Modeling and Optimization of Performance and Cost of Serverless Applications

Article *in* IEEE Transactions on Parallel and Distributed Systems · October 2020

DOI: 10.1109/TPDS.2020.3028841

2 authors:

Changyuan Lin
University of British Columbia
10 PUBLICATIONS **160** CITATIONS
SEE PROFILE

Hamzeh Khazaei
York University
95 PUBLICATIONS **2,136** CITATIONS
SEE PROFILE

Modeling and Optimization of Performance and Cost of Serverless Applications

Changyuan Lin, *Student Member, IEEE,* Hamzeh Khazaei, *Member, IEEE,*

Abstract—Function-as-a-Service (FaaS) and serverless applications have proliferated significantly in recent years because of their high scalability, ease of resource management, and pay-as-you-go pricing model. However, cloud users are facing practical problems when they migrate their applications to the serverless pattern, which are the lack of analytical performance and billing model and the trade-off between limited budget and the desired quality of service of serverless applications. In this paper, we fill this gap by proposing and answering two research questions regarding the prediction and optimization of performance and cost of serverless applications. We propose a new construct to formally define a serverless application workflow, and then implement analytical models to predict the average end-to-end response time and the cost of the workflow. Consequently, we propose a heuristic algorithm named Probability Refined Critical Path Greedy algorithm (PRCP) with four greedy strategies to answer two fundamental optimization questions regarding the performance and the cost. We extensively evaluate the proposed models by conducting experimentation on AWS Lambda and Step Functions. Our analytical models can predict the performance and cost of serverless applications with more than 98% accuracy. The PRCP algorithms can achieve the optimal configurations of serverless applications with 97% accuracy on average.

Index Terms—Cloud Serverless Computing; Performance Modeling; Performance Optimization; Cost Modeling; Cost Optimization;

1 Introduction

Cloud computing has drawn extensive attention from both academia and industry over the past decade. A rapidly increasing number of applications and services are shifted to the cloud because of the high scalability, availability, pay-as-you-go billing model, and low overhead on infrastructure management [1], [2]. Meantime, many cloud service providers, such as Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure, have emerged to offer diverse cloud services and infrastructures in the cloud computing market. With the increased popularity in containerization, service-oriented and event-driven architecture, there is also a growing trend in the paradigm of cloud computing shifting from Infrastructure-as-a-Service (IaaS) to Container-as-a-Service (CaaS) and Platform-as-a-Service (PaaS), and then to Function-as-a-Service (FaaS) [3]. Based on lightweight virtualization solutions, specifically containers, and container-orchestration systems, like Kubernetes, many cloud providers have launched their FaaS platforms, such as AWS Lambda, Google Cloud Functions, and Azure Functions.

With the development of FaaS platforms and evolution of the cloud computing paradigm, the serverless computing paradigm, in which the application is abstracted as a group of functions hosted on FaaS platforms orchestrated in a workflow has emerged. Serverless computing brings another revolution to cloud computing by rendering the management of underlying infrastructure unnecessary, as FaaS platforms take over all the operational responsibilities such as function deployment, resource management, scaling, and monitoring. Developers can mainly focus on the business logic of functions, thus expediting the application development. The small-granularity billing and scaling in FaaS platforms can also decrease the cost substantially [4], [5].

Despite the fact that serverless solution is cost-effective and eases the resource management, there are pain points hindering its wide adoption by potential users, including the lack of performance and cost model [6], [7], [8], and the trade-off analysis between the performance and cost of serverless applications [8], [9]. Performance and cost modeling and optimization are non-trivial and necessary steps towards guaranteeing the service level agreement (SLA) of serverless applications in an economic manner, which is basically finding an acceptable trade-off between performance and cost. The difficulties of such modeling and optimization problems lie in the following aspects:

1) FaaS platforms introduce a new GB-second billing model depending on the memory size, function duration, and the number of invocations.
2) The workflow, which represents the orchestration of functions in serverless applications, can be very complex in terms of the number of functions, diverse structures, and the number of configuration combinations.
3) Classical methods used in the performance modeling of parallel computing and traditional software, such as Petri net and queuing theory, are not suitable for the serverless application because of its independently-operated, event-driven, closed-source, and infrastructure-agnostic architecture.

- *C. Lin is with the Department of Electrical and Computer Engineering, University of Alberta, AB, Canada. E-mail: changyua@ualberta.ca*
- *H. Khazaei is with the Department of Electrical Engineering and Computer Science, York University, Toronto, ON, Canada. E-mail: hkh@yorku.ca*

Manuscript received September, 2020

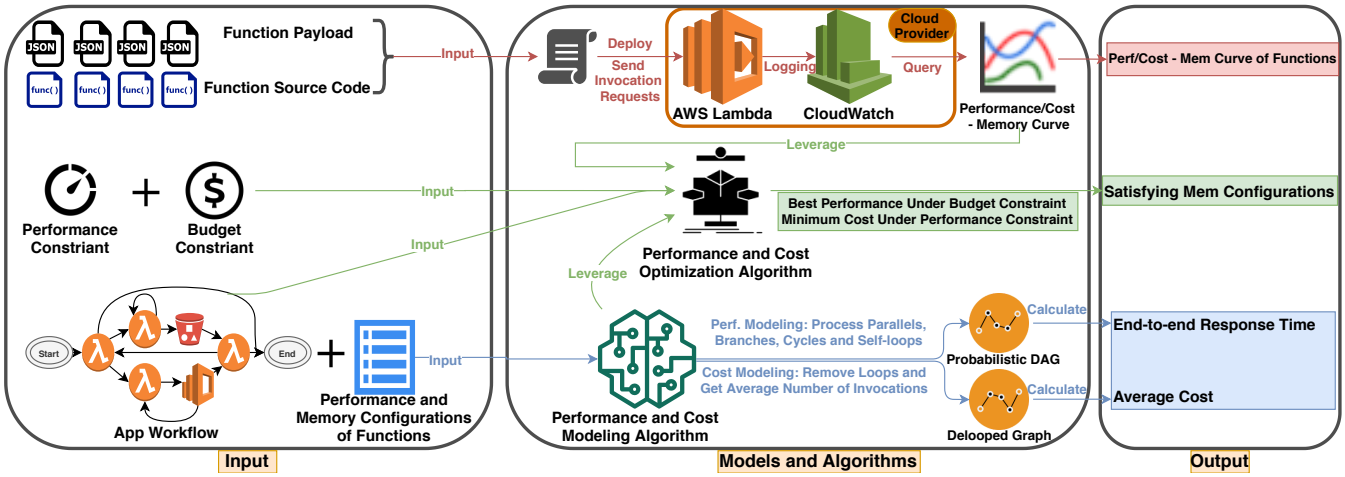

Fig. 1. Overview of the proposed approach for modeling and optimization of performance and cost of serverless applications. The three boxes from left to right illustrate the input, performance and cost models and optimization algorithms, and the corresponding output. The cloud provider part can be replaced with any FaaS platforms. We use AWS for experimental evaluation in this work. The proposed approach can accomplish three types of tasks (depicted by three different colors): (i) Deploy user-provided function source code and payload to FaaS platform, invoke functions and query the execution log, and give the performance-memory curve of functions as shown in Figure 2 as the output. (ii) Take the workflow orchestration of the serverless application and memory configuration and performance (response time) of functions as inputs, process inputs using the performance and cost models, and give the application end-to-end response time and the average cost under the given configuration as the output. (iii) For user-defined performance and cost constraints of a workflow, leverage the proposed performance cost models and optimization algorithms with the performance-memory curve of functions, and give the satisfying memory configuration that achieves the best performance under a budget constraint or the minimum cost under a performance constraint as the output.

Such a significant gap leads us to the following research questions:

- **RQ1:** Can we obtain the end-to-end response time and cost of a serverless application when orchestration and configuration of the application are given?
- **RQ2:** Can we calculate the best performance under a given budget or the minimum cost for a desired performance for a serverless application?

To answer these research questions, we propose new analytical models and a greedy algorithm with four strategies to respectively model and optimize the performance and cost of serverless applications. The overview of the proposed approach is illustrated in Figure 1. We evaluated the model and algorithm using serverless applications deployed on AWS composed of three types of generic functions representing CPU, network, and disk intensive workloads, with parallels, branches, cycles, and self-loops in their workflows. As verified by the experimental evaluation, our analytical model can precisely give the end-to-end response time of serverless applications with over 98% accuracy, and estimate the average cost with an accuracy of over 99%, independent of the complexity of their workflows. The greedy algorithm can effectively and efficiently resolve two types of optimization problems specified in RQ2 with over 97% accuracy.

The main contributions of this paper are three-fold:
(1) We propose a formal definition of the serverless application workflow to abstract sequence, parallelism, branch, cycle, and loop in the workflow and new features introduced by the serverless computing paradigm, such as the GB-second billing model and the orchestration of FaaS functions. (2) We propose, to the best of our knowledge, the first analytical models to accurately get the average end-to-end response time and cost of serverless applications with parallels, branches, cycles, and self-loops in their workflows. (3) We present a heuristic algorithm for optimizing performance and cost of serverless applications, which can give the optimal configurations of functions achieving the best performance for a given budget and the minimum cost for satisfying a given performance. All our algorithms, scripts, and experimental results are available in the artifact repository[1].

The rest of the paper is organized as follows: Section 2 briefly introduces the background including FaaS, serverless application, and serverless workflow. Section 3 presents the construction of the serverless workflow and performance and cost models, namely answering the RQ1. In Section 4, we propose performance and cost optimization problems in RQ2 and the PRCP algorithm to answer them. Section 5, presents the experimental evaluation of the aforementioned models and algorithm. In Section 6, we survey the latest related literature in the areas of performance modeling, workflow scheduling, and serverless architecture. Section 7 discusses and concludes our work.

2 Background

In this section, we give a brief introduction to FaaS, serverless application, and serverless workflow.

2.1 FaaS

In the Function as a Service (FaaS) paradigm, the function is a chunk of code that abstracts a part of an application implementing business logic. The notion of function in FaaS is similar to a function in functional programming or a method in object-oriented programming, responsible for handling one task required by the application statelessly. The source code of the function and its dependencies are packaged

1. https://github.com/pacslab/SLApp-PerfCost-MdlOpt

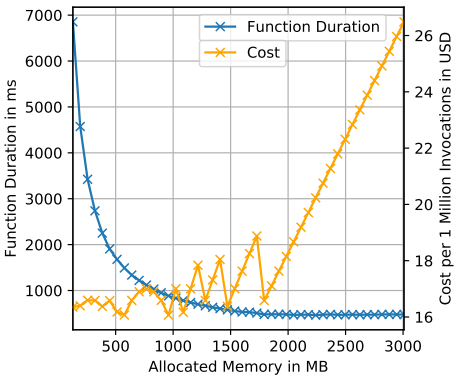

Fig. 2. The cost and performance with regard to the amount of allocated memory of a CPU-intensive function (hashing) deployed on AWS Lambda. The blue line is the performance-memory curve of the function.

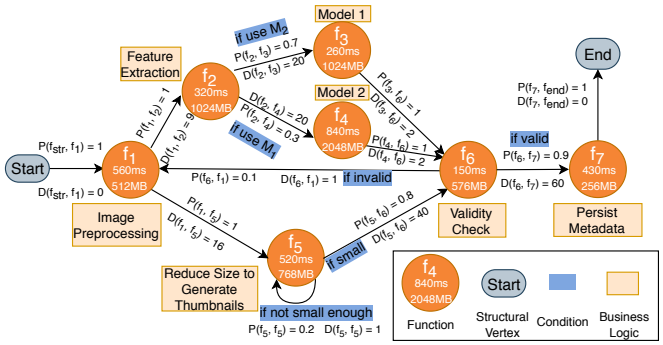

Fig. 3. Workflow of a serverless image classification application composed of seven FaaS functions. The two numbers on each function represent the response time and allocated memory, respectively. P and D are the transition probability function and the delay function.

together, and the function is running in an ephemeral isolated environment provided by lightweight virtualization solutions such as Docker containers, Unikernels [10], or even processes [11].

AWS launched a computing service called Lambda in 2014 [12], which could store, package, and deploy functions uploaded by users, handle events and requests of functions, monitor the resource usage, and autoscale function containers correspondingly. This was the first time that the function execution was offered as a cloud service, namely FaaS. Today, besides AWS Lambda, many FaaS platforms with similar functionalities are provided by public cloud service providers and open-source communities such as Google Cloud Functions, Azure Functions, and OpenFaaS.

As FaaS platforms take over operational responsibilities, besides uploading the source code of functions, users have limited control over resources on FaaS platforms. Taking AWS Lambda as an example, the amount of allocated memory during execution and the concurrency level are only options for tuning the performance of functions. The amount of allocated memory is between 128 MB and 3,008 MB in 64MB increments [13]. Previous researches have proven that computational power and network throughput are in proportion to the amount of allocated memory, and disk performance also increases with larger memory size due to less contention [14], [15]. By reserving and provisioning more instances to host functions, high concurrency level can decrease fluctuations in the function performance incurred by cold starts (container initialization provisioning delay if no warm instance is available) and reduce the number of throttles under very heavy request loads [16].

FaaS platforms also introduce a new GB-second billing model depending on the allocated memory size, function duration, and the number of invocations. For example, AWS Lambda charges $0.000016667 for every GB-second and $0.20 per 1M function invocations. The billed duration is the function duration rounded up to the nearest 100ms and metered at a granularity of 100ms [17]. Due to the rounding and billing granularity, the cost fluctuates erratically as the memory size changes, see our experiment results shown in Figure 2. The new billing model and such irregularities make the modeling and optimization problems non-trivial.

2.2 Serverless Application

The serverless application decouples its business logic into a group of serverless functions hosted on FaaS platforms and leverages necessary cloud services such as bucket storage, message queue, and pub/sub messaging service to build a stateless and event-driven software system [18], [19]. FaaS platforms and those cloud services shift most operational responsibilities to cloud service providers. Hence, the overhead of server management, monitoring, failover, and scaling is eliminated for serverless applications, like there are no servers to manage at all, namely serverless.

To complete the business logic of the application, interactions among decoupled functions are indispensable. In most cases, a coordinator is required to chain together components of the application, handle events among functions, and trigger functions in the correct order defined by the business logic. Typically, a message queue, a pub/sub messaging service, an event bus, or a workflow coordinator like AWS Step Functions Express Workflows act as such a coordinator [6], [18], [20]. Figure 3 illustrates an example of a serverless application composed of seven functions.

2.3 Serverless Workflow

In general, the serverless workflow is the orchestration of functions in the serverless application to implement the entire business logic. AWS defines that the serverless workflow describes a process as a series of individual functions and coordinates them [21]. As shown in many examples of serverless applications deployed in the production environment, there can be four types of structures in the serverless workflow, including parallel, branch, cycle, and self-loop [22], [23], [24]. Many research works have proven that workflows based on the directed acyclic graph (DAG) and Petri net are effective for performance and cost modeling of systems in parallel, distributed, and scientific computing [25], [26], [27]. However, none of them are favorable for the serverless paradigm, as cycles and loops are not allowed in DAGs, and cause the state explosion problem without efficient solutions in Petri nets. The disadvantages of Petri nets also lie in its high complexity and limited support of non-functional requirements in cloud computing [28]. Hence, performance and cost modeling of serverless applications requires a loose and concise definition of the serverless

workflow, which adapts to new features of the serverless computing paradigm.

3 Performance and Cost Modeling

In this section, we first construct the serverless workflow of a serverless application, and then propose two analytical models to predict the performance and cost, i.e., answering the RQ1.

3.1 Definition of the serverless workflow

The serverless workflow of a serverless application G_s is defined as a weighted directed graph:

$$G_s = (V, E, P, D, RT, RTTP, M, NI, C) \tag{1}$$

where

- V is a finite set of $|V|$ vertices $\{f_1, f_2, ..., f_n\}$, such that $n = |V|$, representing FaaS functions, integrated cloud services, or structural vertices;
- $E \subseteq V \times V$ is a finite set of directed edges. The directed edge from $f_u \in V$ to $f_v \in V$, denoted as $e_i = (f_u, f_v)$, represents the interaction between vertex f_u and f_v defined by the business logic;
- $P : V \times V \to [0, 1]$ is a transition probability function. $P(f_u, f_v)$ identifies the probability of invoking f_v after finishing the execution of f_u. A transition probability of 0 represents the corresponding edge does not exist;
- $D : V \times V \to [0, +\infty)$ is a delay function. $D(f_u, f_v)$ identifies the delay from f_u to f_v incurred by the interaction/coordination method;
- $RT : V \to [0, +\infty)$ is the response time of a function. $RT(f_u)$ is the response time of function f_u;
- $RTTP$: $V \to \wp([0, +\infty) \times [0, 1])$ is a function representing all possible values of the response time and corresponding probability of interim B-nodes, which are defined in Section 3.3.1 to process branches. $RTTP(f) \triangleq \varnothing$ for any f which is not a B-node;
- $M : V \to \mathbb{N}$ is a memory function. $M(f_u)$ is the size of the allocated memory of function f_u;
- $NI : V \to [0, +\infty)$ is a function representing the average number of invocations of each function in V, per execution of the serverless workflow G_s;
- $C : V \to [0, +\infty)$ is a cost function. $C(f_u)$ is the cost per invocation of function f_u;

Besides FaaS functions, vertices can also represent other cloud services, such as a MapReduce service or a database operation service. As those services are similar to functions in terms of the execution, we collectively call them functions for brevity. V can also contain several non-functional structural vertices that facilitate developing the workflow and do not incur any delay and cost, such as start node and end node. Typically, the execution of the serverless workflow starts with a particular function as a trigger, and then the following functions will be invoked to complete the business logic [29]. Therefore, we include a start node f_{str} and an end node f_{end} in V defining the entry point and the endpoint of the workflow, respectively. Figure 3 shows an example of a serverless workflow for image classification.

Based on the definition of the serverless workflow, we define the following notations:

1) A simple path in the serverless workflow is a finite sequence of distinct vertices and edges $s = f_1 e_1 f_2 e_2 ... e_{n-1} f_n$ such that (i) $f_i \in V$ for all integers $1 \leq i \leq n$, (ii) $e_i \in E$ for all integers $1 \leq i \leq n-1$, and (iii) $e_i = (f_i, f_{i+1})$ for all integers $1 \leq i \leq n-1$.
2) The transition probability of the simple path $s = f_1 e_1 f_2 e_2 ... e_{n-1} f_n$ is defined as Equation (2).

$$TPP(s) = \prod_{i=1}^{n-1} P(f_i, f_{i+1}) \tag{2}$$

3) The delay (response time) of a simple path $s = f_1 e_1 f_2 e_2 ... e_{n-1} f_n$, denoted as $DLY(s)$, is defined as Equation (3), namely the sum of the response time of functions and the delay incurred by edges in this path. In particular, we use $DLY^-(s)$ to denote the delay of a simple path without considering the response time of the first and last vertices in the simple path, defined as Equation (4).

$$DLY(s) = \sum_{i=1}^{n} RT(f_i) + \sum_{i=1}^{n-1} D(f_i, f_{i+1}) \tag{3}$$

$$DLY^-(s) = \sum_{i=2}^{n-1} RT(f_i) + \sum_{i=1}^{n-1} D(f_i, f_{i+1}) \tag{4}$$

4) $ASP(f_u, f_v)$ denotes all simple paths between vertex f_u and f_v, which is the set of all possible simple paths in the workflow graph, starting from f_u and ending at f_v, with f_u and f_v included.
5) The shortest path length between two vertices f_u and f_v, denoted as $SPL(f_u, f_v)$, is specified as the length of the shortest simple path from f_u to f_v. The length of a simple path is the number of edges in it.
6) $SUB(f_u, f_v)$ denotes the subgraph between $f_u \in V$ and $f_v \in V$, which is derived from G_s by removing all vertices and edges not in any paths in $ASP(f_u, f_v)$.
7) $out(f_u)$ denotes the set of all edges starting from vertex f_u, defined as the following.

$$out(f_u) = \{e \in E : e = (f_u, f) \text{ for some } f \in V\}$$

For convenience, Table 1 includes the definitions of notations and parameters used in the performance and cost models as well as optimization algorithms.

3.2 Structures in the serverless workflow

In this section, we define four types of structures in the serverless workflow, namely parallel, branch, cycle, and self-loop, as shown in Figure 4.

3.2.1 *Parallel*

Let us consider all simple paths between vertices $f_u \in V$ and $f_v \in V$. If there is more than one simple path whose transition probability is 1, we define the subgraph composed of all simple paths with the transition probability of 1 as a ***parallel structure***.

TABLE 1
Definition of Notations used in models and algorithms

Notation	Definition
G_s	the serverless workflow (eq. (1)) $G_s = (V, E, P, D, RT, RTTP, M, NI, C)$
f	a FaaS function, cloud service, or structural vertex
$e = (f_u, f_v)$	a directed edge from f_u to f_v
s	a simple path
$TPP(s)$	transition probability of a simple path s
$DLY(s)$	delay of a simple path s
$ASP(f_u, f_v)$	the set of all simple paths between f_u and f_v
$SPL(f_u, f_v)$	the shortest length of simple paths between f_u and f_v
$SUB(f_u, f_v)$	the subgraph between f_u and f_v
$out(f_u)$	the set of all edges starting from f_u
$ERT(G)$	the end-to-end response time of the workflow G
$\wp$	power set
$\uplus$	disjoint union
$\Downarrow$	restriction of a function
$\mapsto$	map an element in the function domain
$\|A\|$	the cardinal number of a set
G_p	a parallel structure
G_b	a branch structure
G_c	a cycle structure
G_l	a self-loop structure
G_{pr}	probabilistic DAG used in the performance model
G_{dl}	de-looped graph (DAG) used in the cost model
G_{perf}	the graph used in the performance model (eq. (5))
G_{cost}	the graph used in the cost model (eq. (12))
$SRT(G)$	the RT of a parallel/branch/cycle/self-loop
$EI(G)$	the expected number of iterations of a cycle/self-loop
$DI(G)$	the delay incurred by iterations of a cycle/self-loop
f_B	a B-node used in procedures of processing branches
f_P	a P-node used in procedures of processing parallels
PGC	price per GB-second of FaaS functions
PI	price per invocation of FaaS functions
$W(s)$	weight of a simple path s
BCR	Benefit-cost ratio
TH	BCR threshold
$\beta(f_u)$	the slope of the performance-memory curve of f_u

Namely, we define the subgraph $G_p = (V_p, E_p)$ as a parallel structure, such that $|SP_p| > 1$, where

$$V_p = \{f \in V : \text{f is in s for some s} \in SP_p\}$$

$$E_p = \{e \in E : e \text{ is in } s \text{ for some s} \in SP_p\}$$

$$SP_p = \{s \in ASP(f_u, f_v) : TPP(s) = 1\}$$

As the parallel shown in Figure 4, the vertices and edges of the parallel structure are depicted in blue color. There are two simple paths with the transition probability of 1 between f_1 and f_4, namely $f_1e_1f_2e_3f_4$ and $f_1e_2f_3e_4f_4$. Functions f_2 and f_3 are processed in parallel, indicating that f_1 leverages interactions to invoke f_2 and f_3 at the same time after finishing its execution, and f_4 starts execution only after both f_2 and f_3 are completed.

3.2.2 Branch

Let us consider all simple paths between vertices $f_u \in V$ and $f_v \in V$. If there is more than one simple path whose transition probability is less than 1, but can sum up to 1 in total, we define the subgraph composed of those simple paths with the transition probability not equal to 1 as a ***branch structure***.

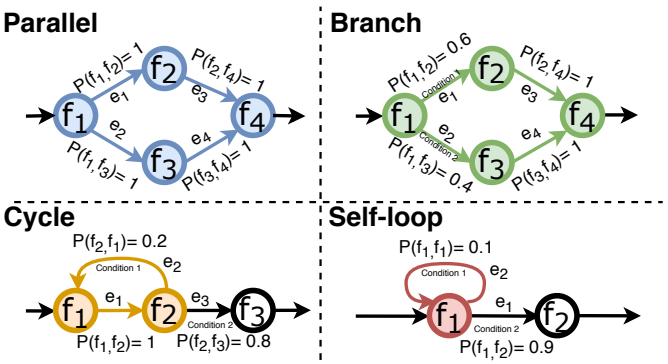

Fig. 4. Four types of structures in the serverless workflow.

Namely, we define the subgraph $G_b = (V_b, E_b)$ as a branch structure, such that $|SP_b| > 1$, where

$$V_b = \{f \in V : \text{f is in s for some s} \in SP_b\}$$

$$E_b = \{e \in E : e \text{ is in } s \text{ for some s} \in SP_b\}$$

$$SP_b = \{s \in ASP(f_u, f_v) : TPP(s) \neq 1\}$$

$$\sum_{s \in SP_b} TPP(s) = 1$$

As the branch shown in Figure 4, the vertices and edges of the parallel structure are depicted in green color. There are two simple paths between f_1 and f_4 whose transition probabilities are not equal to 1, but can add up to 1. Hence, vertices and edges in simple paths $f_1e_1f_2e_3f_4$ and $f_1e_2f_3e_4f_4$ form a branch structure. After completing f_1, the workflow continues with only one path in the branch, depending on the satisfied condition.

3.2.3 Cycle

Considering all simple paths between vertices $f_u \in V$ and $f_v \in V$, we define the subgraph $G_c = (V_c, E_c)$ as a ***cycle structure*** between vertices f_u and f_v, where

$$V_c = \{f \in V : \text{f is in s for some s} \in ASP(f_u, f_v)\}$$

$$E_c = \{e \in E : \text{e is in s for some s} \in ASP(f_u, f_v)\} \uplus \{(f_v, f_u)\}$$

such that

- $0 < P(f_v, f_u) < 1$, invoking f_u again after completing f_v is a possible event;
- $\sum_{s \in ASP(f_u, f_v)} TPP(s) = 1$, transition probabilities of all simple paths between f_u and f_v sum up to 1;
- $SUB(f_u, f_v)$ is a DAG, the subgraph between f_u and f_v does not have any loops and cycles;
- $SPL(f_{str}, f_u) < SPL(f_{str}, f_v)$, compared to f_v, f_u is closer to the entry point;
- $\sum_{(f_v, f_i) \in out(f_v)} P(f_v, f_i) = 1$, the transition probabilities of all edges starting from f_v add up to 1.

As the cycle shown in Figure 4, vertices f_1 and f_2 and edges e_1 and e_2 form a cycle, depicted in orange color. After completing f_2, depending on the satisfied condition, the workflow will either invoke f_3, or enter the cycle by invoking f_1.

3.2.4 *Self-loop*

A self-loop is a special case of cycle with only one vertex and one edge. Considering a function f_u connected by an edge (f_u, f_u) to itself, we define the subgraph $G_l = (V_l, E_l) = (\{f_u\}, \{(f_u, f_u)\})$ as a ***self-loop structure***, such that

- $0 < P(f_u, f_u) < 1$, invoking f_u again after completing f_u is a possible event;
- $\sum_{(f_u, f_i) \in out(f_v)} P(f_u, f_i) = 1$, the transition probabilities of all edges starting from f_u add up to 1.

As the self-loop depicted in red color shown in Figure 4, after accomplishing f_1, the workflow will either invoke f_2, or enter the self-loop by invoking f_1 again.

3.3 Performance Modeling

We propose a performance model to get the end-to-end response time of the serverless workflow. As M, NI, and C defined in G_s are relevant to cost instead of performance, we do not consider them in the performance model for brevity. Specifically, we only consider the following graph with part of elements in G_s, defined as

$$G_{perf} = (V, E, P, D, RT, RTTP) \tag{5}$$

With different methods for different structures, the performance model trims the graph of G_{perf} by removing, adding, and modifying vertices, edges and elements, and converts G_{perf} into a ***probabilistic DAG***, denoted as G_{pr}. We define the probabilistic DAG as

$$G_{pr} = (V, E, P, D, RT, RTTP) \tag{6}$$

such that

- G_{pr} is a DAG without any cycles and loops;
- $\sum_{s \in ASP(f_{str}, f_{end})} TPP(s) = 1$, the transition probabilities of all simple paths between the start node and the end node in G_{pr} can sum up to 1.

The end-to-end response time of the serverless workflow G_{perf}, denoted as $ERT(G_{perf})$, is defined as Equation (7).

$$ERT(G_{perf}) = \sum_{s \in ASP(f_{str}, f_{end})} TPP(s)\, DLY(s) \tag{7}$$

where $ASP(f_{str}, f_{end})$ is the set of all simple paths between the start node and end node in G_{pr}, which is the probabilistic DAG converted from G_{perf} by the proposed performance model.

To convert a serverless workflow into a probabilistic DAG, the model needs to remove cycles and self-loops from the workflow and trim parallel paths. In the following subsections, we describe how the performance model processes four types of structures in the serverless workflow and converts the workflow graph into a probabilistic DAG.

3.3.1 *Process branches*

The workflow selects only one path in the branch depending on the satisfied condition. Each condition has a probability specified by the transition probability function. The main idea of processing the branch is to simplify the workflow by replacing branch paths with an interim B-node while retaining the information, including response time and the probability of each branch path.

Consider a branch $G_b = (V_b, E_b)$ between vertices f_u and f_v, let SRT denote the response time of a structure, we can obtain the branch structure's expected value of the response time by Equation (8).

$$SRT(G_b) = \sum_{s \in ASP(f_u, f_v)} TPP(s)\, DLY(s) \tag{8}$$

After calculating the expected value of the delay, the performance model trims G_s by first removing all vertices in V_b except f_u and f_v from V, namely we get a smaller set of vertices, denoted as V^*, defined as

$$V^* = V \backslash (V_b \backslash \{f_u, f_v\})$$

Correspondingly, we remove all elements relevant to removed vertices by restricting functions as follows:

$$E^* = E \Downarrow V^*, P^* = P \Downarrow V^*, D^* = D \Downarrow V^*$$
$$RT^* = RT \Downarrow V^*, RTTP^* = RTTP \Downarrow V^*$$

Then, we add an interim vertex called "B-node", denoted as f_B, to V, and connect f_u, f_B, and f_v in sequence by adding two edges (f_u, f_B) and (f_B, f_v) to E. We extend functions in the graph tuple as follows:

$$V' = V^* \uplus \{f_B\}, E' = E^* \uplus \{(f_u, f_B), (f_B, f_v)\}$$
$$P' = P^* [(f_u, f_B) \mapsto 1, (f_u, f_B) \mapsto 1]$$
$$D' = D^* [(f_u, f_B) \mapsto 0, (f_u, f_B) \mapsto 0]$$
$$RT' = RT^* [f_B \mapsto SRT(G_b) - RT(f_u) - RT(f_v)]$$

The above procedures simplify the workflow graph, as multiple vertices in the branch are replaced by an interim B-node, whose response time is the weighted average response time of branch paths. However, simply using the weighted average value would compromise the accuracy of the performance model. An example is when a branch is in parallel to other paths, as shown in Figure 5. Therefore, we retain the information, including response time and probability of each branch path using $RTTP$ for later processing.

Specifically, we define $RTTP$ as a function $RTTP : V \rightarrow \wp([0, +\infty) \times [0, 1])$, which represents all possible values of the response time and corresponding probability of all original paths replaced by interim B-nodes. When processing branches and adding B-nodes, we have

$$RTTP' = RTTP^* [f_B \mapsto \mathrm{B}]$$

where

$$\mathrm{B} = \left\{ \left(\mathrm{DLY}^{-}(\mathrm{s}), \mathrm{TPP}(\mathrm{s}) \right) | \forall \mathrm{s} \in \mathrm{ASP}(\mathrm{f_u}, \mathrm{f_v}) \right\}$$

After processing branches, the workflow graph is updated as $G_{perf} \leftarrow G' = (V', E', P', D', RT', RTTP')$.

3.3.2 *Process parallels*

For a given parallel structure $G_p = (V_p, E_p)$ between vertices f_u and f_v, the workflow executes all parallel paths after completing f_u and invokes f_v only after finishing the executions of all paths. Hence, as shown in Equation (9), the response time of a parallel structure is the longest delay of parallel paths in it. Therefore, the main idea of processing

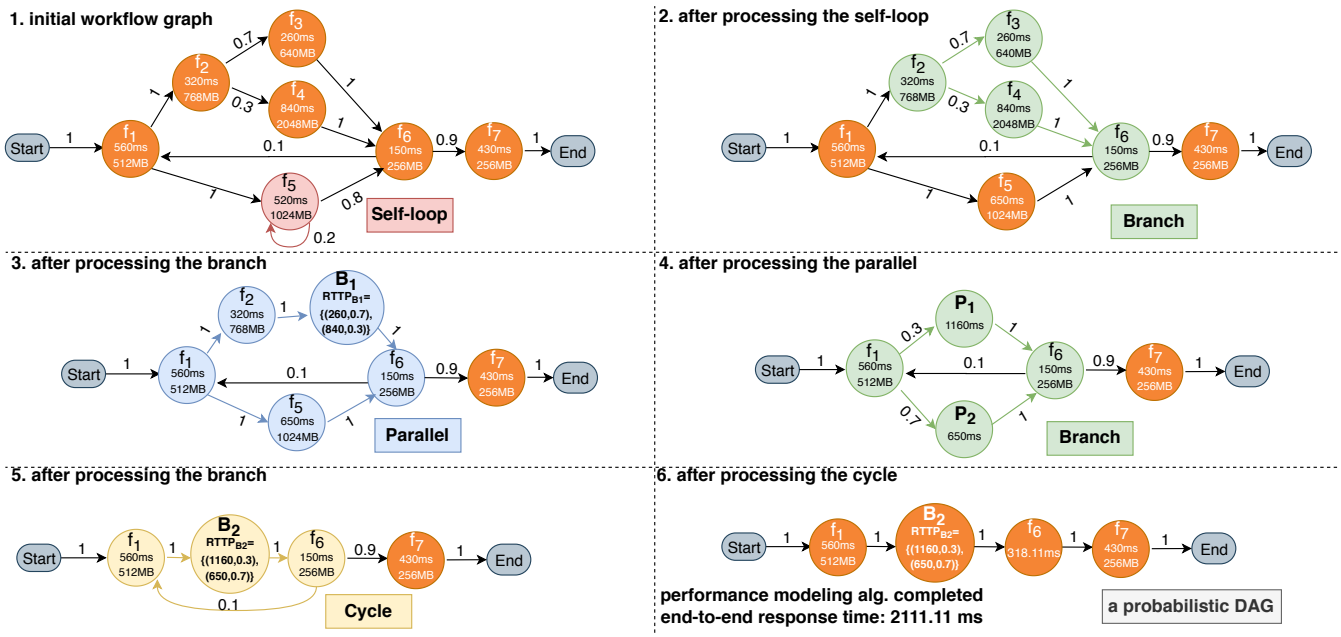

Fig. 5. Steps of the performance modeling algorithm solving the end-to-end response time of the serverless application shown in Figure 3. The interim B-node and P-node (B_1, B_2, P_1, and P_2) are depicted in bold. The value of $RTTP$ of the B-node retains the response time and probability of each path in the branch. The number on each edge represents the transition probability, and the transition delay is considered as zero for brevity.

the parallel is to retain the parallel path with the longest delay and prune other paths.

$$SRT\left(G_p\right) = \max\left\{DLY\left(s\right) | \forall s \in ASP\left(f_u, f_v\right)\right\} \quad (9)$$

For a parallel structure without any interim B-nodes in any paths, the delay of each path is deterministic. We can directly use Equation (9) to get the delay of the parallel structure, which is also deterministic. However, if there are B-nodes in any paths, the delay of the parallel structure may have a probability distribution instead of being a fixed value. Since the response time of the B-node, recorded by $RTTP$, varies based on probabilities under different conditions, the delay of the path with B-nodes is subject to a probability distribution, making the delay of the parallel structure probabilistic. In this case, the performance model leverages Algorithm 1 to get a set of tuples, denoted as $RTTP\_List$, which has all possible values of response time and corresponding probabilities of the parallel structure.

After calculating the delay using Equation (9) or deriving $RTTP\_List$, similar to the trimming step in Section 3.3.1, the performance model removes all vertices in V_p except f_u and f_v from V, and then restricts functions. We have

$$\begin{aligned} V^* &= V \backslash \left(V_p \backslash \left\{f_u, f_v\right\}\right) \\ E^* &= E \Downarrow V^*, P^* = P \Downarrow V^*, D^* = D \Downarrow V^* \\ RT^* &= RT \Downarrow V^*, RTTP^* = RTTP \Downarrow V^* \end{aligned}$$

Then, we add the interim vertex called "P-node" to V. If the parallel structure does not have any B-nodes, we only add one P-node, denoted as f_P and let

$$\begin{aligned} &V' = V^* \uplus \left\{f_P\right\}, E' = E^* \uplus \left\{\left(f_u, f_P\right), \left(f_P, f_v\right)\right\} \\ &P' = P^*\left[\left(f_u, f_P\right) \mapsto 1, \left(f_P, f_v\right) \mapsto 1\right] \\ &D' = D^*\left[\left(f_u, f_P\right) \mapsto 0, \left(f_P, f_v\right) \mapsto 0\right] \\ &RT' = RT^*\left[f_P \mapsto SRT\left(G_p\right) - RT\left(f_u\right) - RT\left(f_v\right)\right] \end{aligned}$$

If V_p contains B-nodes, by using Algorithm 1, we have $RTTP\_List = \left\{\left(rt_1, pr_1\right), \left(rt_2, pr_2\right), ..., \left(rt_M, pr_M\right)\right\}$, which has M tuples of the response time and probability. Then, M P-nodes will be added as follows:

$$\begin{aligned} &V' = V^* \uplus_{i=1}^{M} \left\{f_{P_i}\right\}, E' = E^* \uplus_{i=1}^{M} \left\{\left(f_u, f_{P_i}\right), \left(f_{P_i}, f_v\right)\right\} \\ &\text{for all } 1 \leq i \leq M\text{:} \\ &P' = P^*\left[\left(f_u, f_{p_i}\right) \mapsto pr_i, \left(f_{p_i}, f_v\right) \mapsto 1\right] \\ &D' = D^*\left[\left(f_u, f_{p_i}\right) \mapsto 0, \left(f_{p_i}, f_v\right) \mapsto 0\right] \\ &RT' = RT^*\left[f_{p_i} \mapsto tp_i\right] \end{aligned}$$

After processing paralles, the workflow graph is updated as $G_{perf} \leftarrow G' = \left(V', E', P', D', RT', RTTP'\right)$.

3.3.3 *Process cycles*

For processing cycles, the main idea is to remove the cycle and add the delay incurred by cycle iterations to the last vertex's response time in the cycle. For a given cycle $G_c = \left(V_c, E_c\right)$ between vertices f_u and f_v, the expected value of the number of cycle G_c iterations, denoted as $EI\left(G_c\right)$, namely the average number of times the workflow enters the cycle after completing f_v, can be expressed as Equation (10).

$$\begin{aligned} EI\left(G_c\right) &= \sum_{n=1}^{\infty}\left[1 - P\left(f_v, f_u\right)\right]\left[P\left(f_v, f_u\right)\right]^{n-1}\left(n-1\right) \\ &= \frac{P\left(f_v, f_u\right)}{1 - P\left(f_v, f_u\right)} \end{aligned} \quad (10)$$

By multiplying the expected number of cycle iterations and the time required for each iteration, we can calculate the expected delay incurred by cycle iterations.

$$DI\left(G_c\right) = \left(\sum_{s \in ASP\left(f_u, f_v\right)} TPP\left(s\right) DLY\left(s\right) + D\left(f_v, f_u\right)\right) EI\left(G_c\right)$$

Algorithm 1: Get $RTTP\_List$

Input: a parallel $G_p = (V_p, E_p)$ between $f_u \in V_p$ and $f_v \in V_p$ with n paths
Output: a set of tuples $RTTP\_List$ which has all possible values of response time and corresponding probability of G_P

for *path* $s_i \in ASP(f_u, f_v)$ **do**
  $RTTP\_List_i \leftarrow []$; ▷ results list
  if s_i *has* $m \geq 1$ *B-nodes* $\{f_{B_1}, f_{B_2}, ..., f_{B_m}\}$ **then**
    $\text{RT\_wo\_B} \leftarrow DLY^{-}(s_i) - \sum_{k=1}^{m} RT(f_{B_k})$
    ▷ the delay of the path without B-nodes
    $cmb \leftarrow RTTP(f_{B_1}) \times ... \times RTTP(f_{B_m})$;
    ▷ all combinations of the RT and prob. of all B-nodes by the Cartesian product
    for *each combo* C_j *in cmb* **do**
      $RTC_j \leftarrow \sum_{(rt_k, tp_k) \in C_j} rt_k + RT\_wo\_B$;
      ▷ a possible response time of s_i
      $TPC_j \leftarrow \prod_{(rt_k, tp_k) \in C_j} tp_k$;
      ▷ the corresponding probability
      append (RTC_j, TPC_j) to $RTTP\_List_i$;
    end
  else ▷ the path that does not have any B-nodes
    $RTC \leftarrow RT(s_i) - r_x - r_y$; ▷ RT is deterministic
    append $(RTC, 1)$ to $RTTP\_List_i$; ▷ prob. is 1
  end
end
$rttp\_comb \leftarrow RTTP\_List_1 \times ... \times RTTP\_List_n$;
$RTTP\_List \leftarrow []$;
for *each combo* C_j *in* $rttp\_comb$ **do**
  $RTC_j \leftarrow \max\{rt_k | \forall (rt_k, tp_k) \in C_j\}$; ▷ a possible RT of G_p, use maximum value due to parallelism
  $TPC_j \leftarrow \prod_{(rt_k, tp_k) \in C_j} tp_k$; ▷ the corresponding prob.
  append (RTC_j, TPC_j) to $RTTP\_List$;
end
return $RTTP\_List$

In terms of the response time of the cycle structure, the delay incurred by cycle iterations is equivalent to increasing the response time of f_v by the same amount of time. Therefore, the performance model first removes the edge (f_v, f_u) as

$$E' = E \backslash \{(f_v, f_u)\}$$

Then, the model modifies the transition probability, edge delay, and vertex response time as follows:

$$P^* = P\left[e \mapsto \frac{P(e)}{1 - P(f_v, f_u)}\right], \text{for all } e \in out(f_v)$$
$$D' = D\left[(f_v, f_u) \mapsto 0\right], P' = P^*\left[(f_v, f_u) \mapsto 0\right]$$
$$RT' = RT\left[f_v \mapsto RT(f_v) + DI(G_c)\right]$$

After processing cycles, the workflow graph is updated as $G_{perf} \leftarrow G' = (V, E', P', D', RT', RTTP)$.

3.3.4 *Process self-loops*

The procedure to process self-loops is similar to that used to process cycles. Given a self-loop $G_l = (\{f_u\}, \{(f_u, f_u)\})$, similarly using Equation (10), we can calculate the expected number of self-loop iterations as $EI(G_l) = \frac{P(f_u, f_u)}{1 - P(f_u, f_u)}$, and the delay incurred by self-loop iterations as $DI(G_l) = EI(G_l)(RT(f_u) + D(f_u, f_u))$. Then, the performance model updates the graph as $G_{perf} \leftarrow G' = (V, E', P', D', RT', RTTP)$, where

$$P^* = P\left[e \mapsto \frac{P(e)}{1 - P(f_u, f_u)}\right], \text{for all } e \in out(f_u)$$
$$D' = D\left[(f_u, f_u) \mapsto 0\right], P' = P^*\left[(f_u, f_u) \mapsto 0\right]$$
$$E' = E \backslash \{(f_u, f_u)\}$$
$$RT' = RT\left[f_u \mapsto RT(f_u) + DI(G_l)\right] \quad (11)$$

3.4 Cost Modeling

In this section, we introduce a cost model to get the average cost of the serverless workflow. Since FaaS platforms leverage a GB-second billing model depending on the allocated memory size, rounded-up function duration, and the number of invocations, we consider the following graph with part of elements in G_s, defined as

$$G_{\cos t} = (V, E, P, RT, M, NI, C) \quad (12)$$

Algorithm 2: Performance modeling algorithm

Input: a serverless workflow $G_s = (V_s, E_s)$
Output: the average end-to-end response time of the serverless application

$G' \leftarrow G_s$;
while G' *is not a probabilistic DAG (using eq.* (6)*)* **do**
  loop_list ← find_self_loops(G'); ▷ section 3.2.4
  for *each self-loop* G_i *in loop_list* **do**
    Process self-loop G_i; ▷ Section 3.3.4
  end
  cycle_list ← find_cycles(G'); ▷ section 3.2.3
  for *each cycle* G_i *in self_loop_list* **do**
    Process cycle G_i; ▷ Section 3.3.3
  end
  parallel_list ← find_parallels(G'); ▷ section 3.2.1
  for *each parallel* G_i *in parallel_list* **do**
    Process parallel G_i; ▷ Section 3.3.2
  end
  branch_list ← find_branches(G'); ▷ section 3.2.2
  for *each branch* $G_i = (V_i, E_i)$ *in branch_list* **do**
    Process branch G_i; ▷ Section 3.3.1
  end
end
$ERT \leftarrow \sum_{s \in ASP_{G'}(f_{str}, f_{end})} TPP(s) \cdot DLY(s)$; ▷ eq. (7)
return ERT

All vertices representing FaaS functions have an amount of allocated memory, identified by M. The allocated memory of 0 represents such vertex might be a structural node without any cost, like start and end nodes, or other cloud services to which the GB-second billing model is not applicable. The rounded-up function duration can be directly calculated using the response time of each function defined by RT. For a FaaS function $f_u \in V$, its average cost, per application execution, can be calculated as Equation (13), where PGS is the price per GB-second and PPI is the price per invocation (the cost of handling the invocation request). For vertices representing cloud services with other pricing models, the model obtains their cost from users' input.

$$C(f_u) = NI(f_u)\left(\left\lceil \frac{RT(f_u)}{100} \right\rceil \cdot M(f_u) \cdot PGS + PPI\right) \quad (13)$$

Each vertex $f_u \in V$ has an average number of invocations, denoted as $NI(f_u)$, which depends on the structure of the serverless workflow. Branches can reduce the number of invocations of vertices in them, since the transition probability of paths in branches is less than 1. Conversely, cycles and self-loops can lead to more than once invocation of functions in them, and the number of invocations depends on the expected value of the number of cycle/self-loop iterations, calculated as Equation (10). Hence, for a given cycle $G_c = (V_c, E_c)$ between vertices f_u and f_v, we have the expected value of the number of invocations of each vertex $f_i \in V_c$ as

$$NI\left(f_i\right) = 1 + EI(G_c) = \frac{1}{1 - P(f_v, f_u)}, \forall f_i \in V_c \quad (14)$$

Similarly, for a self-loop $G_l = (\{f_u\}, \{(f_u, f_u)\})$, we have

$$NI\left(f_u\right) = 1 + EI(G_l) = \frac{1}{1 - P(f_u, f_u)} \quad (15)$$

The cost model first leverages Equation (14) and Equation (15) to calculate the average number of invocations of vertices in cycles and self-loops. Then, similar to procedures of processing cycles and self-loops in the performance model, the cost model updates the edge and transition probability to remove cycles and self-loops from the graph. After removing all cycles and self-loops, considering the impact of parallels and branches, the cost model updates NI of each vertex based on the sum of transition probabilities of all simple paths from the start node to it. By following these steps, we convert G_{cost} into a de-looped graph used for cost modeling, denoted as G_{dl}. The average cost of the serverless workflow can be calculated by $\sum_{f \in V} C\left(f\right)$. Algorithm 3 gives the pseudo-code of the cost model.

3.5 Summary, Example, and Analysis

The implementation of the performance model is presented in Algorithm 2. Based on definitions of structures in the serverless workflow, as mentioned in Section 3.1, the algorithm identifies structures in the workflow and leverages different procedures to process structures, defined in Section 3.3, to trim the workflow graph to a probabilistic DAG and obtain the end-to-end response time of the application. Figure 5 illustrates the step-by-step changes of the workflow graph when the performance model works on the serverless workflow shown in Figure 3. The average end-to-end response time of the application is 2111.11ms.

Algorithm 3 describes the implementation of the cost model. Taking the serverless workflow showing in Figure 3 as an example, after completing the first while loop, the algorithm trims the workflow to a de-looped graph shown in Figure 6. Then, in the for loop, the algorithm updates the average number of invocations for each function again and calculates the cost of the applications. The average cost of the application is $41.82 per 1 million executions.

Let us consider the worst case scenario and analyze the time complexities of the performance and cost models. Under the definition of the serverless workflow mentioned in 3.1, the most complex workflow is composed of as many cycles and self-loops as possible, since the cycle and self-loop structures require the least number of vertices,

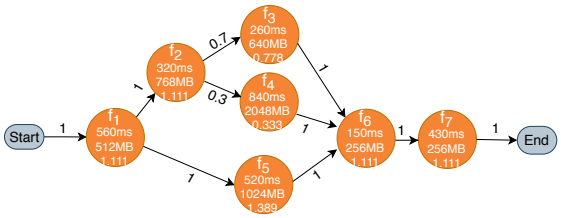

Fig. 6. The de-looped workflow of the serverless application shown in Figure 3. The three numbers on each function represent the response time, allocated memory and average number of invocations, respectively. The number on each edge represents the transition probability.

Algorithm 3: Cost modeling algorithm

Input: a workflow graph G_{cost}
Output: the average cost for each execution of the serverless application

$G' \leftarrow G_{cost}$;
for all $f \in V$, let $NI(f) \leftarrow 1$; ▷ initialization
while G' *is not a DAG* **do**
  loop_list ← find_self_loops(G'); ▷ section 3.2.4
  for *self-loop* $G_l = (f_u, (f_u, f_u))$ *in loop_list* **do**
    $NI(f_u) \leftarrow \frac{1}{1-P(f_u,f_u)}$; ▷ eq. (15)
    $E \leftarrow E \backslash \{(f_u, f_u)\}$; ▷ remove the loop edge
    for *each edge e in out*(f_u) **do**
      $P \leftarrow P\left[e \mapsto \frac{P(e)}{1-P(f_u,f_u)}\right]$; ▷ updat prob.
    end
  end
  cycle_list ← find_cycles(G');
  for *each cycle* $G_i = (V_i, E_i)$ *in cycle_list* **do**
    For each $f \in V_i$ update $NI(f)$ using eq. (14);
    Remove the cycle edge;
    Update transition probabilities of outgoing edges; ▷ similar to steps for self-loops
  end
end
for *each vertex* $f \in V$ **do**
  $tp\_sum \leftarrow \sum_{s \in ASP_{G'}(f_{str}, f)} TPP\left(s\right)$;
  if $tp\_sum < 1$ **then**
    $NI(f) \leftarrow NI(f) \cdot tp\_sum$;
  end
  $cost\_sum \leftarrow cost\_sum + C(f)$; ▷ eq. (13)
end
return *cost_sum*

compared to the parallel and branch structures. Figure 7 illustrates the workflow under the worst case scenario, where the workflow is composed of n functions, $(n+2)$ vertices, $\frac{n(n+3)+2}{2}$ edges, $\frac{n(n-1)}{2}$ cycles, and n self-loops. The time complexities for detecting self-loops and cycles are $O(|E|)$ and $O(C(|V| + |E|))$, respectively, where C is the number of cycles. The performance model and cost model can process cycles and self-loops in $O(n^2)$ and $O(n)$ time, respectively. Hence, the time complexities for the performance and cost modeling under the worst case scenario are $O(n^6)$ and $O(n^5)$, respectively, where n is the number of functions in the workflow. We empirically analyze the AWS and Azure official repositories [30], [31]. The average number of functions in serverless applications in this repository, orchestrated by Amazon Step Functions or Microsoft Azure Functions, is less than 5. We do not find

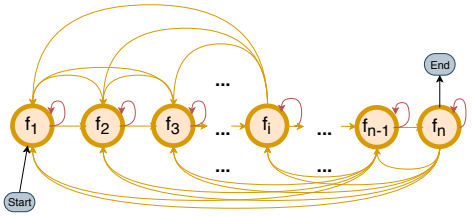

Fig. 7. The workflow of the worst case, which contains n functions, $(n+2)$ vertices, $\frac{n(n+3)+2}{2}$ edges, $\frac{n(n-1)}{2}$ cycles, and n self-loops.

any serverless application in the repositories resembling the worst case topology. The most common typologies appear to be sequential, paralleled, and branched. Our proposed models can calculate the performance and cost of a worst-case topology application with 27 functions and 406 edges in less than a second on a laptop with a 2.70GHz Intel Core i7-3740QM processor and 16 GB of memory. This shows the applicability of the models for now and a foreseeable future.

4 Performance and Cost Optimization

As mentioned in Section 2, the response time of the function varies with the allocated memory, and so does the cost. Therefore, developers can tune the performance and cost of the serverless application by changing the allocated memory size of functions in the application. More specifically, as described by RQ2, very practical problems in serverless computing are to get the best performance under a limited budget or satisfy the performance constraint by the minimum cost. RQ2 leads to two performance and cost optimization problems. To answer them, we introduce the performance profile, define the optimization problem and propose an optimization algorithm in this section.

4.1 Performance Profile of Serverless Functions

On FaaS platforms, increasing the memory of a function can improve the CPU and IO performance, but it does not necessarily reduce the response time of the function significantly, especially when the performance becomes insensitive to the memory when the amount is large enough. As shown in Figure 2, when the allocated memory is greater than 1792MB, the response time remains almost the same, and at this time, the continued increase in memory size incurs additional cost due to the rounding and granularity of billed duration. Similarly, albeit larger allocated memory size leads to a higher price per second of billed duration, the cost may decrease with larger memory, especially before the performance becomes insensitive. For the performance-memory curve of functions, as the allocated memory size increases, response time decreases first and then levels out, since the performance becomes insensitive. However, because of the rounding and billing granularity, there are large fluctuations in the cost-memory curve of functions.

Therefore, as mentioned in the best practices for working with AWS Lambda Functions, to find an optimal allocated memory size satisfying both the trade-off between performance and price and the required memory size for the function execution, a performance profiling phase is highly-recommended to test the performance of the FaaS functions [32]. In the performance profiling phase, the function is invoked using the payload with the average input size multiple times under different allocated memory size, and the average duration of invocations is logged. By doing so, we can acquire a set of viable memory size and a series of function response time under different memory sizes. Figure 2 demonstrates the performance-memory curve of a function obtained in the performance profiling phase. Explicitly, we let $MOpt$ denote a memory option function mapping the function to its viable memory options, defined as Equation (16).

$$MOpt : V \to \wp(\mathbb{N}) \tag{16}$$

The performance profile of a function f_u, which describes the response time of the function under different allocated memory sizes, is defined as Equation (17). $PF(f_u, mem_v)$ is the response time of f_u with the allocated memory size of mem_v, where $f_u \in V$ and $mem_v \in MOpt(f_u)$.

$$PF : \biguplus_{f \in V} \{\{f\} \times MOpt(f)\} \to [0, +\infty) \tag{17}$$

Considering the performance profile of functions, in this section, we extend the definition of serverless workflow as

$$G_s = (V, E, P, D, RT, RTTP, M, NI, C, MOpt, PF)$$

where $MOpt$ and PF are defined as Equation (16) and Equation (17), respectively. Let $MOpt(f) \triangleq \varnothing$ for all f is not a FaaS function.

4.2 Problem Statement

Considering a serverless workflow G_s with n functions, where $V = \{f_1, f_2, ..., f_{n-1}, f_n\}$, we define π as a memory configuration of the workflow, such that

$$\pi \in MOpt(f_1) \times MOpt(f_2) \times ... \times MOpt(f_n)$$

$\pi(f)$ denotes the size of the allocated memory of the function f in this configuration. Let ERT^π denote the end-to-end response time of G_s obtained by the performance model and $C^\pi(f_u)$ denote the cost of the function f_u obtained by the cost model, under the memory configuration π, such that (i) $M(f_i) = \pi(f_i)$, for all $1 \le i \le n$, (ii) $RT(f_i) = PF(f_i, \pi(i))$, for all $1 \le i \le n$. For a given budget limit BC, and a performance constraint PC, we define the following two optimization problems.

4.2.1 *Best Performance under Budget Constraint (BPBC)*

Find a memory configuration π that achieves the minimum average end-to-end response time of the application with the average cost less than or equal to the budget BC.

$$\begin{aligned} &\arg\min_{\pi} \quad ERT^\pi(G_s) \\ &subject\ to \quad \sum_{i=1}^{n} C^\pi(f_i) \le BC \end{aligned} \tag{18}$$

4.2.2 *Best Cost under Performance Constraint (BCPC)*

Find a memory configuration π that achieves the minimum average cost of the application with the average end-to-end response time less than or equal to the performance constraint PC.

$$\begin{aligned} &\arg\min_{\pi} \quad \sum_{i=1}^{n} C^{\pi}(f_i) \\ &subject\ to \quad ERT^{\pi}(G_s) \leq PC \end{aligned} \tag{19}$$

4.3 Problem Complexity Analysis

We prove that BPBC and BCPC problems in the serverless computing paradigm are fundamentally more complex variants of the multiple-choice knapsack problem (MCKP). MCKP is formulated as follows. Given n sets $\mathbb{N}_1, \mathbb{N}_2, ..., \mathbb{N}_n$ of items, where each item in each set has a profit and a weight, by selecting exactly one item from each set, the optimization problem is to find a selection combination $\varsigma \in \mathbb{N}_1 \times \mathbb{N}_2 \times ... \times \mathbb{N}_n$ such that ς maximizes the total profit while the total weight within the capacity. MCKP has been applied to many optimization problems, including resource allocation and workflow optimization in the parallel computing and microservice-based applications [33], [34].

In BPBC and BCPC problems, the n functions, viable allocated memory sizes of each function $MOpt$, and the memory configuration π are equivalent to n sets, a number of items in each set, and selection combination ς in MCKP, respectively. For the BPBC problem, BC corresponds to the knapsack capacity constraint, $-PF(f, M_k)$ can be viewed as the profit of the function $f \in V$ with the allocated memory size of $M_k \in MOpt(f)$, and corresponding cost $C(f)$ is equivalent to the weight. Instead of simply accumulating values, the total profit in the BPBC problem, represents the end-to-end response time of the application, should be derived by the performance modeling algorithm defined in Algorithm 2. Similarly, for the BCPC problem, PC corresponds to the knapsack capacity constraint, the cost $C(f)$ of the function f under the memory $M_k \in MOpt(f)$ can be viewed as the profit of the function, the response time $PF(f, M_k)$ can be deemed as the weight.

Compared to MCKP, the higher complexity of BPBC and BCPC problems lies in calculating the total profit and total weight, which leverages the polynomial-time performance and cost models defined in Section 3. Besides, for a given memory configuration and corresponding response time, we can check whether the average end-to-end response time and total cost under such a configuration satisfy the constraints in polynomial time. Hence, BPBC and BCPC problems in the serverless computing paradigm are fundamentally more complex variants of MCKP.

As MCKP has proven to be a NP-complete problem without any solutions in polynomial time, unless $P = NP$ [35], we have to resort to a heuristic algorithm to solve BPBC and BCPC problems.

4.4 Probability Refined Critical Path Algorithm (PRCP)

In this section, we propose a heuristic algorithm based on the critical path method to solve BPBC and BCPC problems.

4.4.1 *Critical Path Method*

Critical path method (CPM) is a heuristic approach for scheduling problems, which optimizes the scheduling scheme by identifying and rescheduling the path with the longest execution time [36]. CPM has proven to be an effective solution to scheduling problems in many areas including the scientific workflow system [33], distributed computing framework [37], IaaS paradigm [38], and CaaS paradigm [34]. However, previous studies have largely applied CPM to DAG workflows. As mentioned in Section 2, there can be cycles and loops in the serverless workflow, to which the traditional CPM is not applicable.

4.4.2 *Algorithm Design*

We propose a Probability Refined Critical Path Algorithm (PRCP) to solve BPBC and BCPC optimization problems for non-DAG serverless workflows. To work with non-DAG workflow topology, PRCP refines the transition probability of edges and simple paths as well as the weight of simple paths based on the transition probability, and leverages a weight-based definition of the critical path. PRCP recursively optimizes the memory of functions on the critical path using a greedy manner and obtains the best memory configurations satisfying the constraint.

Based on the definition of the serverless workflow, due to transition probabilities and structures in the workflow, the path with the longest delay may not be the critical path in terms of the response time and cost. Therefore, we need to re-define the critical path to take the transition probability of the path and iterations incurred by cycles and loops into account.

For a simple path $s_k = f_1 e_1 f_2 e_2 ... e_{n-1} f_n$, we define the weight of s_k as Equation (20).

$$W(s_k) = \sum_{i=1}^{n} RT(f_i) NI(f_i) TPP(s_k) \tag{20}$$

Given a set of M simple paths, we define the path with i^{th} greatest weight as the i^{th} critical path, such that $1 \leq i \leq M$. We denote $FindCriticalPath(G, i)$ as a procedure which returns the i^{th} critical path in G.

Given a function f_u and a new memory size $mem_v \in MOpt(f_u)$, by assigning the new allocated memory size mem_v to f_u, we define the change of end-to-end response time and the change of cost, as $\Delta ERT(f_u, mem_v)$ and $\Delta C(f_u, mem_v)$, respectively, where

$$\begin{aligned} \Delta ERT(f_u, mem_v) &= ERT^{\gamma}(G_s) - ERT_{curr} \\ \Delta C(f_u, mem_v) &= \sum_{i=1}^{n} C^{\gamma}(f_i) - C_{curr} \end{aligned}$$

such that (i) ERT_{curr} and C_{curr} are the end-to-end response time and cost under the previous configuration π; (ii) $\gamma(f_u) = mem_v$; (iii) $\gamma(f_i) = \pi(f_i)$ for all $1 \leq i \leq n$ and $i \neq u$.

Algorithm 4 demonstrates the pseudocode of the PRCP algorithm solving the BPBC problem. The input are the budget constraint BC and a serverless workflow G_s with n functions, where $V = \{f_1, f_2, ..., f_{n-1}, f_n\}$. PRCP first initializes the workflow by employing the minimum memory configuration π_{min}, such that $\pi_{\min}(f_i) = \min(MOpt(f_i))$

for all $1 \leq i \leq n$. The workflow has the largest end-to-end response time under the minimum memory configuration. PRCP recursively finds the critical path and calculates ΔERT and ΔC under all possible allocated memory size of all functions in the critical path. If there are functions and memory size options that can reduce the cost without increasing the end-to-end response time of the workflow, namely $\Delta C < 0$ and $\Delta ERT \leq 0$, PRCP selects the function and memory size resulting in the largest cost decrease. If not, PRCP chooses the function and memory size achieving the largest end-to-end response time decrease within the budget constraint. If there is no feasible memory size option within the t^{th} critical path, where t is initialized as 1, PRCP will find the $(t+1)^{th}$ critical path in the next iteration, and so on. The iteration ends when there is no feasible memory size option in the least critical path with the current surplus.

Algorithm 4: PRCP-BPBC

Input: Budget constraint BC, serverless workflow $G_s = (V, E, P, D, RT, RTTP, M, NI, C, MOpt, PF)$
Output: memory configuration π satisfying eq. (18)

$M \leftarrow M\left[f \mapsto \min\left(MOpt\left(f\right)\right)\right], \forall f \in V$; ▷ use the minimal memory configuration π_{min}
$G_{dl} \leftarrow (V, E', P', RT, M, NI, C, MOpt, PF)$; ▷ Get the de-looped graph using steps in Section 3.4
$NSP \leftarrow |ASP_{G_{dl}}\left(f_{str}, f_{end}\right)|$; ▷ number of simple paths in G_{dl}
$\pi_{curr} \leftarrow \pi_{min}$; ▷ current memory configuration
$C_{curr} \leftarrow \sum_{i=1}^{n} C^{\pi_{min}}\left(f_i\right)$; ▷ current cost
$ERT_{curr} \leftarrow ERT^{\pi_{min}}\left(G_s\right)$; ▷ current end-to-end RT
$t \leftarrow 1$;
while $BC - C_{curr} > 0$ *and* $t \leq NSP$ **do**
  $s_{cp} \leftarrow FindCriticalPath\left(G_{dl}, t\right)$; ▷ find t^{th} critical path in G_{dl}
  for *all functions* f_i *in* s_{cp} **do**
    for *all selectable memory* $mem_j \in MOpt(f_i)$ **do**
      Calculate $\Delta ERT\left(f_i, mem_j\right)$ and $\Delta C\left(f_i, mem_j\right)$;
    end
  end
  if $\exists mem_v \in MOpt\left(f_u\right)$: $\Delta ERT\left(f_u, mem_v\right) \leq 0$ *and* $\Delta C\left(f_u, mem_v\right) < 0$ **then**
    $f_{tmp}, mem_{tmp} \leftarrow \underset{f_u, mem_v}{\arg\min}\, \Delta C\left(f_u, mem_v\right)$;
  else
    $\boxed{f_{tmp}, mem_{tmp} \leftarrow \underset{f_u, mem_v}{\arg\min}\, \Delta ERT\left(f_u, mem_v\right)}$ s.t. $\Delta C\left(f_u, mem_v\right) \leq BC - C_{curr}$ and $\Delta ERT\left(f_u, mem_v\right) < 0$. If there are multiple functions and memory values that achieve the same maximum ΔERT, select f_u and mem_v that leads to the smallest ΔC;
  end
  if f_{tmp} *and* mem_{tmp} *exist* **then**
    $M \leftarrow M\left[f_{tmp} \mapsto mem_{tmp}\right]$;
    $RT \leftarrow RT\left[f_{tmp} \mapsto PF\left(f_{tmp}, mem_{tmp}\right)\right]$;
    $\pi_{curr} \leftarrow \pi_{curr}\left(f_{tmp}\right) \triangleq mem_{tmp}$;
    $C_{curr} \leftarrow C_{curr} + \Delta C\left(f_{tmp}, mem_{tmp}\right)$;
    $ERT_{curr} \leftarrow ERT_{curr} + \Delta ERT\left(f_{tmp}, mem_{tmp}\right)$;
  else
    $t \leftarrow t + 1$;
  end
end
return π_{curr}

Algorithm 5 provides the pseudocode of the PRCP algorithm solving the BCPC problem. For the the BCPC problem, PRCP algorithm first initializes the workflow by employing the maximum memory configuration π_{max}, such that $\pi_{\max}\left(f_i\right) = \max\left(MOpt\left(f_i\right)\right)$ for all $1 \leq i \leq n$. The workflow has the smallest end-to-end response time under the maximum memory configuration. PRCP recursively finds the critical path, but starts with the path with the smallest weight, then calculates ΔERT and ΔC under all possible allocated memory size of all functions in that path. If there are functions and memory size options that can reduce the end-to-end response time of the workflow without incurring additional cost, namely $\Delta ERT < 0$ and $\Delta C \leq 0$, PRCP selects the function and memory size leading to the largest end-to-end response time decrease. If not, PRCP chooses the function and memory size resulting in the largest cost decrease within the performance constraint. If there is no feasible memory size option within the t^{th} critical path, where t starts with the number of simple paths in G_{dl}, PRCP will find the $(t-1)^{th}$ critical path in the next iteration, and so on. The iteration ends when there is no feasible memory size option even in the most critical path with the current surplus in terms of the performance.

4.4.3 *Benefit/Cost Ratio (BCR) Greedy Strategies*

PRCP algorithm is essentially a greedy heuristics for BPBC and BCPC problems. The steps with a box in Algorithm 4 and Algorithm 5 are greedy strategies. For instance, in the BPBC problem, the greedy strategy is to find the f_u and $mem_v \in MOpt(f_u)$ in each critical path iteration, which can lead to the greatest end-to-end response time reduction. However, the local optimization achieved by such a strategy might compromise the global optimization. As mentioned in Section 4.1 and shown in Figure 2, after the performance becomes insensitive to the memory, the performance gain of the function brought by the increase in memory is insignificant, and due to rounding and billing granularity, the cost may increase significantly.

To avoid bad optimization solutions, we introduce the benefit/cost ratio (BCR) into greedy strategies. Instead of arbitrarily maximizing the end-to-end response time reduction, the strategy is to find the configuration achieving the optimal BCR. In other words, the idea is to find the configuration leading to the optimal benefit for its cost. We propose three BCR greedy strategies for each optimization problem and integrate them into the PRCP algorithm. For the BPBC problem, strategies are MAX, ERT/C, and RT/M. For the BCPC problem, strategies are MAX, C/ERT, and M/RT.

In the MAX strategy for the BPBC problem, the reduction on end-to-end response time is the benefit, and the cost is the increased cost of the workflow incurred by the configuration. Namely, for the function f_u and a selectable memory size $mem_v \in MOpt(f_u)$, BCR is defined as

$$BCR\left(f_u, mem_v\right) = \left|\frac{\Delta ERT\left(f_u, mem_v\right)}{\Delta C\left(f_u, mem_v\right)}\right| \tag{21}$$

The MAX strategy finds the f_u and $mem_v \in MOpt(f_u)$ in each critical path iteration leading to the maximum BCR, which is defined as Equation (21).

The ERT/C strategy has the same definitions of the benefit and cost as the MAX strategy. A BCR threshold,

Algorithm 5: PRCP-BCPC

Input: Performance constraint PC, serverless workflow $G_s = (V, E, P, D, RT, RTTP, M, NI, C, MOpt, PF)$
Output: memory configuration π satisfying eq. (18)

$M \leftarrow M\left[f \mapsto \max\left(MOpt\left(f\right)\right)\right], \forall f \in V$; ▷ use the minimal memory configuration π_{max}
$G_{dl} \leftarrow (V, E', P', RT, M, NI, C, MOpt, PF)$; ▷ Get the de-looped graph using steps in Section 3.4
$NSP \leftarrow |ASP_{G_{dl}}\left(f_{str}, f_{end}\right)|$; ▷ number of simple paths in G_{dl}
$\pi_{curr} \leftarrow \pi_{max}$; ▷ current memory configuration
$C_{curr} \leftarrow \sum_{i=1}^{n} C^{\pi_{max}}\left(f_i\right)$; ▷ current cost
$ERT_{curr} \leftarrow ERT^{\pi_{max}}\left(G_s\right)$; ▷ current end-to-end RT
$t \leftarrow$ NSP;
while $PC - ERT_{curr} > 0$ *and* $t \geq 1$ **do**
  $s_{cp} \leftarrow FindCriticalPath\left(G_{dl}, t\right)$; ▷ find t^{th} critical path in G_{dl}
  for *all functions* f_i *in* s_{cp} **do**
    for *all selectable memory* $mem_j \in MOpt(f_i)$ **do**
      Calculate $\Delta ERT\left(f_i, mem_j\right)$ and $\Delta C\left(f_i, mem_j\right)$;
    end
  end
  if $\exists mem_v \in MOpt\left(f_u\right)$: $\Delta C\left(f_u, mem_v\right) \leq 0$ *and* $\Delta ERT\left(f_u, mem_v\right) < 0$ **then**
    $f_{tmp}, mem_{tmp} \leftarrow \underset{f_u, mem_v}{\arg\min}\, \Delta ERT\left(f_u, mem_v\right)$;
  else
    $\boxed{f_{tmp}, mem_{tmp} \leftarrow \underset{f_u, mem_v}{\arg\min}\, \Delta C\left(f_u, mem_v\right)}$ s.t. $\Delta ERT\left(f_u, mem_v\right) \leq PC - ERT_{curr}$ and $\Delta C\left(f_u, mem_v\right) < 0$. If there are multiple functions and memory values that achieve the same minimum ΔC, select f_u and mem_v that leads to the smallest ΔERT;
  end
  if f_{tmp} *and* mem_{tmp} *exist* **then**
    $M \leftarrow M\left[f_{tmp} \mapsto mem_{tmp}\right]$;
    $RT \leftarrow RT\left[f_{tmp} \mapsto PF\left(f_{tmp}, mem_{tmp}\right)\right]$;
    $\pi_{curr} \leftarrow \pi_{curr}\left(f_{tmp}\right) \triangleq mem_{tmp}$;
    $C_{curr} \leftarrow C_{curr} + \Delta C\left(f_{tmp}, mem_{tmp}\right)$;
    $ERT_{curr} \leftarrow ERT_{curr} + \Delta ERT\left(f_{tmp}, mem_{tmp}\right)$;
  else
    $t \leftarrow t - 1$;
  end
end
return π_{curr}

denoted as TH, is leveraged. The ERT/C strategy keeps a record of the BCR of the configuration in the previous critical path iteration, denoted as BCR_{pre}. Instead of simply maximizing BCR, the ERT/C strategy finds f_u and $mem_v \in MOpt(f_u)$ that results in the maximum BCR such that $BCR \geq BCP_{pre} \cdot TH$.

In the RT/M strategy, we introduce a BCR threshold TH and consider the two-point slope of the performance-memory curve as the BCR, namely

$$BCR\left(f_u, mem_v\right) = \frac{PF\left(f_u, mem_{v+1}\right) - PF\left(f_u, mem_v\right)}{mem_{v+1} - mem_v} \tag{22}$$

where $mem_{v+1} \in MOpt(f_u)$ is the adjacent selectable memory size such that $mem_{v+1} > mem_v$. For each function $f_i \in V$, the RT/M strategy algorithm first calculates the slope of the performance-memory curve by the least squares regression, denoted as $\beta\left(f_i\right)$. Then, the algorithm removes all memory options from the viable memory options whose BCR is smaller than $\beta\left(f_i\right) \cdot TH$. After this step, the memory option function satisfies $\beta\left(f_i\right) \cdot TH$ for all $1 \leq i \leq n$ and all $mem_j \in MOpt(f_i)$. In each critical path iteration, the algorithm finds the f_u and $mem_v \in MOpt(f_u)$ resulting in the greatest end-to-end response time reduction.

TABLE 2
Summary of BCR Greedy Strategies. It contains BCR, optimization goal in each iteration, and the conditions to which the optimization is subject.

Strategy	BCR Def.	Maximize	Subject To
		BPBC Problem	
W/O BCR	N/A	ERT Red.	None
MAX	eq. (21)	BCR	None
ERT/C	eq. (21)	BCR	$BCR \geq BCR_{pre} \cdot TH$
RT/M	eq. (22)	ERT Red.	$\text{BCR}\left(f_i, mem_j\right) \geq \beta\left(f_i\right) \cdot TH$
		BCPC Problem	
W/O BCR	N/A	Cost Red.	None
MAX	eq. (21) $^{-1}$	BCR	None
C/ERT	eq. (21) $^{-1}$	BCR	$BCR \geq BCR_{pre} \cdot TH$
M/RT	eq. (22) $^{-1}$	Cost Red.	$\text{BCR}\left(f_i, mem_j\right) \geq \beta\left(f_i\right) \cdot TH$

Correspondingly, MAX, C/ERT and M/RT are three BCR greedy strategies for the BCPC problem, where the BCR is defined as the multiplicative inverse of the BCR defined for the BPBC problem. In the M/RT strategy, the algorithm calculates the slope of the memory-performance curve. Table 2 gives the summary of the original PRPC algorithm and BCR greedy strategies. Besides the s.t. conditions mentioned in Table 2, as specified in the pseudocode of the PRPC algorithm, all strategies should also satisfy $\Delta C\left(f_u, mem_v\right) \leq BC - C_{curr}$ and $\Delta ERT\left(f_u, mem_v\right) < 0$ for the BPBC problem, and $\Delta ERT\left(f_u, mem_v\right) \leq PC - ERT_{curr}$ and $\Delta C\left(f_u, mem_v\right) < 0$ for the BCPC problem.

4.4.4 *Time Horizon of Modeling and Optimization*

The proposed models and optimization algorithms can be updated by new monitoring data depending on the degree of uncertainty in the underlying FaaS infrastructure and the serverless application itself. However, if the initial performance profiling phase has been done properly, the performance and cost models and optimal configurations should remain valid for a long time. Other reasons for updating the model will be changing the function configurations, the application architecture, function source code, or optimization constraints. From a practical point of view, these parameters are not expected to change frequently. Suppose any of the parameters mentioned above, except function source code, changes. In that case, users only need to rerun performance and cost models and optimization algorithms, which can be completed easily in a short time. If the function source code changes, updating the performance profile is required for the modified function, which is comparatively time-consuming and perhaps costly.

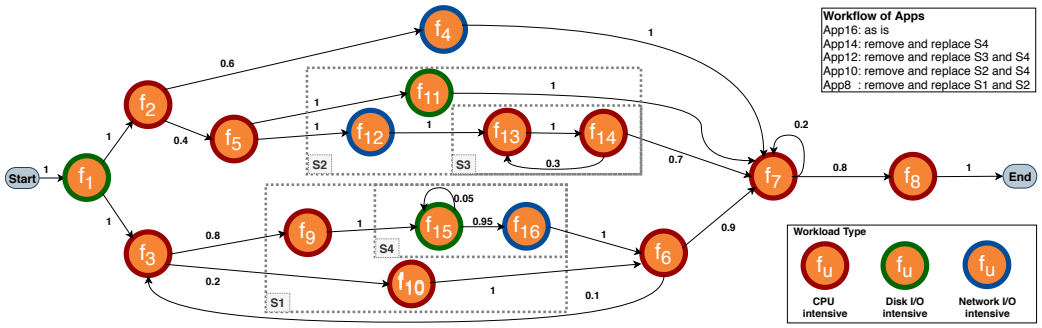

Fig. 8. Workflow of serverless applications used in the experimental evaluation of performance and cost models. The number on each edge represents the transition probability. The workflow of App16 is as shown. The workflow of other four Apps can be obtained by removing and replacing all edges and functions in corresponding box(es) with an edge whose transition probability is 1. The transition delay is defined using the delay model of AWS Step Functions. The applications are composed of functions with three different types of workload depicted by three colors.

5 Experimental Evaluation

We implement the performance and cost models and the PRCP algorithm using Python 3.8, and develop a ready-to-use script to help developers make use of them. We validate the performance and cost model by conducting experiments on six serverless applications deployed on AWS, and examine the performance of the PRCP algorithm using an application with six functions. All algorithms, scripts, and experimental results are available in the artifact repository.

5.1 Performance and Cost Models

5.1.1 *Experimental Design*

To evaluate performance and cost models, we design five serverless applications composed of various number of FaaS functions with mixed types of workload. We deploy functions on AWS Lambda and employ AWS Step Functions as the serverless application coordinator. AWS Step Functions is a serverless workflow coordination service that combines multiple Lambda functions and other serverless services offered by AWS into responsive serverless applications [20].

We design FaaS functions with three different types of workload, namely CPU intensive, disk I/O intensive, and network I/O intensive. The CPU-intensive workloads include string hashing, floating-point arithmetic, and recursive calculation. The disk-intensive workload is to write and read several files to the hard disk drive. The network-intensive workload is designed to download and upload a number of files from and to the AWS S3 bucket. We develop sixteen functions with different input sizes and types of workload and host them on AWS Lambda.

By leveraging AWS Step Functions, we develop five serverless applications using those sixteen functions, which are App8, App10, App12, App14, and App16. The numeric suffix of the application name represents the number of functions in the application. From App8 to App16, we increase the number of functions from eight to sixteen, as well as the number of structures (parallels, branches, cycles, and self-loops) in the application workflow. As a result, the complexity of the workflow grows accordingly. Besides, each application has a combination of all three types of workloads, making them truly representative of actual serverless applications on cloud. The workflow of five serverless applications are shown together in Figure 8.

We deploy sixteen functions on AWS Lambda and obtain their average duration under a feasible memory configuration by invoking each of them 720 times. Five aforementioned serverless applications are developed and deployed on AWS Step Functions. For repeatability and rigorousness, we follow the methodological principles proposed by Papadopoulos et al. [39] and execute repeated experiments and long runs. We execute each application for five periods of two hours with a two-hour interval between two consequent periods. During each period, each application is executed continuously with a 10-second interval between two consequent invocations. For each execution period, we discard executions in the first and last 10 minutes to avoid any transient fluctuations in performance and only adopt invocations between them. By doing so, for each application, logs of 3,000 invocations are ready for analysis.

5.1.2 *Experimental Result*

By giving the workflows and the average duration and allocated memory size of functions as inputs, we leverage the proposed performance and cost models to obtain the average end-to-end response time and cost of five serverless applications. By analyzing logs of 3,000 invocations for each application, we compare the results of performance and cost models with the duration and billing logs reported by AWS.

Figure 9 and Figure 10 illustrate the experimental evaluation result of the performance and cost models. As is evident from the figures, both the average end-to-end response time and cost derived by the proposed performance and cost models are very close to the real values reported by AWS. Regardless of the complexity of the workflow, the accuracy of predicted average end-to-end response time and cost is over 97.5%. Such results indicate the high accuracy of the proposed performance and cost models.

5.2 Performance and Cost Optimization

5.2.1 *Experimental Design*

To evaluate the proposed performance and cost optimization solution, namely the PRCP algorithm, we develop a serverless application named App6. As the name suggests, App6 is composed of 6 functions with three types of workload (CPU-intensive, disk-intensive, network-intensive). For

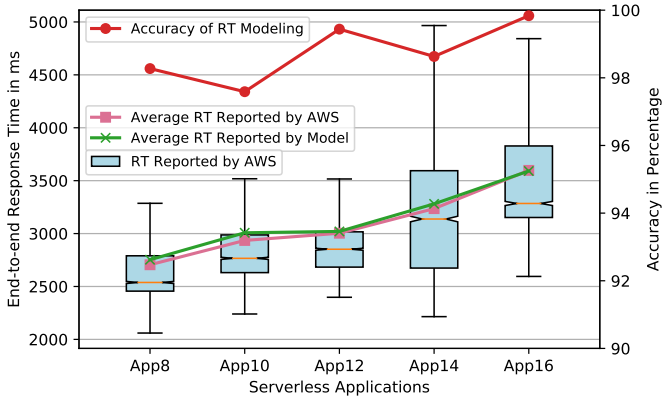

Fig. 9. Experimental evaluation result of the performance model. As the number of functions in the application increases from 8 to 16, the workflow become more complex in terms of structures. The average accuracy is 98.75%. The box plot shows the maximal value, 25%, 50%, 75% percentiles and the minimal value of RT. The notch shows the 95% confidence interval for the median of RT.

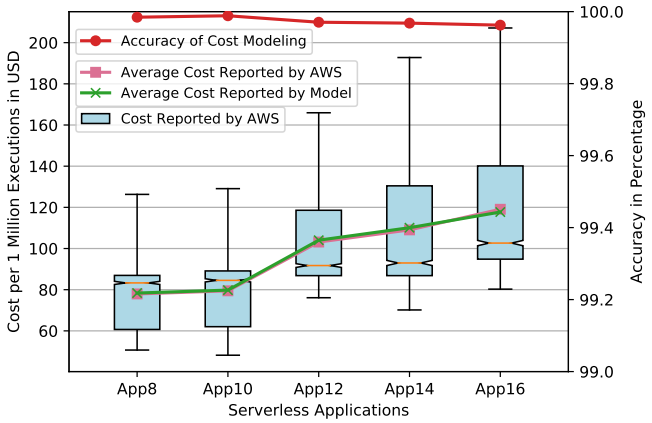

Fig. 10. Experimental evaluation result of the cost model. The average accuracy is 99.97%.

generality, App6 is designed to have all 4 types of structures, namely the parallel, branch, cycle, and self-loop.

As described in Section 4.1, to optimize the performance and cost of the serverless application, a performance profiling phase is required to get the feasible memory configuration $MOpt$ and performance profile PF of serverless functions in the application. We deploy functions on AWS Lambda, which allows the allocated memory to vary between 128 MB and 3,008 MB in 64MB increments, resulting in 46 possible choices. We stipulate that all 46 memory sizes are feasible choices for all functions. For each function, we obtain its performance profile by the performance profiling phase, during which the function is invoked 100 times under each feasible memory size, and the duration is logged.

In order to measure the accuracy of solutions given by the PRCP algorithm, an exhaustive search is necessary to obtain the performance and cost of App6 under all possible memory configurations. However, 6 functions with 46 possible memory choices lead to 9.47 billion states, making the exhaustive search computationally unfeasible. Therefore, we trim the number of memory choices while retain the trend of the response time-memory size curve by sampling the performance profile. After sampling, the viable memory size of each function varies between 128 MB and 3,008 MB

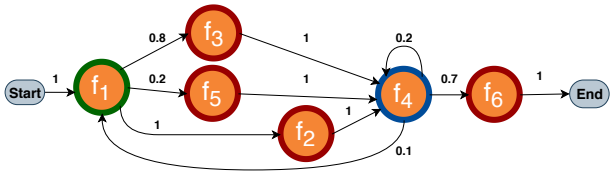

Fig. 11. The workflow of App6. The legend is same as Figure 8.

in 192 MB increments, 16 choices left. Using the proposed performance and cost models, we exhaustively obtain the average end-to-end response time and cost of App6 under 16,777,216 different memory configurations. As evidenced in Section 5.1.2, the performance and cost models can accurately give the average end-to-end response time and cost of the serverless application. Therefore, despite the fact that we do not perform the test on AWS Step Functions to get the performance and cost of App6 under 16,777,216 configurations, which is financially and practically unfeasible, the average end-to-end response time and cost derived from proposed models can be regarded as actual values.

We choose the series of 100 equidistant values between the minimum cost and the maximum cost as the budget constraints and execute the PRCP algorithm to solve the BPBC problem with four types of greedy strategies. Similarly, we use the series of 100 equidistant values between the minimum and the maximum end-to-end response time as the performance constraints and solve the BCPC problem. We compare the best performance and the best cost given by the PRCP algorithm with the actual value derived by the exhaustive search under each constraint value.

5.2.2 *Experimental Result*

Figure 11 shows the workflow of App6, and the performance profile of functions is illustrated as Figure 12. There are 1 parallel, 1 branch, 1 cycle, and 1 self-loop in App6.

Figure 13 depicts the best performance for the BPBC problem achieved by the PRCP algorithm with four greedy strategies. Considering the best performance among solutions given by four strategies, compared to the ideal value, the accuracy of the algorithm is calculated. For the 100 budget constraints, the average accuracy of the PRCP algorithm is 97.40%. Figure 14 illustrates the best cost achieved by the PRCP algorithm solving the BCPC problem. For the 100 performance constraints, the average accuracy of the PRCP algorithm is 99.63%. As is evident from Figure 13 and Figure 14, the accuracy of different greedy strategies varies with different budget and performance constraints. Therefore, the best method is to employ all four greedy strategies in the PRCP algorithm and select the best solution that is closest to the target performance or cost constraint.

6 Related Work

The serverless computing paradigm has attracted a great deal of interest from both academia and industry [8], [19], [40]. However, few research studies have been directed at the modeling and optimization of performance and cost of serverless applications. Therefore, we provide an extensive review of the literature in three fields, which are inherently related to our topic, namely the performance modeling of cloud applications, workflow scheduling in the cloud environment, and serverless architecture.

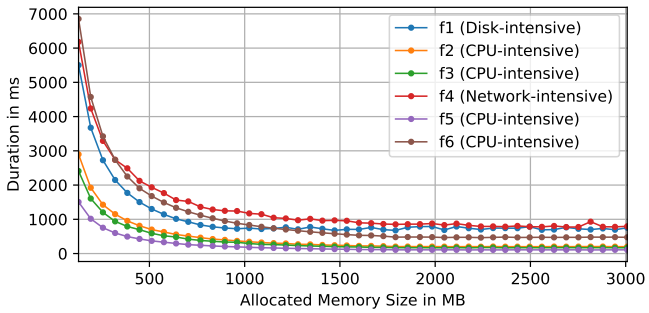

Fig. 12. The performance profile of 6 functions in App6. The allocated memory size varies between 128 MB and 3,008 MB in 64MB increments, resulting in 46 feasible memory choices.

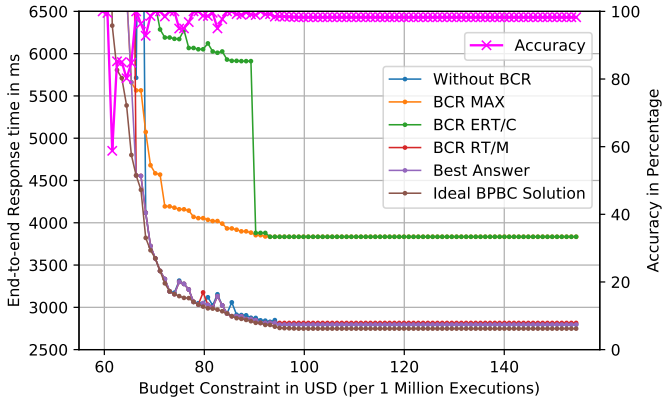

Fig. 13. The result of the PRCP algorithm solving the BPBC problem. BCR threshold is 0.2. The budget constraints are 100 equidistant values between $58.86 (minimum cost) and $163.90 (maximum cost). The average accuracy of the best answer is 97.40%. The best answer is the minimal average response time among solutions.

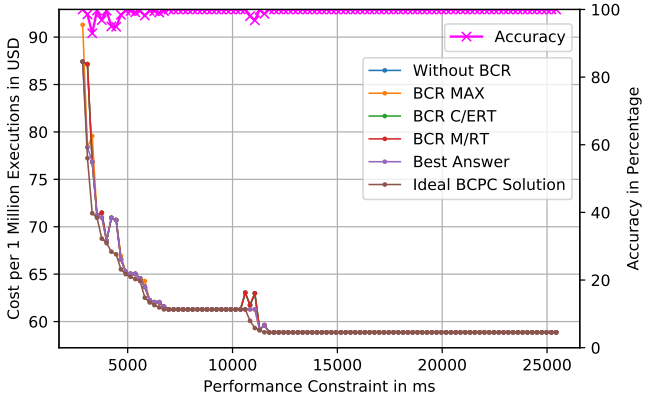

Fig. 14. The result of the PRCP algorithm solving the BCPC problem. BCR threshold is 0.2. The performance constraints are 100 equidistant values between 2748.24 ms (minimum ERT) and 25433.08 ms (maximum ERT). The average accuracy of the best answer is 99.63%. The best answer is the minimal average cost among solutions.

6.1 Performance models for cloud computing

The problem of modeling the performance of cloud architectures and applications has been extensively investigated over the past decade. Quality of service (QoS) requirements including response time, availability, and reliability, as well as monetary cost are the main focus of such models.

The queuing theory has been used to predict such QoS metrics for cloud systems. Khazaei et al. [41] obtained the distribution of response time and blocking probability with regards to the number of servers and buffer size for a cloud center using a semi-Markov $M/G/m/m+r$ queue, where r is the buffer size for incoming job. Using the task blocking probability and mean response delay as indicators, this work has been extended to predict the availability of cloud systems later [42]. Vilaplana et al. [43] utilized $M/M/1$ and $M/M/m$ queues to model a single entry server and processing nodes on cloud to get the total response time with regards to the service rate of processing nodes.

Petri net has also been proven to be an effective formalism for modeling distributed systems with the concurrency and synchronization. Chen et al. [25] leveraged a deterministic and stochastic Petri net to illustrate the performance of producer/consumer based application models in the cloud environment. Cao et al. [26] developed an evaluation model based on Queuing Petri Net to model the throughput of cloud systems with three different architectures. Rista et al. [27] introduced the generalized stochastic Petri net model, composed of the combination of timed and non-timed nets, to predict the throughput and latency of the network in container-based cloud environments.

All these works either assumed the cloud system is designed homogeneously or required the parameters relevant to the underlying resources and incoming requests, which are not reasonable enough for the serverless paradigm because of the infrastructure-agnostic platform, event-driven and highly decoupled software architecture, auto-scaling resources, and elimination of resources management. Besides, these approaches focused more on the performance analysis and did not model the cost. We fill these gaps by proposing a definition of the serverless workflow, and to the best of our knowledge, the first performance and cost models that are compatible with the current serverless paradigm.

6.2 Workflow scheduling in the cloud environment

In the past decade, researchers have extensively studied the workflow scheduling problem for workflow-based applications in the cloud environment. Abrishami et al. [37] presented a Partial Critical Paths algorithm to solve the best cost under the performance constraint problem for QoS-based workflows on the utility grid. Later, they extended the work by considering several cloud features such as the pay-as-you-go and duration-based billing model and applied the algorithm to a workflow instance on IaaS clouds to solve the same problem [38]. Lin et al. [33] proposed a Critical-Greedy algorithm to solve the best performance under the budget constraint problem for scientific applications. Faragardi et al. [44] proposed a Greedy Resource Provisioning with the consideration of heterogeneous cloud resources and the efficiency rate of instances to solve the best performance under the budget constraint problem for workflow applications on IaaS Clouds. Bao et al. [34] designed a Greedy Recursive Critical Path algorithm to find the configuration that achieves the best performance under the budget constraint for microservice-based applications on cloud.

This paper differs from the previous work at least in the following aspects: 1) We consider the new features in the serverless computing paradigm such as the memory-dependent performance rules, GB-second billing model,

independently operated components, and event-driven architecture. 2) The serverless workflow is not confined to DAGs, and cycles and self-loops are allowed to appear in the workflow. 3) We propose a Probability Refined Critical Path Algorithm to solve both BPBC and BCPC problems for serverless workflow based applications.

6.3 Serverless architecture

Many cloud service providers have launched their FaaS platform, including AWS Lambda, Google Cloud Functions, and Microsoft Azure Functions in recent years. Since then, serverless computing has become an emerging technique and research topic in both industry and academia. There have been efforts made to topics including developing new applications [45], [46], migrating to serverless [47], and designing new software engineering methodology [48].

There have been some works on the performance profiling of FaaS platforms. Jackson et al. [49] investigated the impact of programming languages on the duration and cost of FaaS functions. Wang et al. [14] evaluated the performance of functions hosted on three FaaS platforms, which increases with the allocated memory size. They also observed that the underlying infrastructure of platforms is heterogeneous and overhead caused by cold starts. Figiela et al. [15] conducted performance tests by deploying a serverless workflow on four FaaS platforms and measured the data transfer delay between components and the container lifespan.

Several works presented a set of key challenges for the serverless architecture. The challenge that many studies have mentioned is the unpredictable performance and cost due to the lack of performance and cost models [6], [7], [8], [19]. The literate available on solving such a challenge is, however, very limited. Cold start latency and high communication overhead between components are also common concerns. Jonas et al. [19] discussed inadequate storage, high communication overhead, high cold start latency, and lack of predictable performance and cost of applications. Shahrad et al. [50] found the containerization incurs huge overhead and cold start brings the latency as high as 10 times of a small function's execution time. Hellerstein et al. [51] investigated cold start, limited storage, and communication overhead by case studies in distributed computing.

In this paper, we solve the unpredictable performance and cost problem for serverless applications with performance and cost models. We propose a serverless workflow with the consideration of the communication latency, and the cold start latency can be reflected in the model in a plug-and-play manner. We also develop a heuristic algorithm to solve the performance-cost trade-off problem.

7 Conclusion

In this work, we answered two research questions regarding the modeling and optimization of performance and cost of serverless applications. We laid out a formal definition of the serverless workflow with the consideration of several serverless features on clouds. Then we solved the unpredictable performance and cost problems for serverless applications by proposing the performance and cost models. The proposed models could accurately estimate the average end-to-end response time and cost of serverless applications; we checked the validity of the proposed models by extensive evaluation of five serverless applications deployed on Amazon AWS. Also, we solved two optimization problems, namely the best performance under the budget constraint and the best cost under the performance constraint. We answered them by proposing a heuristic algorithm named Probability Refined Critical Path algorithm with four greedy strategies. Again, we verified the validity of the proposed algorithms through experimental evaluations on AWS.

References

[1] M. Armbrust and A. Griffith, "A berkeley view of cloud computing," *UC Berkeley EECS Technical Report EECS-2009-28*, 2009.

[2] P. K. Senyo, E. Addae, and R. Boateng, "Cloud computing research: A review of research themes, frameworks, methods and future research directions," *International Journal of Information Management*, vol. 38, no. 1, pp. 128–139, 2018.

[3] B. Varghese and R. Buyya, "Next generation cloud computing: New trends and research directions," *Future Generation Computer Systems*, vol. 79, pp. 849–861, 2018.

[4] R. Vemula, "A new era of serverless computing," in *Integrating Serverless Architecture*, pp. 1–22, Springer, 2019.

[5] "Financial engines cuts costs 90% using aws lambda and serverless computing." https://aws.amazon.com/solutions/case-studies/financial-engines/, 2018. [Online; accessed February-2-2020].

[6] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and Others, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*, pp. 1–20, Springer, 2017.

[7] A. Eivy, "Be wary of the economics of" serverless" cloud computing," *IEEE Cloud Computing*, vol. 4, no. 2, pp. 6–12, 2017.

[8] E. Van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uță, and A. Iosup, "Serverless is more: From paas to present cloud computing," *IEEE Internet Computing*, vol. 22, no. 5, pp. 8–17, 2018.

[9] E. Van Eyk, A. Iosup, S. Seif, and M. Thömmes, "The spec cloud group's research vision on faas and serverless architectures," in *Proceedings of the 2nd International Workshop on Serverless Computing*, pp. 1–4, 2017.

[10] H. Fingler, A. Akshintala, and C. J. Rossbach, "Usetl: Unikernels for serverless extract transform and load why should you settle for less?," in *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, pp. 23–30, 2019.

[11] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "Sand: Towards high-performance serverless computing," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 923–935, 2018.

[12] "Release: Aws lambda on 2014-11-13." https://aws.amazon.com/releasenotes/release-aws-lambda-on-2014-11-13/, 2014. [Online; accessed February-2-2020].

[13] "Aws lambda limits." https://docs.aws.amazon.com/lambda/latest/dg/limits.html, 2020. [Online; accessed February-2-2020].

[14] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 133–146, 2018.

[15] K. Figiela, A. Gajek, A. Zima, B. Obrok, and M. Malawski, "Performance evaluation of heterogeneous cloud functions," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 23, p. e4792, 2018.

[16] "New–provisioned concurrency for lambda functions." https://aws.amazon.com/blogs/aws/new-provisioned-concurrency-for-lambda-functions/, 2019. [Online; accessed February-2-2020].

[17] "Aws lambda pricing." https://aws.amazon.com/lambda/pricing/, 2020. [Online; accessed February-2-2020].

[18] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pp. 405–410, IEEE, 2017.

[19] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[20] "Aws step functions." https://aws.amazon.com/step-functions/, 2020. [Online; accessed February-2-2020].

[21] "Create a serverless workflow with aws step functions and aws lambda." https://aws.amazon.com/getting-started/tutorials/create-a-serverless-workflow-step-functions-lambda/, 2020. [Online; accessed February-2-2020].

[22] "Ad hoc big data processing made simple with serverless mapreduce." https://aws.amazon.com/blogs/compute/ad-hoc-big-data-processing-made-simple-with-serverless-mapreduce/, 2016. [Online; accessed February-4-2020].

[23] "Serverless reference architecture: Image recognition and processing backend." https://github.com/aws-samples/lambda-refarch-imagerecognition, 2016. [Online; accessed February-4-2020].

[24] "Serverless application lens aws well-architected framework." https://d1.awsstatic.com/whitepapers/architecture/AWS-Serverless-Applications-Lens.pdf, 2019. [Online; accessed February-4-2020].

[25] H. Chen, C. Zhou, Y. Qin, A. Vandenberg, A. V. Vasilakos, and N. Xiong, "Petri net modeling of the reconfigurable protocol stack for cloud computing control systems," in *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pp. 393–400, IEEE, 2010.

[26] Y. Cao, H. Lu, X. Shi, and P. Duan, "Evaluation model of the cloud systems based on queuing petri net," in *International Conference on Algorithms and Architectures for Parallel Processing*, pp. 413–423, Springer, 2015.

[27] C. Rista, M. Teixeira, D. Griebler, and L. G. Fernandes, "Evaluating, estimating, and improving network performance in container-based clouds," in *2018 IEEE Symposium on Computers and Communications (ISCC)*, pp. 00514–00520, IEEE, 2018.

[28] L. Versluis, E. Van Eyk, and A. Iosup, "An analysis of workflow formalisms for workflows with complex non-functional requirements," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pp. 107–112, 2018.

[29] A. Kumar and S. Mahendrakar, *Serverless Integration Design Patterns with Azure: Build powerful cloud solutions that sustain next-generation products*. Packt Publishing, 2019.

[30] "Aws samples." https://github.com/aws-samples/, 2020. [Online; accessed March-16-2020].

[31] "Azure samples." https://github.com/Azure-Samples, 2020. [Online; accessed April-4-2020].

[32] "Best practices for working with aws lambda functions." https://docs.aws.amazon.com/lambda/latest/dg/best-practices.html, 2020. [Online; accessed February-12-2020].

[33] X. Lin and C. Q. Wu, "On scientific workflow scheduling in clouds under budget constraint," in *2013 42nd International Conference on Parallel Processing*, pp. 90–99, IEEE, 2013.

[34] L. Bao, C. Wu, X. Bu, N. Ren, and M. Shen, "Performance modeling and workflow scheduling of microservice-based applications in clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, pp. 2114–2129, Sep. 2019.

[35] O. Goldreich, *P, NP, and NP-Completeness: The basics of computational complexity*. Cambridge University Press, 2010.

[36] J. E. Kelley Jr, "Critical-path planning and scheduling: Mathematical basis," *Operations research*, vol. 9, no. 3, pp. 296–320, 1961.

[37] S. Abrishami, M. Naghibzadeh, and D. H. Epema, "Cost-driven scheduling of grid workflows using partial critical paths," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 8, pp. 1400–1414, 2011.

[38] S. Abrishami, M. Naghibzadeh, and D. H. Epema, "Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds," *Future Generation Computer Systems*, vol. 29, no. 1, pp. 158–169, 2013.

[39] A. V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. Von Kistowski, A. Ali-eldin, C. Abad, J. N. Amaral, P. Tma, and A. Iosup, "Methodological principles for reproducible performance evaluation in cloud computing," *IEEE Transactions on Software Engineering*, 2019.

[40] T. Lynn, P. Rosati, A. Lejeune, and V. Emeakaroha, "A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 162–169, IEEE, 2017.

[41] H. Khazaei, J. Misic, and V. B. Misic, "Performance analysis of cloud computing centers using m/g/m/m+r queuing systems," *IEEE Transactions on parallel and distributed systems*, vol. 23, pp. 936–943, May 2011.

[42] H. Khazaei, J. Mišić, V. B. Mišić, and N. B. Mohammadi, "Availability analysis of cloud computing centers," in *2012 IEEE Global Communications Conference (GLOBECOM)*, pp. 1957–1962, IEEE, 2012.

[43] J. Vilaplana, F. Solsona, I. Teixidó, J. Mateo, F. Abella, and J. Rius, "A queuing theory model for cloud computing," *The Journal of Supercomputing*, vol. 69, no. 1, pp. 492–507, 2014.

[44] H. R. Faragardi, M. R. S. Sedghpour, S. Fazliahmadi, T. Fahringer, and N. Rasouli, "Grp-heft: A budget-constrained resource provisioning scheme for workflow scheduling in iaas clouds," *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[45] A. Pérez, S. Risco, D. M. Naranjo, M. Caballer, and G. Moltó, "On-premises serverless computing for event-driven data processing applications," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pp. 414–421, IEEE, 2019.

[46] Á. L. García, J. M. De Lucas, M. Antonacci, W. Zu Castell, M. David, M. Hardt, L. L. Iglesias, G. Moltó, M. Plociennik, V. Tran, *et al.*, "A cloud-based framework for machine learning workloads and applications," *IEEE Access*, vol. 8, pp. 18681–18692, 2020.

[47] V. Yussupov, U. Breitenbücher, F. Leymann, and C. Müller, "Facing the unplanned migration of serverless applications: A study on portability problems, solutions, and dead ends," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, pp. 273–283, 2019.

[48] F. Samea, F. Azam, M. W. Anwar, M. Khan, and M. Rashid, "A uml profile for multi-cloud service configuration (umlpmsc) in event-driven serverless applications," in *Proceedings of the 2019 8th International Conference on Software and Computer Applications*, pp. 431–435, 2019.

[49] D. Jackson and G. Clynch, "An investigation of the impact of language runtime on the performance and cost of serverless functions," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pp. 154–160, IEEE, 2018.

[50] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 1063–1075, 2019.

[51] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," *arXiv preprint arXiv:1812.03651*, 2018.

Changyuan Lin (Student Member, IEEE) is an M.Sc. student in Software Engineering and Intelligent Systems at the Department of Electrical and Computer Engineering, University of Alberta, Edmonton, AB, Canada. He received his B.E. degree in Microelectronics from East China Normal University, Shanghai, China in 2018. He is a research assistant at the University of Alberta and a visiting research assistant in the Performant and Available Computing Systems Lab at York University, Toronto, ON, Canada. His research interests include cloud computing, serverless architecture, performance modeling, and software engineering.

Hamzeh Khazaei (Member, IEEE) is an assistant professor in the Department of Electrical Engineering and Computer Science at York University. Previously he was an assistant professor at the University of Alberta, a research associate at the University of Toronto and a research scientist at IBM, respectively. He received his PhD degree in Computer Science from the University of Manitoba, where he extended queuing theory and stochastic processes to accurately model the performance and availability of cloud computing systems. His research interests include performance modelling, cloud computing and engineering distributed systems.