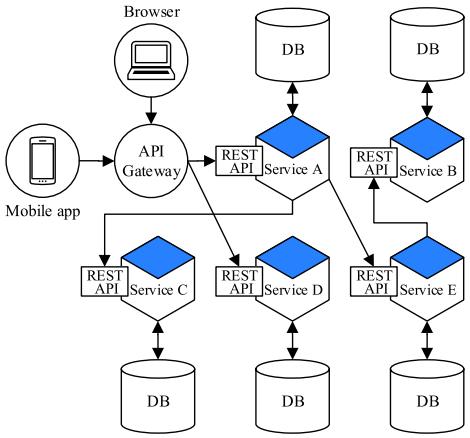

Fig. 1. Microservice architecture [47].

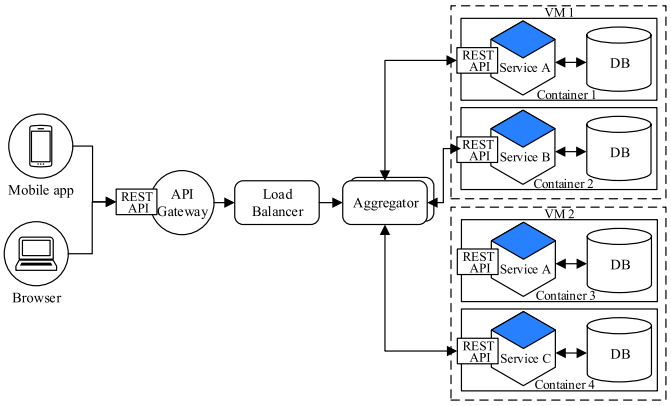

Fig. 2. Microservice composition and deployment [49].

independently while still providing the business process through the composite microservice.

With the support of the aggregator, the provider of an application can define a set of required microservices and their relations using a workflow description language to meet a certain business goal. Then, such applications are published to a single entry point called API gateway [50], which hides the internal architecture and provides APIs tailored to each user.

In the deployment phase, each microservice is encapsulated into an independent execution environment, referred to as a container, which provides certain CPU, memory, and I/O resources [51]. These containers are then grouped and deployed on a collection of interconnected virtual machines (VMs) provisioned by a public cloud provider.

In the execution phase, users are typically unaware of the microservices involved, and therefore only send requests with certain budget constraints, i.e., the maximum financial cost they can afford, to run desired applications. A corresponding fee would be charged once the requested application has been successfully executed.

According to the previous studies including the work in [52], for a given microservice-based application, the microservice mapping scheme, which decides the container-held microservice for each function of the application, plays an important role in achieving the optimal application performance. In particular, our work focuses on microservice-based application scheduling to minimize the end-to-end delay of a given application under a pre-specified

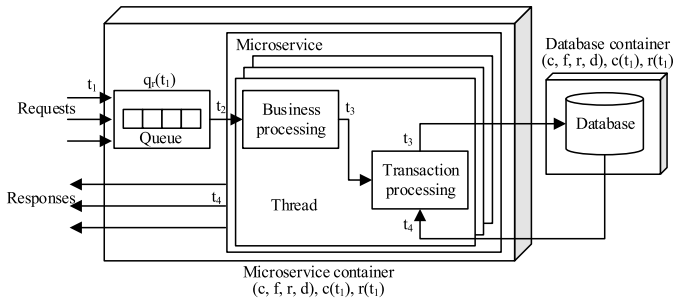

Fig. 3. The process of handling requests in a container.

budget constraint.

4 Performance Modeling and Prediction of a Microservice

We use *processing time (PT)* as the performance metric of a microservice, which denotes the period of time from the time point when a request is made by a user to the time point when the result is received from the microservice. This performance model is quite generic and used by the prediction method for estimating the *PT* of a given microservice that is located in a certain container at any time point t.

4.1 Microservice Performance Modeling

To depict the *PT* of a microservice, we need to define a general procedure to handle concurrent requests in a microservice running in a container, as illustrated in Figure 3.

The overall time overhead for processing a request consists of three components. The first overhead is incurred when all concurrent requests are competing for computing resources provided by the container. The second overhead is introduced to execute the pre-defined business logic within a microservice in a multi-threading situation. The third overhead arises from executing related transactions in the backend database. Therefore, we define the *PT* of handling a request for a microservice in a container at time point t as follows:

$$PT(t) = f_1(\mathcal{R}) + f_2(C) + f_3(\mathcal{D}) + \lambda, \tag{1}$$

where

- $f_1(\mathcal{R})$ denotes the request caching time ($t_2 - t_1$ in Figure 3) under the concurrent execution situation in the container,
- $f_2(C)$ denotes the processing time ($t_3 - t_2$ in Figure 3) to execute the pre-defined business process of a request,
- $f_3(\mathcal{D})$ denotes the transaction processing time ($t_4 - t_3$ in Figure 3) of performing I/O operations in the backend database, and
- λ is a regression constant.

4.1.1 Modeling of Request Caching: $f_1(\mathcal{R})$

Let vector $H(t)$ represent the hardware profile[1] of a given container that consists of both static configuration and dynamic usage information at time point t, defined as:

$$H(t) = h_s \cup h_d(t), \tag{2}$$

1. We define the CPU, memory and disk properties of a container by setting related parameters. More details can be found at https://docs.docker.com/engine/reference/run/.

where h_s denotes the set of static features and $h_d(t)$ denotes the set of dynamic features at time point t. Typically, the hardware specifications of the container are considered static while the workloads in the container are dynamic in nature, therefore resulting in a time-varying resource capacity.

Inspired by Wu's work [11], we use $C_h(t)$ to represent the effective processing (EP) power provided by the available hardware resources of the container at time point t. The EP power determines the number of usable CPU cycles (i.e. the total number of CPU cycles minus those used for system overheads and other background workloads). The relationship between the $H(t)$ and $C_h(t)$ can be described as:

$$C_h(t) = H(t) \cdot \theta_h + \vartheta_h, \tag{3}$$

where θ_h is the coefficient vector of a regression estimate for hardware profiles and ϑ_h denotes the set of regression constants.

We define the request caching time $T_r(t)$ in the container at time point t as follows:

$$T_r(t) = \delta_1 \cdot \frac{q(t)}{C_h(t)} + \delta, \tag{4}$$

where $q(t)$ is the number of waiting requests in the queue, δ_1 is the coefficient vector, and δ is the regression constant. Let $\mathcal{R} = \frac{q(t)}{C_h(t)}$, and we have:

$$f_1(\mathcal{R}) = \delta_1 \cdot \mathcal{R} + \delta. \tag{5}$$

4.1.2 Modeling of Business Processing: $f_2(C)$

To model the processing time for handling a business request in a microservice, we need to consider the software features of the microservice (i.e. program). Note that a business process in a microservice has a fixed time complexity related to the size of input parameters [17], without considering the I/O operations of database. Based on this observation, we also employ a regression model to estimate the business processing time $T_b(t)$ of a microservice at time point t:

$$T_b(t) = \theta_1 \cdot \frac{f(\alpha)}{C_h(t) \cdot x} + \vartheta, \tag{6}$$

where $f(\alpha)$ denotes the empirical time complexity of the program defined in [17], α is the size of input parameters, x represents the number of concurrent threads in the microservice, θ_1 denotes the coefficient vector of the regression estimate for the processing overhead, and ϑ denotes the set of regression constants. Let $C = \frac{f(\alpha)}{C_h(t) \cdot x}$, and we have:

$$f_2(C) = \theta_1 \cdot C + \vartheta. \tag{7}$$

For a specific microservice running in a given container, we assume that the available hardware resources $H(t)$ of the container and the business processing time $T_b(t)$ in that container follow a joint probability distribution $P_{H(t),T_b(t)}$. By using the least squares estimate, we calculate the business processing time estimation error as:

$$I_{\theta,\vartheta} = \int (\hat{T}_b(t) - T_b(t)) dP_{H(t),T_b(t)}, \tag{8}$$

where $\hat{T}_b(t)$ denotes the actual amount of processing time. The best estimator for the actual execution time is given by minimizing Eq. (8). To obtain the best estimation, one would need an accurate distribution $P_{H(t),T_b(t)}$, which is very hard to find in practice. The

problem may not be tractable if this distribution is complex [53]. Let $I^*_{\theta,\vartheta}$ denote the best estimation given by Eq. (8):

$$I^*_{\theta,\vartheta} = \min(I_{\theta,\vartheta}). \tag{9}$$

The goal of the estimation is to have a high probability that the estimation error is within some permissible limit. Using the result from the sample size estimation [54], we have:

$$P((I_{\theta,\vartheta} - I^*_{\theta,\vartheta}) > \epsilon) \leq 1 - 8(\frac{64e}{\epsilon} \ln \frac{64e}{\epsilon})^d e^{-\epsilon^2 \frac{k}{512}}, \tag{10}$$

where k is the sample size and ϵ is an acceptable estimation error.

As mentioned in [53], the same argument holds on the estimation bound. Note that this estimation bound does not depend on the form of distribution $P_{H(t),T_b(t)}$, but the drawback is that it only ensures the closeness of the estimation error to the best possible linear approximation. It is possible that the latter itself may not be satisfactory if the underlying relationship is non-linear.

4.1.3 Modeling of Transaction Processing: $f_3(\mathcal{D})$

In this paper, we focus on the online transaction processing (OLTP) context for estimating transaction processing time because OLTP systems are very common in business processing scenarios, such as order entry, retail sale, and financial transaction.

OLTP systems have an emphasis on fast query processing and data integrity in multi-access environments. Previous studies [55] show that an I/O-bound transaction has a latency directly proportional to the time it waits for a logical I/O on average plus a relatively fixed time cost of CPU and an overhead for database operation. Based on this conclusion, we define the processing time $T_d(t)$ of a transaction within a database at time point t as follows:

$$T_d(t) = \phi_1 \cdot \frac{N_r(t)}{C_h(t) \cdot y} + \varphi, \tag{11}$$

where $N_r(t)$ is the number of records in the database at time point t, y denotes the multi-programming level (MPL) [55] of the database, ϕ_1 is the coefficient vector of the regression estimate for the query overhead of the transaction, and φ denotes the set of regression constants. Let $\mathcal{D} = \frac{N_r(t)}{C_h(t) \cdot y}$, we have:

$$f_3(\mathcal{D}) = \phi_1 \cdot \mathcal{D} + \varphi. \tag{12}$$

4.2 Microservice Performance Prediction

Based on the above performance models, we identify four categories of features that may affect the performance of a microservice, as tabulated in Table 1.

All these features have been defined and explained in the previous section, except *hardware features*, which could be an excessively long list. We only consider the most significant properties, such as the size of used RAM, which provides important information on the amount of concurrent workloads in the container, and the buffer size, which largely affects the performance of I/O operations. These features can be collected by some system tools such as the *free* command in Linux.

Once these features are measured or calculated using either system or third-party tools, we combine them to obtain an estimate of the microservice processing time based on our regression techniques, as shown in Figure 4. Note that we choose different workloads for the program and feed them to the *trend-prof* tool [17] to calculate the empirical time complexity of the program.

TABLE 1
Features used for performance prediction.

Description	Symbol	Belong to
Hardware feature		
Number of CPU cores	c	$\mathcal{R}, C, \mathcal{D}$
Frequency of each CPU core	f	$\mathcal{R}, C, \mathcal{D}$
Total RAM size	r	$\mathcal{R}, C, \mathcal{D}$
Total disk space	d	$\mathcal{R}, C, \mathcal{D}$
Size of used RAM at t	$r(t)$	$\mathcal{R}, C, \mathcal{D}$
Size of used buffer at t	$b(t)$	$\mathcal{R}, C, \mathcal{D}$
Request feature		
Number of already queued requests at t	$q(t)$	$\mathcal{R}$
Business processing feature		
Size of input parameters	α	C
Empirical time complexity of a program	$f(\alpha)$	C
Number of concurrent threads	x	C
Database and transaction feature		
Number of records in the database at t	$N_r(t)$	$\mathcal{D}$
Multi-programming level (MPL) of database	y	$\mathcal{D}$

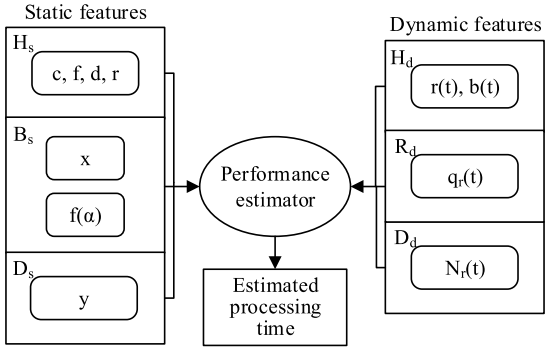

Fig. 4. The prediction model of a microservice.

5 Analytical Models and Problem Formulation

Microservice-based application scheduling in clouds can be characterized by properties from three components, i.e., *application model*, *system model*, and *performance model*, as illustrated in Figure 5. The application model defines "the structure of a business workflow-based application being scheduled"; the system model describes "the VMs and underlying network that are used to execute the application"; the performance model corresponds to "what we attempt to optimize". This three-component view is also consistent with the well-known three-field notation used to define classical scheduling problems [56].

5.1 Application Model

As commonly adopted in workflow methods, a microservice-based application can be modeled as a directed acyclic graph (DAG) $G_a = (V_a, E_a)$, where vertices $V_a = \{f_1, f_2, \cdots, f_m\}$ represent m functions, and f_1 and f_m denote the start and end functions, respectively. The execution dependency between a pair of adjacent functions f_i and f_j is denoted by a directed edge $e_{i,j} \in E_a$ between them, with weight $w_{i,j}$ denoting the size of data transferred from f_i to f_j. A function f_i receives a data input from each of its preceding functions and performs a predefined business logic. Upon successful execution, the output data of f_i is sent to its succeeding functions immediately. As shown in Figure 5(a), f_i receives input data $w_{j,i}$ and $w_{k,i}$ from f_j and f_k, performs its own function, and sends output data $w_{i,p}$ to f_p once it completes its execution.

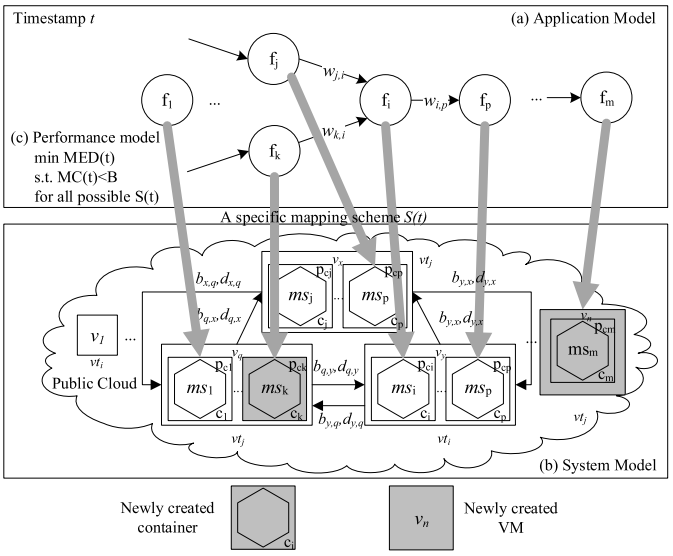

Fig. 5. Analytical models of a microservice-based application.

5.2 System Model

The underlying environment for running microservice-based applications is supported by IaaS where VMs are reserved, deployed and executed on interconnected physical computers. We model such a cloud environment as a fully-connected directed graph $G_v = (V_v, E_v)$ consisting of a set $V_v = \{v_1, v_2, ..., v_n\}$ of VMs and a set E_v of directed links between them. Each communication link $l_{i,j}$ between VMs v_i and v_j is associated with bandwidth (BW) $b_{i,j}$ and minimum link delay (MLD) $d_{i,j}$.

Each function of an application is implemented by a microservice, and each microservice needs to be deployed in a container, whose computing capability is defined as a four-tuple $p_{c_i} = (c, f, r, d)$, as shown in Table 1. As illustrated in Figure 5(a, b), function f_i is mapped to and implemented by microservice ms_i, which is then encapsulated in container c_i with computing capability p_{c_i} deployed on VM v_y.

In IaaS, cloud providers lease computing resources in the form of interconnected VMs. Typically, different types of VMs with different computing capabilities characterized by the number of virtual CPU cores (c), CPU frequency (f), RAM (r) and disk size (d) are provisioned[2] to satisfy different application needs. Different VM types are charged at different prices. Table 2 shows the configurations and prices of five on-demand instance types offered by Amazon EC2[3]. Note that Amazon adopts a step billing model, whereby partial VM hour usage is rounded up to one hour. Our work targets computing environments in a single public cloud where physical machines are interconnected by a high-bandwidth intranet, and hence we do not specifically consider data transfer cost.

More generally, we consider a public cloud environment with k VM types $VT = \{vt_1, vt_2, \cdots, vt_k\}$, where each type vt_i is associated with both cost- and performance-related attributes, and defined as:

$$vt_i = \{p_{vt_i}, C_{vt_i}\}, \tag{13}$$

2. For example, Amazon provides an "Instance Types Matrix": https://aws.amazon.com/ec2/instance-types/#instance-type-matrix.

3. EC2 pricing policy: http://aws.amazon.com/ec2/pricing/.

TABLE 2
Configurations and prices of five different memory-optimized VM instances in Amazon EC2, US East (Ohio).

Name	vCPU	ECU	RAM	Storage	Prices
r3.large	2	6.5	15GiB	1×32GB SSD	$0.166/hr
r3.xlarge	4	13	30.5GiB	1×80GB SSD	$0.333/hr
r3.2xlarge	8	26	61GiB	1×160GB SSD	$0.665/hr
r3.4xlarge	16	52	122GiB	1×320GB SSD	$1.33/hr
r3.8xlarge	32	104	244GiB	2×320GB SSD	$2.66/hr

where $p_{vt_i} = (c, f, r, d)$ denotes the computing capability of the VM of type vt_i, and C_{vt_i} denotes the financial cost for using this type of VM per time unit.

Figure 5(b) shows one VM instance of type vt_i denoted by v_y and two VM instances of type vt_j denoted by v_q and v_x in an example system model. Each communication link (such as $e_{x,y}$ from v_x to v_y) is directed and associated with bandwidth ($b_{x,y}$) and minimum link delay ($d_{x,y}$).

5.3 Performance Model

As shown in Figure 5(c), we focus on microservice-based application scheduling for *minimum end-to-end delay (MED)* under a user-specified *budget constraint (BC)* in a public cloud. We start with the generic notations of *execution time (ET)* and *monetary cost (MC)* for independent functions, and extend them to our application model with inter-function dependencies to identify and quantify the execution time and monetary cost components in public clouds.

One key advantage of using microservices and containers is the convenience of fast replication and migration. Therefore, we consider replicated microservices running as different instances, each of which is allowed to implement the same function needed by different applications.

5.3.1 $ET(t)$ and $MC(t)$ of a Microservice

To calculate ET and MC of function f_i, one must specify a scheduling scheme that defines the mapping between functions and microservices. In this paper, we define the scheduling scheme $\mathcal{S}(t)$ as a function that maps each function f_i in an application to a microservice ms_j at a scheduling time step t:

$$\mathcal{S}(t): f_i \rightarrow ms_j, \forall f_i \in V_a, \exists ms_j \in MS(t), \tag{14}$$

where $MS(t)$ denotes the collection of candidate microservices at time point t.

Suppose that a specific mapping $\mathcal{S}_p(t): f_p \rightarrow ms_i$ in $\mathcal{S}$ maps a function f_p to a microservice ms_i, which is encapsulated into a container c_j deployed on a VM v_k of type vt_p. The execution time $ET_i(t)$ of ms_i starting at time point t can be calculated as:

$$ET_i(t) = IT_i(t) + PT_i(t) + RT_i(t), \tag{15}$$

where $IT_i(t)$ denotes the initialization time of ms_i, $PT_i(t)$ denotes the overhead for business processing in ms_i and can be calculated according to Eq. (1) in Section 4, and $RT_i(t)$ denotes the time ms_i takes to transfer its output data to all of its succeeding microservices:

$$RT_i(t) = \sum_{\forall e_{i,j} \in E_a} RT_{i,j}(t) = \sum_{\forall e_{i,j} \in E_a} \left(\frac{w_{i,j}}{b_{i,j}} + d_{i,j}\right). \tag{16}$$

Note that the initialization time $IT_i(t)$ involves three different cases:

$$IT_i(t) = \begin{cases} 0, & ms_i \in MS(t) \quad (17a) \\ IT_{c_j}(t), & \text{create } c_j \text{ on } v_k \quad (17b) \\ IT_{v_k}(t) + IT_{c_j}(t), & \text{create } v_k \text{ and } c_j \quad (17c) \end{cases}$$

In Case 1 denoted by Eq. (17a), if there exists a microservice $ms_i \in MS(t)$ that can implement function f_p at t, no initialization is needed. Figure 5 shows such a case with function f_i and microservice ms_i.

In Case 2 denoted by Eq. (17b), if there is no suitable microservice to implement function f_i at t, a new microservice ms_i must be first created, then encapsulated into a container c_j, and finally deployed on an existing VM v_k. Therefore, the initialization of ms_i involves the creation of a new container c_j. For example, since there is no microservice available to implement function f_k in Figure 5 at t, a new microservice ms_k needs to be created, encapsulated into a container, and deployed on an existing VM v_q to implement f_k. The newly created microservice ms_k is then added into $MS(t)$.

In Case 3 denoted by Eq. (17c), if no microservice is available to implement function f_i and none of the existing VMs is suitable to execute microservice ms_i, we must first create a new VM v_k, then create container c_j on v_k to encapsulate microservice ms_i, and finally execute ms_i to implement f_i. Hence, the initialization of ms_i involves the creation of VM v_k and container c_j at t. For example, to implement function f_m in Figure 5, we first create a new VM v_n, then create a new container c_m on it, and finally launch a new microservice ms_m in c_m. The newly created microservice ms_m is added into $MS(t)$.

The monetary cost $MC_i(t)$ of executing ms_i at time point t is thus calculated as follows:

$$MC_i(t) = \frac{C_{vt_p}}{N_k(t)} \cdot ET_i(t). \quad (18)$$

where C_{vt_p} denotes the financial cost for using the VM of type vt_p per time unit, $ET_i(t)$ denotes the total time for executing ms_i at t, and $N_k(t)$ denotes the number of concurrently running microservices at t. Eq. (18) indicates that the monetary cost on VM v_k is shared by the microservices that have been deployed on v_k during the execution time $ET_i(t)$.

5.3.2 $ET(t)$ and $MC(t)$ of an Application

To calculate the $ET(t)$ of an application, for microservice ms_i, we denote its earliest start time and earliest finish time as $est(ms_i)$ and $eft(ms_i)$, and denote its latest start time and latest finish time as $lst(ms_i)$ and $lft(ms_i)$, respectively. The slack time $slack(ms_i)$, defined as $lst(ms_i) - est(ms_i)$ or $lft(ms_i) - eft(ms_i)$, is the amount of time the execution of microservice ms_i can be delayed without affecting the end-to-end delay of the entire application. The *critical path* (CP) is the longest execution path in the application graph weighted with time cost, which consists of all the microservices with zero slack time.

Once a scheduling scheme $\mathcal{S}(t)$ is determined, the $ET(t)$ can be calculated as the total time incurred on the CP, and the $MC(t)$ is obtained by summating the execution cost of each microservice ms_i in its container on the corresponding VM. Note that the CP might change at each step when a microservice is rescheduled or migrated to a different VM instance.

TABLE 3
Notations and parameters for the MAS-BC problem

Parameters	Definitions
$G_a = (V_a, E_a)$	microservice-based application
m	the number of functions in the application
f_i	the i-th function in the application
$e_{i,j}$	dependency edge from function f_i to f_j
$w_{i,j}$	transferred data size of dependency edge $e_{i,j}$
$G_v = (V_v, E_v)$	public cloud environment
n	the number of VMs in the cloud
v_i	the i-th VM in the cloud
$l_{i,j}$	network link between VM v_i and v_j
$b_{i,j}$	bandwidth of link $l_{i,j}$
$d_{i,j}$	minimum link delay of link $l_{i,j}$
ms_i	the i-th microservice
c_i	the i-th container
p_{c_i}	the computing capability of c_i
k	the number of VM types in the cloud
vt_i	the i-th VM types in the cloud
p_{vt_i}	the computing capability of vt_i
C_{vt_i}	the financial cost for using vt_i per time unit
t	timestamp
$\mathcal{S}(t)$	a scheduling scheme that maps functions to microservices on t
$\mathcal{S}_i(t)$	a specific mapping for function f_i
$\mathcal{S}^i(t)$	the i-th scheduling scheme
$ET_i(t)$	the execution time for a microservice ms_i on t
$MC_i(t)$	the monetary cost for executing ms_i on t
$MS(t)$	the collection of already deployed microservices on t

5.4 Problem Formulation

Based on the analytical models constructed above, we formally define a Microservice-based Application Scheduling problem for Minimum End-to-end Delay (MED) under a user-specified Budget Constraint in a public cloud, referred to as *MAS-BC*:

Definition 1 (MAS-BC). *Given a DAG-structured, microservice-based application $G_a(V_a, E_a)$, a fully-connected computing environment $G_v(V_v, E_v)$, a collection of existing microservices $MS(t)$, a set of available VM types $VT = \{vt_1, vt_2, \cdots, vt_k\}$, and a financial budget B, we wish to find a schedule $\mathcal{S}(t) : f_i \rightarrow ms_j, \forall f_i \in V_a, \exists ms_j \in MS(t)$, such that the MED of the mapped application is achieved at time point t:*

$$MED(t) = \min_{all\ possible\ \mathcal{S}(t)} \Big(\sum_{all\ f_i \in CP} ET_i(t) \Big), \quad (19)$$

subject to the budget constraint:

$$MC(t) = \sum_{all\ f_i \in V_a} MC_i(t) \leq B. \quad (20)$$

For convenience of reference, we summarize and tabulate the notations and parameters used in the definition of MAS-BC in Table 3.

5.5 Complexity Analysis

Previous studies have shown that a special case of MAS-BC, where the workflow only consists of a set of independent modules with zero data transfer time between any pair, is essentially the *multiple-choice knapsack problem* (MCKP) [57], which is known to be NP-complete. Since a special case is NP-complete, the original MAS-BC problem is also NP-complete. The NP-completeness proof by restriction is established in [58].

The above complexity analysis of MAS-BC rules out the existence of any polynomial-time optimal solution unless $P = NP$.

Therefore, we shall focus on the design of a heuristic approach to this optimization problem.

6 Greedy Recursive Critical Path Algorithm

To solve MAS-BC, we propose a heuristic approach using a Greedy Recursive Critical Path algorithm, referred to as GRCP, as shown in Alg. 1. GRCP recursively selects and maps the functions both on the critical path and on the non-critical paths using a greedy procedure in Alg. 2 and Alg. 3, respectively. It then refines the generated mapping scheme in Alg. 4 by minimizing the performance degradation if the budget constraint is violated. The time complexity of GRCP is $O(m^2+m\cdot n)$, where m is the number of functions in the given application and n is the number of available VMs. It is worth pointing out that GRCP sets a performance threshold β for each VM: if the current workload of a VM has reached this threshold, no more microservices are allowed to be deployed on this VM.

Algorithm 1 ***GreedyRPC(*** $G_a, \&MS(t), G_v, VT, B, t$***)***

Calculate execution time for each function in G_a based on $\mathcal{S}^0(t)$ assuming unlimited budget and resource homogeneity;
$P_1 \leftarrow$ *FindCriticalPath*$(G_a, \mathcal{S}^0(t))$;
$MED_1 \leftarrow \sum_{all\ f_i \in P_1} ET_{\mathcal{S}_i^0(t)}(t)$;
$i \leftarrow 1$;
while $|MED_i - MED_{i-1}| \leq \epsilon$ **do**
  $\mathcal{S}^i(t) \leftarrow$ *MapCriticalPath*$(P_i, \&MS(t), G_v, VT, t)$;
  MapNonCriticalPath$(P_i, G_a, \&MS(t), G_v, VT, t, \&\mathcal{S}^i(t))$;
  $i \leftarrow i + 1$;
  Calculate new execution time for each function in G_a based on the current mapping scheme $\mathcal{S}^i(t)$;
  $P_i \leftarrow$ *FindCriticalPath*$(G_a, \mathcal{S}^i(t))$;
  $MED_i \leftarrow \sum_{all\ f_j \in P_i} ET_{\mathcal{S}_j^i(t)}(t)$;
  if $MED_i < MED$ **then**
    $MED \leftarrow MED_i$;
    $\mathcal{S}(t) \leftarrow \mathcal{S}^i(t)$;
  end if
end while
$MED \leftarrow$ *RefineMapping*$(G_a, \&MS(t), G_v, VT, B, t, \&\mathcal{S}(t))$;
return *MED*;

Algorithm 2 ***MapCriticalPath(*** $P, \&MS(t), G_v, VT, t$***)***

$\mathcal{S}(t) \leftarrow \varnothing$;
MapFunctionOnCP$(f_1, \&MS(t), G_v, VT, t, \&\mathcal{S}(t))$;
for $i = 2$ to N **do**
  MapFunctionOnCP$(f_i, \&MS(t), G_v, VT, t, \&\mathcal{S}(t))$;
end for
return $\mathcal{S}(t)$;

Function *MapFunctionOnCP*$(f_i, \&MS(t), G_v, VT, t, \&\mathcal{S}(t))$
$T1[i], T2[i], ET1, ET2 \leftarrow MaxValue$;
for all $ms_j \in MS(t)$ **do**
  if ms_j can implement f_i **then**
    if $i = 1$ **then**
      $T1[j] \leftarrow ET_j(t)$;
    else
      $T1[j] \leftarrow ET_j(t) + RT_{S_{i-1}(t),j}(t)$;
    end if
  end if
end for
for all $v_k \in G_v$ **do**
  if $(p_{v_k} + p_{c_k}) < \beta \cdot p_{v_k}$ **then**
    Create a new microservice ms_k to implement f_i on v_k;
    if $i = 1$ **then**
      $T2[k] \leftarrow ET_k(t)$;
    else
      $T2[k] \leftarrow ET_k(t) + RT_{S_{i-1}(t),k}(t)$;
    end if
  else
    $T2[k] \leftarrow MaxValue$;
  end if
end for
$ET1 \leftarrow min(T1[j])$;
$ET2 \leftarrow min(T2[k])$;
if $ET1 \leq ET2$ **then**
  $\mathcal{S}(t) \leftarrow \mathcal{S}(t) \bigcup \{f_i \rightarrow ms_j\}$;
else
  $\mathcal{S}(t) \leftarrow \mathcal{S}(t) \bigcup \{f_i \rightarrow ms_k\}$;
  $MS(t) \leftarrow MS(t) \bigcup ms_k$;
end if

In Algorithm 1, the initial execution time for each function is determined first by assuming unlimited budget (i.e., for each function, we select a candidate microservice with the best performance) and resource homogeneity (i.e., the underlying VMs have the same computing capability and network bandwidth). Under such assumptions, GRCP only needs to consider G_a for execution time calculation. It then computes the CP P_1 with these initial execution times using a procedure defined in *FindCriticalPath*(), which employs a well-known polynomial Longest Path (LP) algorithm since the CP is essentially the longest execution path in a DAG in terms of MED.

After computing P_1, GRCP removes the assumption of computing and resource homogeneity and adapts the optimal pipeline mapping algorithm based on a greedy strategy to the CP mapping problem, as shown in Alg. 2. For each function f_i on the CP, this greedy-based algorithm needs to compare the cost of selecting an existing candidate microservice in $MS(t)$ with the cost of creating a new microservice and VM.

GRCP adopts a recursive priority-based greedy algorithm [59] to schedule non-critical functions that are not located on the CP, as shown in Alg. 3. It first inserts all critical functions into a queue Q. At step i, a function f_i is dequeued from Q and its unmapped immediate downstream functions are sorted in a decreasing order, denoted as $SIDM(f_i)$, according to their computation and communication requirements. We can either map these unmapped functions $f_j \in SIDM(f_i)$ to the existing microservices in $MS(t)$ with the minimum performance cost, or create new VMs and microservices to achieve the minimum time cost for data transfer and microservice execution. Finally, each function f_j is inserted at the end of Q. The above procedure is recursively performed until Q is empty so that all functions in G_a are assigned to $MS(t)$ and deployed on VMs in G_v.

If the monetary cost of the mapping scheme generated from Algs. 2 and 3 exceeds the budget, we need to refine such mapping scheme by reallocating the microservices on the CP to further reduce the MC of the application, as shown in Alg. 4. In this process, the algorithm considers the ratio of the time and cost difference [27] and either maps each function on the CP to a different microservice or creates a new microservice on a different VM, with an attempt to minimize the impact of cost increase on

Algorithm 3 ***MapNonCriticalPath***$(P, G_a, \&MS(t), G_v, VT, t, \&\mathcal{S})$

Notations:
$SIDM(f_i)$: a set of unmapped immediate downstream functions of f_i sorted in a decreasing order;

$T1[i], T2[i], ET1, ET2 \leftarrow MaxValue$;
Insert all functions on P into a queue Q;
$i = 1$;
while $Q \neq \emptyset$ **do**
  $f_i = pop(Q)$;
  for all $f_j \in SIDM(f_i)$ **do**
    for all $ms_k \in MS(t)$ **do**
      if ms_k can implement f_j **then**
        $T1[k] \leftarrow ET_k(t) + RT_{\mathcal{S}_i(t),j}(t)$;
      end if
    end for
    for all $v_l \in G_v$ **do**
      if $(p_{v_l} + p_{c_j}) < \beta \cdot p_{v_l}$ **then**
        Create ms_l to implement f_j on v_l;
        $T2[l] \leftarrow ET_l(t) + RT_{\mathcal{S}_i(t),l}(t)$;
      else
        $T2[l] \leftarrow MaxValue$;
      end if
    end for
    $ET1 \leftarrow min(T1[k])$;
    $ET2 \leftarrow min(T2[l])$;
    if $ET1 \leq ET2$ **then**
      $\mathcal{S}(t) \leftarrow \mathcal{S}(t) \bigcup \{f_j \rightarrow ms_k\}$;
    else
      $\mathcal{S}(t) \leftarrow \mathcal{S}(t) \bigcup \{f_j \rightarrow ms_l\}$;
      $MS(t) \leftarrow MS(t) \bigcup ms_l$;
    end if
  end for
  $i \leftarrow i + 1$;
end while

Algorithm 4 ***RefineMapping***$(G_a, \&MS(t), G_v, VT, B, t, \&\mathcal{S}(t))$

Calculate the monetary cost C for G_a based on the current scheduling scheme $\mathcal{S}(t)$;
if $C \leq B$ **then**
  return MED on $\mathcal{S}(t)$;
end if
while $C > B$ **do**
  $P \leftarrow FindCriticalPath(G_a, \mathcal{S}(t))$;
  for all $f_i \in P$ **do**
    for all $ms_j \in MS(t)$ **do**
      if ms_j can implement f_i **then**
        Calculate the ET increase $\Delta T_{i,j}$ and
        MC decrease $\Delta C_{i,j}$;
        $\Delta(ms_{i,j}) \leftarrow \frac{\Delta T_{i,j}}{\Delta C_{i,j}}$, where $\Delta C_{i,j} > 0$;
      else
        $\Delta(ms_{i,j}) \leftarrow MaxValue$;
      end if
    end for
    $\Delta(ms_i) \leftarrow min(\Delta(ms_{i,j}))$;
    $k \leftarrow$ index of the microservice having the $min(\Delta(ms_{i,j}))$;
    for all $v_j \in G_v$ **do**
      if $(p_{v_j} + p_{c_i}) < \beta \cdot p_{v_j}$ **then**
        Calculate the ET increase $\Delta T_{i,j}$ and
        MC decrease $\Delta C_{i,j}$;
        $\Delta(v_{i,j}) \leftarrow \frac{\Delta T_{i,j}}{\Delta C_{i,j}}$, where $\Delta C_{i,j} > 0$;
      else
        $\Delta(v_{i,j}) \leftarrow MaxValue$;
      end if
    end for
    $\Delta(v_i) \leftarrow min(\Delta(v_{i,j}))$;
    $l \leftarrow$ the index of microservice having the $min(\Delta(v_{i,j}))$;
  end for
  $\Delta(P) \leftarrow min(\Delta(ms_i), \Delta(v_i))$;
  $h \leftarrow$ the index of function having the $\Delta(P)$;
  if $\Delta(ms_i) \leq \Delta(v_i)$ **then**
    Map function f_h to ms_k and update $\mathcal{S}(t)$;
  else
    Map function f_h to ms_l and update $\mathcal{S}(t)$;
  end if
  Update MED and C based on $\mathcal{S}(t)$;
end while
return MED;

the end-to-end delay. For each function on the CP, the algorithm first calculates the minimum ratio of time increase over the cost decrease on a microservice or a VM, and then selects the function that achieves the global minimum ratio over the entire application graph for rescheduling.

Note that if multiple microservices are deployed on the same VM, the resource is shared among those microservices running concurrently, and the computing or transport time needs to be adjusted accordingly. We adopt the approach proposed in [59] to determine the number of concurrent microservices and the corresponding performance impact on the same VM.

7 Implementation and Experimental Results

We implement both the microservice performance modeling method and the GRCP algorithm proposed in this work. The software package and experimental results can be downloaded from the following public github repository: https://github.com/sselab/microservice.

7.1 Validation of Performance Models

7.1.1 Experimental Setup

To validate our performance model for microservice, we employ Acme Air[4] for the subject microservice-based application. Acme Air is an open-source benchmark that emulates transactional workloads for a fictitious airline's Web site, and has been widely used for performance evaluation of microservice-based applications [2].

As shown in Figure 6, there are four microservices in Acme Air system:

- The frontend service receives requests from users and communicates with the other three business microservices to perform authentication, flight search, booking, and cancelation.
- The authentication service (AU) performs user authentication and manages Web sessions.
- The flight-booking service (FB) processes flight search requests from clients and books flights.
- The customer service (CS) manages information associated with each client including booked flights and user information.

4. https://github.com/acmeair/acmeair

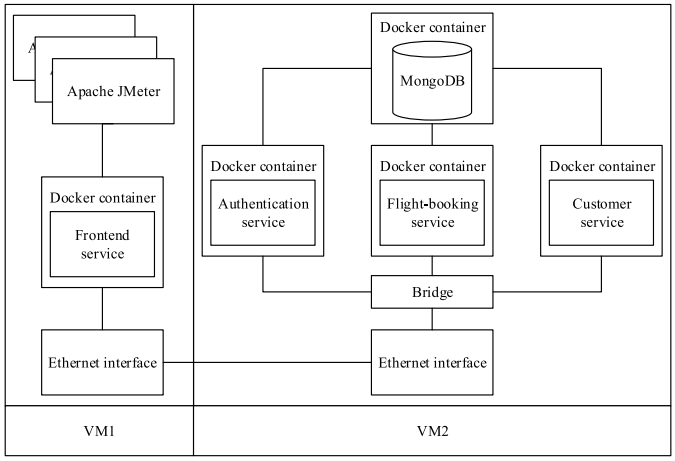

Fig. 6. The deployment of Acme Air system in our experiment.

TABLE 4
Specifications of benchmark and test machines.

Name	Processor	Cores	Frequency	RAM	Storage
Benchmark Machines (BM)					
BM1	Intel Xeon E5-2650 v2	6	1.7GHz	12GiB	60GB
BM1	Intel Xeon E5-2680 v3	10	2.6GHz	20GiB	100GB
Test Machines (TM)					
TM1	Intel Xeon E5-2650 v2	8	1.7GHz	16GiB	60GB
TM2	Intel Xeon E5-2680 v3	16	2.6GHz	32GiB	120GB
TM3	Intel Xeon E5-2683 v3	12	2.0GHz	24GiB	80GB

To validate our performance model, we use a set of physical machines with different hardware specifications to run the same Acme Air system, which is deployed on a single machine. The specifications of these benchmark machines (BM) and test machines (TM) are provided in Table 4.

We use Java 1.8 as the language runtime, Docker 1.9.1 as the running environment, MongoDB 3.2.0 as the database, and Apache JMeter 2.13 as the workload driver. As shown in Figure 6, these microservices are connected using the bridge configuration mode: the network interface in each container is connected to the virtual Ethernet bridge, which is then connected to the physical network of the host machine by using *iptables*. Based on this configuration, we create two VM instances on each physical machine and deploy these software modules according to the following rules: i) Apache JMeter and frontend services run on the first VM with 2 cores, 4GB RAM, and 10GB disk, and ii) business microservices and MongoDB run on the other VM and equally share the remaining resource. Note that we do not duplicate microservice and database instances to avoid introducing any possible performance interference.

We adopt locally weighted regression method [60] in our experiments. More specifically, the *independent variables* are the features listed in Table 1, and the predicted processing time is measured as the *dependent variable*. As shown in Figure 3, for each execution of microservice, we record four time points named t_1 through t_4 to calculate time intervals of the three phases, and the sizes of used cache and RAM at time point t_1. To account for the variations in the dependent variable caused by system randomness, we perform five repetitions for each combination of the independent variables, and calculate the average of these measurements.

TABLE 5
Regression errors in performance estimation.

MS	BM1			BM2		
	RMSE(ns)	NRMSE	R^2	RMSE(ns)	NRMSE	R^2
AU	2.3128e+9	5.72%	0.9931	1.7862e+9	4.46%	0.9972
FB	1.9765e+10	4.38%	0.9892	1.5342e+10	5.01%	0.9901
CS	1.6574e+10	3.76%	0.9925	1.7213e+10	4.85%	0.9873

7.1.2 Experimental Results

We first run each of these three microservices on the benchmark machines with a series of combinations of different values of the features listed in Table 1. For each microservice, we perform regressions on the three estimated expressions, i.e. Eqs. 4, 6, and 11, and provide in Table 5 the results of the root mean square error (RMSE), normalized root mean square error (NRMSE), and R-square (R^2), which clearly indicate that a high level of goodness-of-fit is achieved in these regressions. With these fitted curves, we are able to provide a good estimate of the processing time of a given microservice-based application on these correlated features.

After obtaining the regression curves of the microservices, we employ these curves to predict the processing time of the microservices on the test machines in Table 4, which vary on CPU architecture, frequency, RAM size, and disk space. Also, we run three microservices on these test machines to measure the actual processing time. We plot the actual processing time (y-axis) and the predicted processing time (x-axis) of *FB* microservice on these three test machines in Figures 7 to 9, each of which consists of four sub-figures representing request caching (T_r), business processing (T_b), transaction processing (T_d), and total processing time, respectively. The straight line ($y = x$) in each figure provides a baseline of our regression results. Note that these predictions are calculated by the regression function obtained on three benchmark machines. Constructing the regression function incurs a low overhead as it is done only once for each microservice, and it takes a fixed number of computations to predict the processing time of each microservice on the test machines.

Table 6 tabulates the RMSE and NRMSE in the prediction of microservice processing time on the test machines. These results show that we are able to obtain an accurate prediction of microservice processing time on each test machine, which has a set of completely different hardware parameters from the benchmark machines.

7.2 Evaluation of Scheduling Algorithms

7.2.1 Simulation Settings

The proposed GRCP algorithm is implemented in Java and runs on a Windows 7 PC equipped with a 2-core 3.30GHz CPU and 8 GBytes memory. For performance comparison, we also implement two existing algorithms, namely, Greedy A* [61] and BHEFT [24], on the same computing platform. Greedy A* (GA*) explores the least-cost path of the search tree in the solution space to find the optimal plan, instead of searching all feasible paths. BHEFT is a bi-criteria heuristic based on the well-known DAG scheduling heuristic HEFT [25] to meet the specific requirements of budget-deadline constrained planning. GA* and BHEFT have a time complexity of $O(n^5)$ and $O(n^8)$, respectively, where n is the number of available VM instances. The main reason for choosing these two algorithms for comparison is because they are two well-known and general benchmarking algorithms for the evaluation of

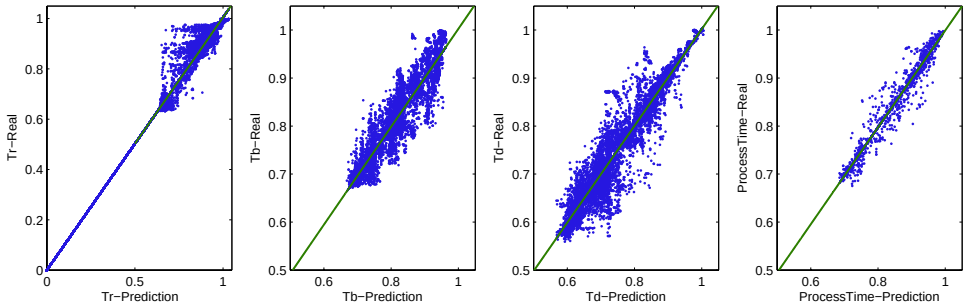

Fig. 7. Prediction of processing time for FB on TM1.

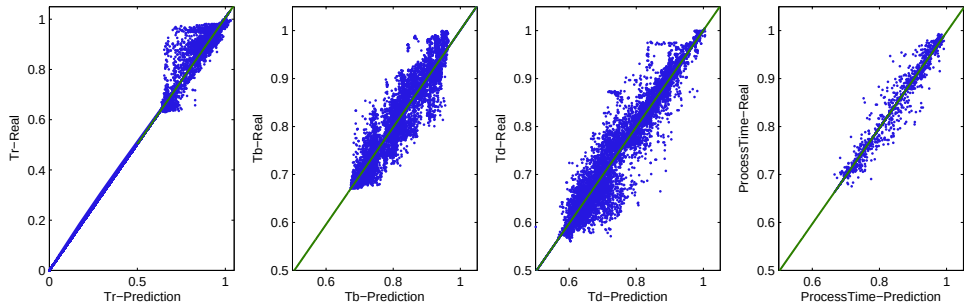

Fig. 8. Prediction of processing time for FB on TM2.

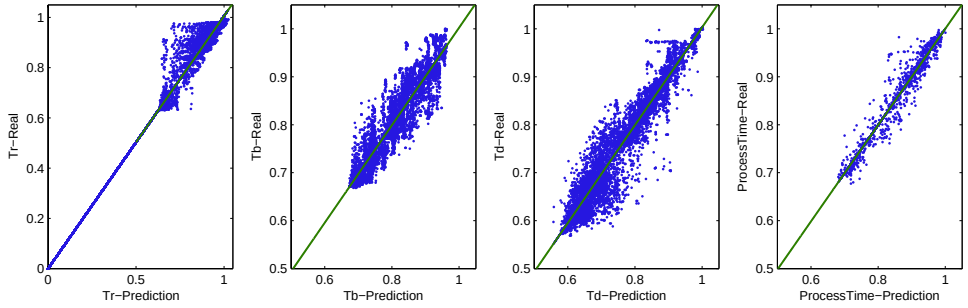

Fig. 9. Prediction of processing time for FB on TM3.

TABLE 6
Errors in performance prediction.

MS	TM1		TM2		TM3	
	RMSE	NRMSE	RMSE	NRMSE	RMSE	NRMSE
AU	1.5263e+9	12.63%	9.1223e+8	15.01%	8.3145e+8	18.19%
FB	7.4678e+9	8.82%	8.5218e+9	10.93%	7.1892e+9	13.75%
CS	1.1262e+10	10.91%	2.1452e+10	9.34%	1.2182e+10	8.99%

scheduling algorithms and have been widely used in many recent studies [62], [63].

To evaluate the algorithm performance, we consider different problem sizes from small to large scales. The problem size is defined as a three-tuple (m, n, k), where m is the number of microservices in an application, n is the maximum number of VMs in the cloud, and k is the number of available VM types. We use a synthetic workflow generator developed in [64] to generate simulated applications of different structures in a random manner.

We employ a pricing model for VMs according to real-world public cloud providers such as Amazon EC2. The price is a linear function of the number of processing units in the VM type. The financial cost of each VM type is priced according to the number of base processing units. When generating a given number of VM types, type vt_j has $2j$ base processing units, $j = 1, 2, \cdots, k$.

We compare our GRCP algorithm with GA* and BHEFT in terms of end-to-end delay under the same budget constraint. The MED performance improvement of GRCP over the other algorithms in comparison is defined as:

$$Imp(other) = \frac{MED_{other} - MED_{GRCP}}{MED_{other}} \times 100\%,$$

where MED_{other} is the MED achieved by the other algorithms in comparison, i.e., GA* and BHEFT, and MED_{GRCP} is the MED obtained by GRCP.

7.2.2 Simulation Results

A) Comparison with Optimal Solutions

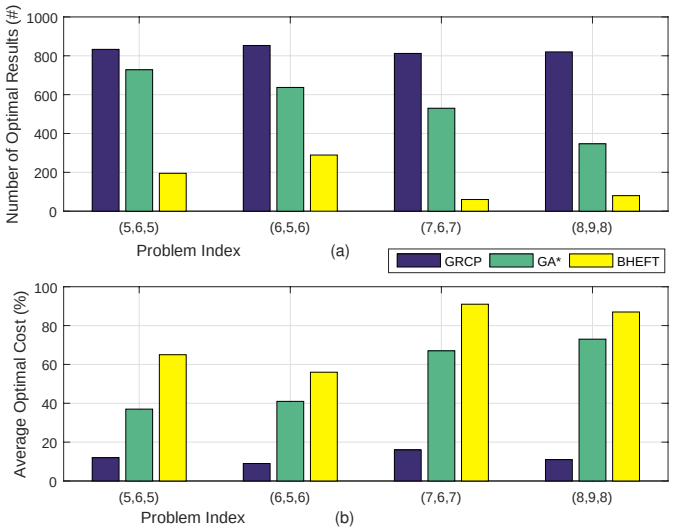

Fig. 10. The number of optimal results (a) and the average optimal costs (b) among 1,000 instances produced by GRCP, GA*, and BHEFT with different problem sizes.

We first compare the performance of GRCP with exhaustive search-based optimal solutions in small-scale problems with 5, 6, 7, and 8 microservices and available VM types. For each problem size, we randomly generate 50 problem instances with different application workloads and DAG topologies. For each problem instance, we run an adapted version of the three algorithms to generate the maximized and minimized cost, and then obtain the corresponding least cost C_{min} and maximum cost C_{max} by finding the minimum and maximum values from generated costs. We then iterate through 20 budget levels at an equal interval, run the three algorithms on these instances, and compare the MED results with the optimal ones computed by an exhaustive search approach. Figure 10(a) shows the number of optimal results among 1,000 instances (50 application instances × 20 budget levels) (upper) produced by GRCP, GA*, and BHEFT with different problem sizes. For those instances that do not provide optimal results, we calculate their average optimal costs, i.e. the average relative distance (%) between every generated result to the corresponding optimal result, and illustrate them in Figure 10(b) (lower). We observe that in a statistical sense, GRCP is more likely to achieve the optimality and the quality of results is also better, which indicates the efficacy of GRCP.

B) Comparison with GA* and BHEFT

We further consider 20 problem sizes from small to large scales, indexed from 1 to 20. In each problem size, we randomly generate 50 problem instances, and for each of them, we also run the algorithms by iterating through 20 budget levels. We provide in Table 7 the average MED improvement percentages together with standard deviations achieved by GRCP over GA* and BHEFT across all the 20 budget levels.

For each of the 20 budget levels from low to high values, indexed from 1 to 20, we run the algorithms by iterating through 20 problem sizes from small to large scales. We provide in Table 8 the average MED improvement percentages together with standard deviations achieved by GRCP over GA* and BHEFT across all the 20 problem sizes.

For a better illustration, we plot the overall performance improvement percentage of GRCP over GA* in Figure 11, where

TABLE 7
The average MED improvement percentages (Imp) of GRCP over GA* and BHEFT and the running times (RTs) of three algorithms with different problem sizes.

Index, *(m,n,k)*	Imp% (GA*)	Imp% (BHEFT)	RT (ms) (GA*)	RT (ms) (BHEFT)	RT (ms) (GRCP)
1, (5, 5, 5)	6.22	22.82	22	3	5
2, (10, 8, 6)	5.73	12.53	14	4	2
3, (15, 10, 7)	5.01	19.73	37	3	2
4, (20, 18, 8)	8.24	17.44	87	5	2
5, (25, 25, 9)	5.41	8.65	80	7	2
6, (30, 32, 10)	6.88	12.37	1792	7	3
7, (35, 30, 11)	5.93	2.55	9133	11	3
8, (40, 25, 12)	7.81	6.19	17355	15	4
9, (45, 38, 13)	10.42	6.39	96923	21	4
10, (50,43,14)	9.53	2.93	221170	28	5
11, (55, 40, 15)	8.96	4.54	694092	39	7
12, (60, 52, 16)	9.27	4.92	333756	51	8
13, (65, 50, 17)	11.31	5.59	695062	64	9
14, (70, 61, 18)	12.09	4.78	1197795	79	9
15, (75,70,19)	14.78	4.81	1276250	95	10
16, (80, 68, 20)	16.19	7.61	1362262	114	11
17, (85, 72, 21)	20.68	9.65	770605	132	13
18, (90, 81, 22)	13.91	4.92	1475434	150	18
19, (95, 76, 23)	15.30	4.43	995372	180	22
20, (100, 92, 24)	14.31	7.06	1653262	197	26

TABLE 8
The average MED improvement percentages (Imp) of GRCP over GA* and BHEFT and the running times (RTs) of three algorithms with different budget levels.

Budget level	Imp% (GA*)	Imp% (BHEFT)	RT (ms) (GA*)	RT (ms) (BHEFT)	RT (ms) (GRCP)
1	8.37	4.06	15	4	5
2	11.90	5.22	15	4	3
3	13.57	4.07	31	5	3
4	15.12	4.60	69	5	3
5	2.75	6.39	66	6	4
6	9.56	5.56	1375	8	11
7	10.76	5.06	6994	9	13
8	11.88	4.41	13263	14	19
9	19.00	3.89	85077	20	30
10	21.18	3.85	169400	29	59
11	18.74	3.05	499399	39	94
12	16.68	6.41	307964	50	105
13	8.61	4.18	645925	64	120
14	19.24	6.11	1616266	81	230
15	13.28	5.92	1332081	94	266
16	13.83	5.16	1424321	112	244
17	7.88	3.55	1431567	120	282
18	15.33	6.85	1435785	143	668
19	14.02	5.97	1520061	158	724
20	13.85	6.70	1473776	183	847

x axis denotes the budget increment across 20 levels and y axis denotes the index of 20 problem sizes from small to large scales. Each point (x, y, imp) in the 3D plot represents the average performance improvement across all 50 problem instances of the same problem size under the same budget level. These performance results show that GRCP achieves an average of 20% performance improvement over GA*. Similarly, we plot the overall performance improvement percentage of GRCP over BHEFT in Figure 12. These performance results show that GRCP achieves an average of 35% performance improvement over BHEFT. Such performance improvements are considered significant for the execution of microservice-based applications.

8 Conclusions

The microservice architectural style is gaining increasing popularity in both academia and industry. Performance modeling and

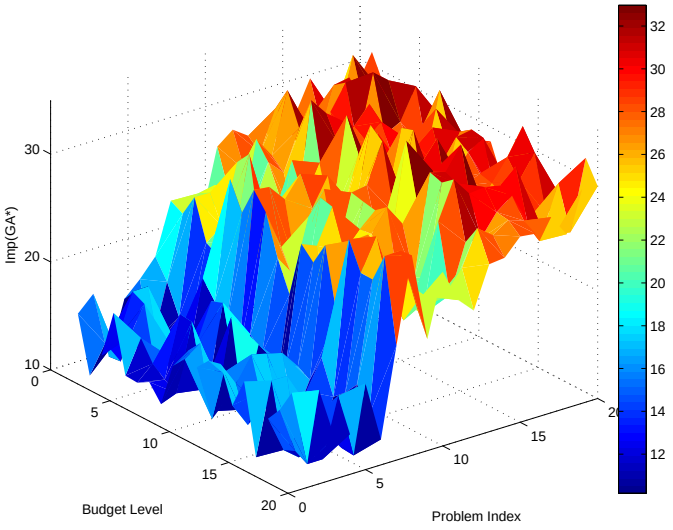

Fig. 11. The overall performance improvement percentages (%) of GRCP over GA* with varying budget levels and problem sizes.

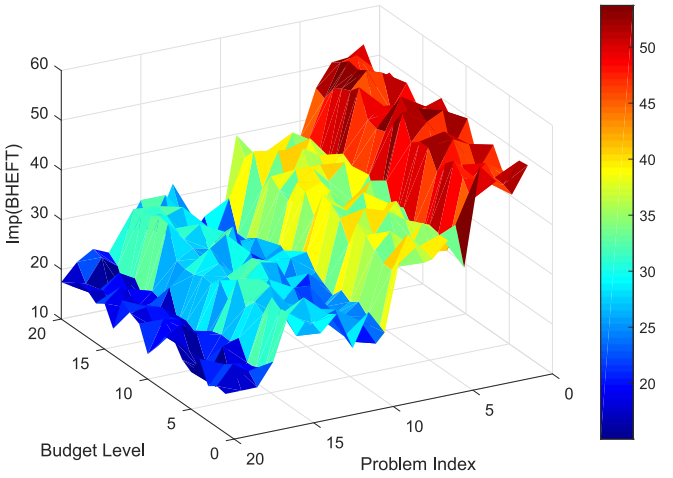

Fig. 12. The overall performance improvement percentages (%) of GRCP over BHEFT with varying budget levels and problem sizes.

task scheduling are two important issues for microservice-based applications. In this paper, we designed a performance modeling and prediction method for microservice-based applications, formulated the MAS-BC problem, and proposed a heuristic solution. Our work on performance modeling and optimization has been validated and evaluated through both experimental and simulation results, and has great potential to improve the performance of a wide range of real-life microservice-based applications deployed in public clouds.

Our future work includes the development of a practical benchmark for microservice applications based on our performance models, and the design of scheduling algorithms in computing environments with multiple concurrently running microservice-based applications.

Acknowledgments

This work is supported by the National Natural Science Foundation of China under Grant No. 61202040 with XiDian University and U.S. National Science Foundation under Grant No. CNS-1828123 with New Jersey Institute of Technology. This work is also supported by the Fundamental Research Funds for the Central Universities (Grant No. JB171005).

References

[1] M. Fowler and J. Lewis, "Microservices," *ThoughtWorks. http://martinfowler.com/articles/microservices.html [last accessed on February 11, 2017]*, 2014.

[2] T. Ueda, T. Nakaike, and M. Ohara, "Workload characterization for microservices," in *Workload Characterization (IISWC), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 1–10.

[3] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, and M. Villari, "Open issues in scheduling microservices in the cloud," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 81–88, 2016.

[4] C. Pahl and P. Jamshidi, "Microservices: A systematic mapping study," in *Proceedings of the 6th International Conference on Cloud Computing and Services Science*, 2016, pp. 137–146.

[5] Pattern: Service instance per container: https://microservices.io/patterns/deployment/service-per-container.html.

[6] Nagios, "Nagios it infrastructure monitoring," *https://www.nagios.org/ [last accessed on February 11, 2017]*, 2017.

[7] Ganglia, "Ganglia monitoring system," *http://ganglia.info/ [last accessed on February 11, 2017]*, 2017.

[8] Heapster, "Compute resource usage analysis and monitoring of container clusters," *https://github.com/kubernetes/heapster [last accessed on February 11, 2017]*, 2017.

[9] Kubernetes, "Compute resource usage analysis and monitoring of container clusters," *https://kubernetes.io/ [last accessed on February 11, 2017]*, 2017.

[10] Mesos, "Apache mesos," *http://mesos.apache.org/ [last accessed on February 11, 2017]*, 2017.

[11] Q. Wu and V. V. Datla, "On performance modeling and prediction in support of scientific workflow optimization," in *2011 IEEE World Congress on Services*. IEEE, 2011, pp. 161–168.

[12] S. Chen, Y. Liu, I. Gorton, and A. Liu, "Performance prediction of component-based applications," *Journal of Systems and Software*, vol. 74, no. 1, pp. 35–43, 2005.

[13] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 74–89, 2003.

[14] S. Jang, X. Wu, V. Taylor, G. Mehta, K. Vahi, and E. Deelman, "Using performance prediction to allocate grid resources," *Texas A&M University, College Station, TX, GriPhyN Technical Report*, vol. 25, 2004.

[15] S. M. Sadjadi, S. Shimizu, J. Figueroa, R. Rangaswami, J. Delgado, H. Duran, and X. J. Collazo-Mojica, "A modeling approach for estimating execution time of long-running scientific applications," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–8.

[16] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. De Supinski, and M. Schulz, "A regression-based approach to scalability prediction," in *Proceedings of the 22nd annual international conference on Supercomputing*. ACM, 2008, pp. 368–377.

[17] S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson, "Measuring empirical computational complexity," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2007, pp. 395–404.

[18] E. Thereska, B. Doebel, A. X. Zheng, and P. Nobel, "Practical performance models for complex, popular applications," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 1. ACM, 2010, pp. 1–12.

[19] S. Abrishami and M. Naghibzadeh, "Deadline-constrained workflow scheduling in software as a service cloud," *Scientia Iranica*, vol. 19, no. 3, pp. 680–689, 2012.

[20] M. Mao and M. Humphrey, "Auto-scaling to minimize cost and meet application deadlines in cloud workflows," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*. IEEE, 2011, pp. 1–12.

[21] M. Mao, J. Li, and M. Humphrey, "Cloud auto-scaling with deadline and budget constraints," in *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*. IEEE, 2010, pp. 41–48.

[22] C. Hoffa, G. Mehta, T. Freeman, E. Deelman, K. Keahey, B. Berriman, and J. Good, "On the use of cloud computing for scientific workflows," in *eScience, 2008. eScience'08. IEEE Fourth International Conference on*. IEEE, 2008, pp. 640–645.

[23] L. Zeng, B. Veeravalli, and X. Li, "Scalestar: Budget conscious scheduling precedence-constrained many-task workflow applications in cloud," in *Advanced Information Networking and Applications (AINA), 2012 IEEE 26th International Conference on*. IEEE, 2012, pp. 534–541.

[24] W. Zheng and R. Sakellariou, "Budget-deadline constrained workflow planning for admission control," *Journal of grid computing*, vol. 11, no. 4, pp. 633–651, 2013.

[25] H. Topcuoglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE transactions on parallel and distributed systems*, vol. 13, no. 3, pp. 260–274, 2002.

[26] H. Arabnejad and J. G. Barbosa, "A budget constrained scheduling algorithm for workflow applications," *Journal of grid computing*, vol. 12, no. 4, pp. 665–679, 2014.

[27] C. Q. Wu, X. Lin, D. Yu, W. Xu, and L. Li, "End-to-end delay minimization for scientific workflows in clouds under budget constraint," *IEEE Transactions on Cloud Computing*, vol. 3, no. 2, pp. 169–181, 2015.

[28] J. Lewis and M. Fowler, "Microservices," *http://martinfowler.com/articles/microservices.html, last accessed: 2016-12-1*, 2014.

[29] H. Kurhinen, "Developing microservice-based distributed workflow engine," *University of Applied Sciences*, 2014.

[30] S. Alpers, C. Becker, A. Oberweis, and T. Schuster, "Microservice based tool support for business process modelling," in *2015 IEEE 19th International Enterprise Distributed Object Computing Workshop*. IEEE, 2015, pp. 71–78.

[31] D. Guo, W. Wang, G. Zeng, and Z. Wei, "Microservices architecture based cloudware deployment platform for service computing," in *2016 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. IEEE, 2016, pp. 358–363.

[32] H. Kang, M. Le, and S. Tao, "Container and microservice driven design for cloud infrastructure devops," in *Cloud Engineering (IC2E), 2016 IEEE International Conference on*. IEEE, 2016, pp. 202–211.

[33] T. Vresk and I. Čavrak, "Architecture of an interoperable iot platform based on microservices," in *39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, 2016, pp. 1196–1201.

[34] D. D'Agostino, L. Roverelli, G. Zereik, A. De Luca, R. Salvaterra, A. Belfiore, G. Lisini, G. Novara, and A. Tiengo, "A microservice-based portal for x-ray transient and variable sources," PeerJ Preprints, Tech. Rep., 2016.

[35] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, "Performance evaluation of microservices architectures using containers," in *Network Computing and Applications (NCA), 2015 IEEE 14th International Symposium on*. IEEE, 2015, pp. 27–34.

[36] W. Hasselbring, "Microservices for scalability: Keynote talk abstract," in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. ACM, 2016, pp. 133–134.

[37] M. Villamizar, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, M. Lang *et al.*, "Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures," in *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*. IEEE, 2016, pp. 179–182.

[38] H. Knoche, "Sustaining runtime performance while incrementally modernizing transactional monolithic software towards microservices," in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. ACM, 2016, pp. 121–124.

[39] R. Heinrich, A. van Hoorn, H. Knoche, F. Li, L. E. Lwakatare, C. Pahl, S. Schulte, and J. Wettinger, "Performance engineering for microservices: research challenges and directions," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. ACM, 2017, pp. 223–226.

[40] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao, "Benchmarking microservice systems for software engineering research," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 2018, pp. 323–324.

[41] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *Cloud Engineering (IC2E), 2018 IEEE International Conference on*. IEEE, 2018, pp. 159–169.

[42] M. Gotin, F. Lösch, R. Heinrich, and R. Reussner, "Investigating performance metrics for scaling microservices in cloudiot-environments," in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM, 2018, pp. 157–167.

[43] T. Engel, M. Langermeier, B. Bauer, and A. Hofmann, "Evaluation of microservice architectures: A metric and tool-based approach," in *International Conference on Advanced Information Systems Engineering*. Springer, 2018, pp. 74–89.

[44] B. Saman, "Monitoring and analysis of microservices performance," *Journal of Computer Science and Control Systems*, vol. 10, no. 1, p. 19, 2017.

[45] M. Gribaudo, M. Iacono, and D. Manini, "Performance evaluation of massively distributed microservices based applications," in *31st European Conference on Modelling and Simulation, ECMS 2017*. European Council for Modelling and Simulation, 2017, pp. 598–604.

[46] N. Ford, "Polyglot programming," *http://memeagora.blogspot.com/2006/12/polyglot-programming.html, last accessed: 2016-12-1*, 2006.

[47] C. Richardson, "Pattern: Microservice architecture," *Microservice Architecture. http://microservices.io/patterns/microservices.html [last accessed on February 11, 2017]*, 2017.

[48] OASIS, "Advanced message queuing protocol," *Advanced Message Queuing Protocol. https://www.amqp.org/ [last accessed on February 11, 2017]*, 2017.

[49] A. Gupta, "Microservice design patterns," *Microservice Design Patterns. http://blog.arungupta.me/microservice-design-patterns/ [last accessed on February 11, 2017]*, 2017.

[50] C. Richardson, "Pattern: Api gateway," *API Gateway. http://microservices.io/patterns/apigateway.html [last accessed on February 11, 2017]*, 2017.

[51] ——, "A pattern language for microservices," *A pattern language for microservices. http://microservices.io/patterns/index.html [last accessed on February 11, 2017]*, 2014.

[52] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Software*, vol. 35, no. 3, pp. 24–35, 2018.

[53] Q. Wu, N. S. Rao, and S. S. Iyengar, "On transport daemons for small collaborative applications over wide-area networks," in *PCCC 2005. 24th IEEE International Performance, Computing, and Communications Conference, 2005.* IEEE, 2005, pp. 159–166.

[54] N. S. Rao, "Vector space methods for sensor fusion problems," *Optical Engineering*, vol. 37, no. 2, pp. 499–504, 1998.

[55] J. Duggan, U. Cetintemel, O. Papaemmanouil, and E. Upfal, "Performance prediction for concurrent database workloads," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, 2011, pp. 337–348.

[56] P. Brucker, Y. N. Sotskov, and F. Werner, "Complexity of shop-scheduling problems with fixed number of jobs: a survey," *Mathematical Methods of Operations Research*, vol. 65, no. 3, pp. 461–481, 2007.

[57] S. Martello and P. Toth, *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.

[58] M. R. Gary and D. S. Johnson, "Computers and intractability: A guide to the theory of np-completeness," 1979.

[59] Q. Wu and Y. Gu, "Supporting distributed application workflows in heterogeneous computing environments," in *Parallel and Distributed Systems, 2008. ICPADS'08. 14th IEEE International Conference on*. IEEE, 2008, pp. 3–10.

[60] W. S. Cleveland, "Robust locally weighted regression and smoothing scatterplots," *Journal of the American statistical association*, vol. 74, no. 368, pp. 829–836, 1979.

[61] A. Sekhar, B. Manoj, and C. S. R. Murthy, "A state-space search approach for optimizing reliability and cost of execution in distributed sensor networks," in *International Workshop on Distributed Computing*. Springer, 2005, pp. 63–74.

[62] H. Chen, X. Zhu, D. Qiu, L. Liu, and Z. Du, "Scheduling for workflows with security-sensitive intermediate data by selective tasks duplication in clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 9, pp. 2674–2688, 2017.

[63] X. Li, L. Qian, and R. Ruiz, "Cloud workflow scheduling with deadlines and time slot availability," *IEEE Transactions on Services Computing*, vol. 11, no. 2, pp. 329–340, 2018.

[64] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, "Characterizing and profiling scientific workflows," *Future Generation Computer Systems*, vol. 29, no. 3, pp. 682–692, 2013.

Liang Bao received the PhD degree in computer science from XiDian University, P.R. China, in 2010. He is currently an associate professor with the School of Software, XiDian University. His research interests include software architecture, cloud computing and big data. He is a member of the IEEE.

Chase Wu received the PhD degree in computer science from Louisiana State University in 2003. He was a research fellow in the Computer Science and Mathematics Division at Oak Ridge National Laboratory during 2003-2006 and an assistant and associate professor at University of Memphis during 2006-2015. He is currently an associate professor with the Department of Computer Science at New Jersey Institute of Technology. His research interests include big data, distributed computing, computer networks, scientific visualization, and cyber security. He is a member of the IEEE.