Astrea: Auto-Serverless Analytics Towards Cost-Efficiency and QoS-Awareness

Jananie Jarachanthan, Li Chen, *Member, IEEE*, Fei Xu, *Member, IEEE*, and Bo Li, *Fellow, IEEE*

Abstract—With the ability to simplify the code deployment with one-click upload and lightweight execution, serverless computing has emerged as a promising paradigm with increasing popularity. However, there remain open challenges when adapting data-intensive analytics applications to the serverless context, in which users of *serverless analytics* encounter the difficulty in coordinating computation across different stages and provisioning resources in a large configuration space. This paper presents our design and implementation of *Astrea*, which configures and orchestrates serverless analytics jobs in an autonomous manner, while taking into account flexibly-specified user requirements. *Astrea* relies on the modeling of performance and cost which characterizes the intricate interplay among multi-dimensional factors (e.g., function memory size, degree of parallelism at each stage). We formulate an optimization problem based on user-specific requirements towards performance enhancement or cost reduction, and develop a set of algorithms based on graph theory to obtain the optimal job execution. We deploy *Astrea* in the AWS Lambda platform and conduct real-world experiments over representative benchmarks, including Big Data analytics and machine learning workloads, at different scales. Extensive results demonstrate that *Astrea* can achieve the optimal execution decision for serverless data analytics, in comparison with various provisioning and deployment baselines. For example, when compared with three provisioning baselines, *Astrea* manages to reduce the job completion time by 21% to 69% under a given budget constraint, while saving cost by 20% to 84% without violating performance requirements.

Index Terms—Cloud computing, serverless computing, resource provisioning, modeling, optimization

1 Introduction

Serverless computing has gained its popularity due to its compelling properties of lightweight runtime, ease of management, high elasticity and fine-grained billing. With serverless architectures, which facilitate Function-as-a-Service (FaaS) in cloud computing, developers are able to concentrate only on the logic, free from the burden of configuring environments, managing virtual machine (VM) clusters and paying for VM instances even though they are idle. Such a favorable computation mode has been deployed by cloud providers such as Amazon Lambda [1], Google Cloud Functions [2], and Microsoft Azure Functions [3], widely utilized in applications such as real-time video encoding [4], Internet-of-Things applications [5], interactive data analytics [6], and *etc.*

- *Jananie Jarachanthan and Li Chen are with the School of Computing and Informatics, University of Louisiana at Lafayette, Lafayette, LA 70504 USA. E-mail: {jananie.jarachanthan1, li.chen}@louisiana.edu.*
- *Fei Xu is with the School of Computer Science and Technology, East China Normal University, Shanghai 200050, China. E-mail: fxu@cs.ecnu.edu.cn.*
- *Bo Li is with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong, China. E-mail: bli@cse.ust.hk.*

Manuscript received 21 Sept. 2021; revised 27 Apr. 2022; accepted 27 Apr. 2022. Date of publication 3 May 2022; date of current version 26 July 2022.
This work was supported in part by National Science Foundation under Grant OIA-2019511, in part by the Louisiana Board of Regents under Grant LEQSF (2019-22)-RD-A-21, in part by NSFC under Grant 61972158, in part by the Science and Technology Commission of Shanghai Municipality under Grants 20511102802 and 18DZ2270800, in part by RGC RIF under Grant R6021-20, and in part by RGC GRF under Grants 16209120 and 16200221.
(Corresponding author: Li Chen.)
Recommended for acceptance by J. Carretero Perez.
Digital Object Identifier no. 10.1109/TPDS.2022.3172069

However, when adapting data-intensive analytics applications (e.g., MapReduce, Spark jobs) to serverless platforms, there have emerged a number of challenges, and one particular challenge is how to efficiently process the massive amount of intermediate data, also referred to as ephemeral data in contrast to the persistent input and output data. Such intermediate data requires to be shared between stateless functions in different stages. For instance, unlike the traditional VM-to-VM or server-to-server transmission of intermediate data in the MapReduce shuffle phase, function-to-function networking in serverless platforms does not support bulk data transfer, aligned with the original design philosophy of serverless computing. Consequently, a mapper function needs to output the intermediate data into the external storage, such as the object store S3 [7] or the distributed cache Redis [8], to be later fetched as the input for reducer functions. The cost and latency imposed by the ephemeral data sharing above raise serious application performance and cost efficiency issues in utilizing the serverless analytics.

Existing efforts have proposed a number of ephemeral data storage solutions for serverless analytics ([6], [9], [10], [11], *etc.*). For example, Pocket [9] is designed and implemented as a distributed storage system shared by serverless jobs, which places data across multiple tiers of storage to offer high-throughput and low-latency services. Locus [6], a data analytics framework customized for the serverless environment, orchestrates the shuffling of intermediate data in a serverless MapReduce job, leveraging a hybrid of fast and slow storage.

Despite these research efforts, there is no general guidance on the coordination and resource provisioning for serverless analytics among the large configuration space, including the

1045-9219 © 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: East China Normal University. Downloaded on December 15,2022 at 03:36:29 UTC from IEEE Xplore. Restrictions apply.

memory size of each function, the degrees of parallelism in each computation stage, and *etc.* Cloud users may easily deploy their serverless analytics suboptimally, at the risk of violating their Quality of Service (QoS) objectives (e.g., responding within a latency threshold) [12] or incurring extra billing costs which could have been avoided by a better configuration. Essentially, users still encounter the critical challenge of serverless provisioning for Big Data analytics (as evidenced in Section 2.3): *given a large configuration space and different types of user requirements (latency-oriented or budget-driven), how could users take advantage of the salient features of serverless computing without concerning about the underlying complexities (ephemeral data management and resource configuration), while achieving the maximum gain with respect to the performance or cost?* More specifically, how to achieve the best possible job performance with a limited budget, and how to minimize the cost without violating the QoS objective? To address this challenge, we argue that a general framework, in the middle of developers for data analytics and cloud providers for FaaS, needs to be built, to judiciously handle the job deployment and hide the underlying complexity. A comprehensive solution is expected to optimally configure and orchestrate the serverless analytics jobs in an autonomous manner, according to the flexibly-specified requirements from users.

To this end, we design a general framework, called *Astrea*, which automatically configures and orchestrates lambda functions for data analytics and machine learning jobs to navigate the tradeoff between performance and cost.

- First, *Astrea* derives mathematical models for both the monetary cost and the job completion time for a job upon submission, based on the user-specific objectives. The configurations characterized in the models include the number of stages in the job workflow, the degree of parallelism in each stage (i.e., the number of lambda functions in each stage), and the type of lambda function (i.e., the memory size of the requested lambda), which are coupled with the orchestration of all the functions invoked for a job.
- Second, building upon the model, *Astrea* obtains the optimal job execution plan based on the graph theory. Specifically, we construct two Directed Acyclic Graphs (DAGs) models for the completion time and the monetary cost, respectively, to formulate two optimization problems: (1) given a budget constraint, a configuration and job execution optimization is formulated with the objective of minimizing the job completion time, (2) under a Quality of Service (QoS) requirement, a configuration and job execution optimization is formulated with the objective of minimizing monetary cost.
- Third, upon the submission of an analytics job, *Astrea* calculates the optimal configuration for resource allocation and task assignment by solving our formulated optimization problem towards a specific objective. Given the limitations on the maximum concurrency per job and the maximum temporary storage per function on today's serverless platforms [13], it is likely that a large data analytics job does not have any feasible solution for direct deployment (as evidenced in Section 4.3). Therefore, we have designed an extended version of *Astrea* to accommodate large jobs through judicious multi-round executions.

Finally, we have implemented and deployed *Astrea* on AWS Lambda and evaluated its performance with real-world experiments on various workloads of Big Data analytics and machine learning at different scales, including Wordcount (1 GB, 10 GB, and 20 GB), Sort (100 GB), various queries over the Uservisits dataset (25.4 GB and 126.8 GB) and Rankings dataset (6.38 GB) [14], K-nearest neighbors classification (10 GB), and K-means clustering (10 GB). We compared *Astrea* with the following provisioning and deployment solutions on AWS: i) three provisioning baselines on AWS Lambda, ii) Apache Spark [15] on Amazon EC2 [16] and SageMaker [17], respectively, and iii) Amazon Elastic MapReduce (EMR) [18].

Extensive experimental results have demonstrated that *Astrea* can optimize the job performance (i.e., minimize the completion time) constrained by a budget, and minimize the monetary cost without violating a performance requirement. Compared with three baselines, *Astrea* achieves the performance improvement of about 42% to 69% for Wordcount benchmarks with three different input sizes, up to 21% for Sort, and 57% improvement for Query over Uservisits dataset, and at least 29% for other queries workloads. With respect to the monetary cost, a reduction up to 80% is achieved for Wordcount, up to 21% reduction for Sort, at least 42% reduction for Query over Uservisits dataset, and up to 84% reduction for other queries. Compared to Spark on EC2 for Wordcount and queries workloads, *Astrea* saves cost by at least 92% while achieving a similar or up to 2X better job performance. Compared to Spark on SageMaker for machine learning jobs, *Astrea* can achieve up to 97% cost reduction while completing jobs at least 36% faster. For the comparison with EMR, a commercial VM-based MapReduce platform, *Astrea* exhibits up to 77% faster completion and 65% cost saving for Wordcount and Sort workloads. Our extended version of *Astrea* has also been demonstrated to effectively deploy large jobs through multiple rounds, yielding both cost reduction and performance improvement compared to Apache Spark on EC2. As evidenced, *Astrea* successfully navigates the tradeoff between job completion time and monetary cost according to flexible requirements, outperforming the existing solutions which are either suboptimal or incomplete.

The rest of the paper is organized as follows. Section 2 presents the background of serverless analytics, examines the intricate interplay among cost and performance factors, and compares our proposed solution with the related works. Section 3 models the performance and monetary cost for a MapReduce job in the serverless platform. Section 4 designs algorithms for *Astrea* to optimize job completion time or monetary cost. Section 5 implements *Astrea* and demonstrates its advantages over three baselines with real-world experiments. Finally, Section 6 presents concluding remarks and future directions.

2 Background, Related Works and Motivation

In this section, we present the background and related work of serverless computing, with a particular focus on data

Authorized licensed use limited to: East China Normal University. Downloaded on December 15,2022 at 03:36:29 UTC from IEEE Xplore. Restrictions apply.

analytics. Having observed the current limitations in serverless analytics, we seek to understand the application performance (i.e., job completion time) of Big Data analytics running in the serverless platform, as well as the incurred monetary cost.

2.1 Serverless Computing for the Next Generation of Cloud

Serverless computing has recently emerged as a popular computing pattern in cloud computing, facilitating higher-level and finer-grained Function-as-a-Service (FaaS) to cloud users. With serverless platform, users simply upload their code and dependencies with a click on a button, and pay for the total runtime of their computation. Compared with traditional cloud computing by renting VMs, users in serverless computing no longer deal with the deployment and maintenance complexities, and no longer pay for VM runtime when there is no computation workload.

Due to the favorable properties of serverless computing, the past several years have witnessed a wide array of real-world applications transformed and deployed in the serverless environment, including data analytics [19], software compilation [20], machine learning [21], and *etc.* For example, a data analytics application has been implemented on serverless infrastructure, which can process real-time data from various sources with serverless functions and generate analytical results in real time to the user [22]. FaaS targets fine-grained use scenarios, whereas massive data featured scenarios can be the opposite. Still, in such scenarios where analytics applications require concurrent handling of massive data, it has been shown that serverless deployment is promising [6], [9].

2.2 Serverless Analytics in AWS Lambda

Facilitated by cloud service providers, serverless functions, such as AWS Lambda [1], Google Cloud Functions [2] and Microsoft Azure Functions [3], are essential in serverless computing. With AWS Lambda as an example, a user uploads the code, which will be scaled and executed by the serverless infrastructure transparent to the user. The default limit for concurrent executions is a maximum of 1,000 lambdas, 512 MB of temporary storage and 900 seconds of timeout [13], which makes it challenging to accommodate large-scale data analytics in the serverless environment [6]. The state-of-the-art serverless implementation for the MapReduce framework leverages S3 as the remote storage for intermediate data and AWS Lambda as the computation environment for mapping and reducing [23]. This framework uses three types of lambda functions, namely the mapper, the coordinator and the reducer. Concurrent mappers and asynchronous reducers will communicate through a coordinator, which calculates the numbers of objects to be handled by the mappers and the reducers, and the number of steps required in the reducing phase, based on the memory limit of each function.

However, there is no general guidance on the coordination and resource allocation for serverless analytics. Among the large space of design, including the type of function memory, the degrees of parallelism in each computation phase, *etc.*, cloud users may easily specify suboptimal deployment for their serverless analytics, at the risk of violating their QoS objectives (e.g., responding within a latency threshold), or incurring extra billing cost which could have been avoided by a better configuration. Essentially, the problem is that the user still has to deal with the complexities of resource configuration, which compromises the salient features of serverless computing.

TABLE 1
Partial Orchestration of a MapReduce Job for 10 Input Objects Used in Motivation Experiments in AWS Lambda

number of objects per mapper	1	2	3	4	5
number of mappers	10	5	4	3	2
number of objects per reducer	1	2	3	4	5
step 1 (number of reducers)	1	3	2	1	1
step 2 (number of reducers)	-	2	1	-	-
step 3 (number of reducers)	-	1	-	-	-
step 4 (number of reducers)	-	-	-	-	-

Therefore, we are motivated to provide a framework that takes over the challenging tasks and hides complexities from users, so that well-planned orchestration and optimal resource configuration could be generated according to the user-specific concern about application performance and monetary cost. In what follows, we will use an experimental example to illustrate and analyze the important factors impacting performance and cost, which will be further characterized in our modeling.

2.3 Factors Impacting Performance and Cost

With MapReduce on AWS Lambda as a simple example, we next present the completion time and monetary cost of the job given different configurations, to understand the key factors in the workflow that impact cost and performance. This job is implemented with three types of lambda functions as mapper, coordinator and reducer, following the framework aforementioned [23], with a total of 10 objects with 2 MB total size in S3 as input data. Based on the total amount of data to be processed by each lambda function, i.e., the number of objects in this setting, the job can be executed with different degrees of parallelism.

Table 1 presents five orchestration examples, when we vary the number of objects handled by each function. For instance, as shown in the second column entry in Table 1, given that each mapper processes 2 objects, a total of 5 mapper functions will be invoked to process the 10 input objects, which generate 5 objects as intermediate data. Then, given that each reducer handles 2 objects, 3 reducers will be launched in step 1, and their output of 3 objects will be further processed by 2 reducer lambdas in step 2. Finally, a reducer lambda reads the 2 objects from the previous step and generates the final result in step 3.

The resource allocation for a lambda function mainly refers to the memory allocation, which can be specified from 128 MB to 3008 MB in 64 MB increments in AWS Lambda.[1] The monetary cost incurred by a lambda function depends on the duration of the lambda, which is impacted by the memory allocation, and the PUT and GET requests made from the lambda [24].

1. Platform resource limitations can be updated over time. Currently, the memory size limit on AWS Lambda is increased to 10,240 MB.

Authorized licensed use limited to: East China Normal University. Downloaded on December 15,2022 at 03:36:29 UTC from IEEE Xplore. Restrictions apply.

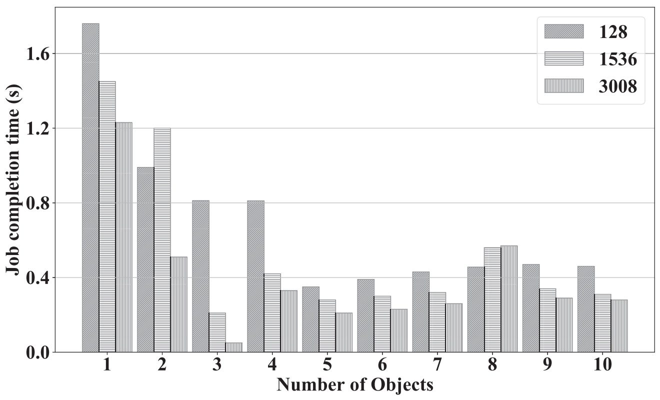


Fig. 1. Job completion time with the number of objects processed per lambda in three types of memory allocation.

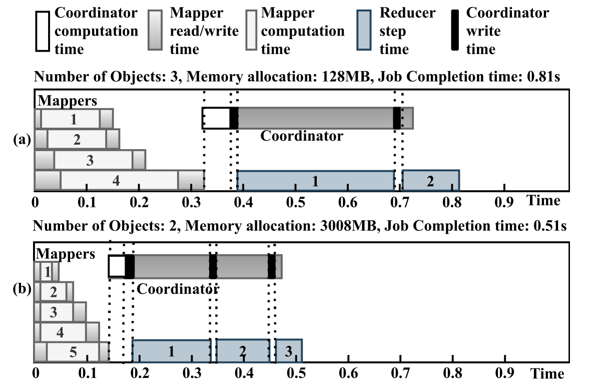

Coordinator computation time
Mapper read/write time
Mapper computation time
Reducer step time
Coordinator write time
Number of Objects: 3, Memory allocation: 128MB, Job Completion time: 0.81s
Mappers
Coordinator
(a)
Number of Objects: 2, Memory allocation: 3008MB, Job Completion time: 0.51s
(b)
Time

Fig. 3. Job timeline with two sample configurations.

Figs. 1 and 2 illustrate the experimental results of the job performance and cost, when we alternate the lambda orchestration (with different number of objects processed per lambda) and the memory allocation, respectively. The job performance relies on the completion time, which is impacted by the number of steps for reducing, and when the slowest reducer finishes in each step. The runtime of each lambda relies on both its computation time and the network transfer time when reading from and writing to S3. With respect to the cost, the job consists of the lambda invocation cost, lambda runtime cost, S3 storage cost and S3 request cost, which depend on the number and size of objects, and the number and memory type of lambdas.

As observed in the figures, when we vary the configuration setting, there is a complicated interplay among multiple factors that collectively determine the final job completion time and cost. More specifically, when the number of objects per lambda increases from 1 to 4, the job completion time exhibits a decreasing trend as shown in Fig. 1. This is because with each lambda processing more data, the number of sequential reducer steps will decrease. Although each lambda takes a bit longer to process more data, the time reduction on the reducing phase dominates, leading to faster job completion. Similarly, with each function processing more data (the number of objects increasing from 1 to 4), the total number of lambdas becomes smaller and the number of S3 read/write decreases, resulting in the cost reduction as shown in Fig. 2. When increasing the number of objects beyond 5, the number of lambdas and the coordination no longer change, but the data distribution among lambdas becomes more skewed. For example, the numbers of objects processed by mappers become (5,5), (6,4), (7,3), (8,2) and (9,1), when the number of objects per lambda is set from 5 to 9. The skewness will cause unbalanced computation time and data transfer time, prolonging the completion time and increasing the cost, as observed.

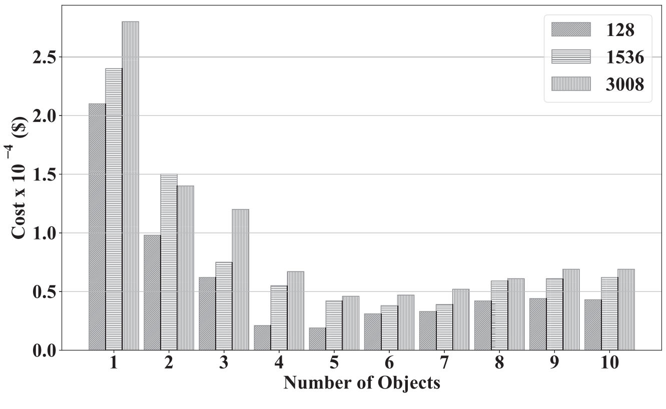


Fig. 2. Monetary cost with the number of objects processed per lambda in three types of memory allocation.

We further present a microscopic analysis by decomposing the job completion time in Fig. 3, with two sample configurations. The mapping phase completes when the slowest mapper finishes, and the coordinator is then launched to coordinate the reducing phase. When each lambda function handles 3 objects with 128 MB memory, there will be 4 mappers according to Table 1, followed by two steps of reducing, each with 2 and 1 reducer lambda(s), respectively. The second configuration sets the number of objects as 2 and memory as 3008 MB for each lambda, resulting in 5 mappers followed by 3, 2 and 1 reducer(s) in three consecutive steps. Although the number of reducer steps increases from Figs. 3(a) to 3(b), each function with the largest memory block is much faster, which eventually leads to a shorter job completion time.

Even with such a toy example with an incomplete exploration in the configuration space, we have witnessed the intricate interplay among multi-dimensional factors. Clearly, it is challenging for cloud users to identify the best configuration according to their flavor on performance boost or cost saving. In this paper, we argue that cloud users should be hidden from such complexities to completely enjoy the ease of management burden. Therefore, we are motivated to design and implement a framework, called *Astrea*, to automate the deployment of serverless analytics in an optimal manner according to user-specific requirements. In particular, we hope to explore the whole design space of the coupled orchestration and configuration of lambdas, to seek optimal solutions regarding improving latency performance and reducing monetary cost. With the knowledge of key factors, we will leverage mathematical modeling to characterize the interdependencies in the next section, and formulate the optimization problems with flexible objectives and constraints. Moreover, due to the serverless platform limitations, such as the maximum allowable function concurrency and the cap on the temporary storage size of a function [13], it is challenging to deploy a large-scale data analytics job on the serverless platform, to be further elaborated in Section 4.3. To fill this gap,

Authorized licensed use limited to: East China Normal University. Downloaded on December 15,2022 at 03:36:29 UTC from IEEE Xplore. Restrictions apply.

we are further motivated to extend our methodology and framework design to accommodate large-scale workloads.

2.4 Related Works

In this subsection, we summarize the existing efforts and discuss our proposed framework in comparison.

Data Analytics on Serverless Platforms. Although a number of applications have easily and successfully transitioned into the serverless environment (e.g., [4]), there still remain open challenges for data analytics jobs to be implemented with serverless architecture, due to their heavy demand for the storage of intermediate data [6], [21].

PyWren [10] presents a prototype to execute MapReduce jobs with lambda functions, using S3 as intermediate data storage. Similarly, Flint [25] enhances the PySpark MapReduce framework in the serverless environment, and leverages the Amazon Simple Queue Service (SQS) [26] for the shuffling of intermediate data. Extending the idea of Elastic MapReduce (EMR) [18], Amazon AWS presented a serverless architecture [23] for MapReduce jobs with S3 as intermediate storage. MARLA [11] follows the same architecture and handles the invoking of multiple mapper lambdas in a different way. Kappa [27] is a programming framework that simplifies serverless development for general computing. With the help of checkpointing and FIFO queues, Kappa's concurrency API provides mechanisms for launching and synchronizing parallel tasks. Wukong [19] and NIMBLE [28] investigate task-level scheduling for cost-efficiency and performance improvement in serverless analytics. Wukong [19] adopts decentralized scheduling and task clustering in its serverless framework that can reduce data movement over the network and improve cost effectiveness. NIMBLE [28] further pipelines the task execution by considering task-level dependencies.

On the other hand, there are research efforts on enhancing the intermediate data storage. Pocket [9] uses EC2 VMs as ephemeral storage, enables auto-scaling and provides pay-per-use service to cloud functions. Jiffy [29] further enables fine-grained far-memory sharing and multiplexing across concurrent jobs. Locus [6] leverages a small number of expensive fast ElastiCache (Redis) [8] instances combined with the much cheaper S3 service. Gadepalli, *et al.* [30] applied serverless computing at the edge at near-native speed, while having a small memory footprint and optimized invocation time. Lambada [31] supports function-to-function communication through different types of shared serverless storage like Amazon S3, Amazon DynamoDB, and Amazon SQS depending on the data size. InfiniCache [32] presents the first in-memory object cache for serverless functions to improve I/O performance. Amoeba [33] switches between the IaaS (Infrastructure-as-a-Service)-based and serverless-based deployment by monitoring loads and predicting the CPU and memory usage of these platforms. These storage-related existing efforts lack fine-grained serverless function provisioning and configuration towards flexible requirements of end-to-end performance and monetary cost. Our work, in contrast, does not rely on more expensive storage. Instead, we explore the optimization space merely using the naive and cheapest S3, through a dedicated function orchestration and provisioning.

TABLE 2
Comparison of *Astrea* With Existing Frameworks of Serverless Data Analytics and Machine Learning

Efforts	App. type	Inter. data stor.	Res. prov. strat.	Cost min.	Perf. opt.	Fine. mod.
PyWren [10]	MapReduce style	S3	heu. based	X	X	X
Flint [25]	Query style	SQS	heu. based	X	X	X
Locus [6]	MapReduce style	S3 & Redis	heu. based	X	X	✓
Lambada [31]	Query style	S3, DynamoDB, & SQS	heu. based	X	X	✓
Kappa [27]	MapReduce style	S3 & Redis	heu. based	X	X	X
Cirrus [21]	Machine Learning	Redis	heu. based	X	✓	✓
SIREN [34]	Machine Learning	S3	opt.-based	X	✓	✓
λDNN [35]	Machine Learning	VMs	opt.-based	✓	✓	✓
BATCH [36]	Machine Learning	VMs	opt.-based	✓	✓	✓
Gillis [12]	Machine Learning	REST API	opt.-based	✓	X	✓
AMPS-Inf [37]	Machine Learning	S3	opt.-based	✓	X	✓
Astrea	MapReduce style	S3	opt.-based	✓	✓	✓

The 2nd to the 7th columns respectively represent "Application type," "Intermediate data storage," "Resource provisioning strategy," "Cost minimization," "Performance optimization," and "Fine-grained modeling". For the 4th column items, "heu." is short for "heuristic" and "opt." stands for "optimization".

Machine Learning on Serverless Platforms. There are a number of efforts that study the employment of serverless platforms for machine learning workloads. Cirrus [21], SIREN [34] and λDNN [35] proposed strategies to provision and coordinate serverless functions for the iterative model training process of machine learning. The comparative study of distributed machine learning training over FaaS and IaaS systematically depicts the tradeoff in the design space [38]. For a particular Graph Neural Network (GNN) training, Dorylus [39] divides the training pipeline into a set of fine-grained tasks, of which the processing of tensors is deployed on lambdas while the graph structure related operations are executed on CPU servers. On the other hand, focusing on machine learning inference, BATCH [36] is prototyped on AWS Lambda for model serving, where requests are buffered to be later processed in a batch. BARISTA [40] is a scalable serving system for deep learning prediction services. Gillis [12] and AMPS-Inf [37] focus on automatic model partitioning and resource provisioning for large model inference in the serverless environment, with the awareness of both Service Level Objective (SLO) and cost.

In summary, Table 2 presents a high-level comparison of *Astrea* with the existing representative efforts, from the perspectives of the focused application type, intermediate data storage, resource provisioning strategy, cost and performance optimizations, and fine-grained modeling. Different from all the existing works, we propose a framework to automatically configure and orchestrate the MapReduce job in serverless environment towards flexibly specified objectives. Fine-grained modeling is employed in *Astrea* to formulate

Authorized licensed use limited to: East China Normal University. Downloaded on December 15,2022 at 03:36:29 UTC from IEEE Xplore. Restrictions apply.

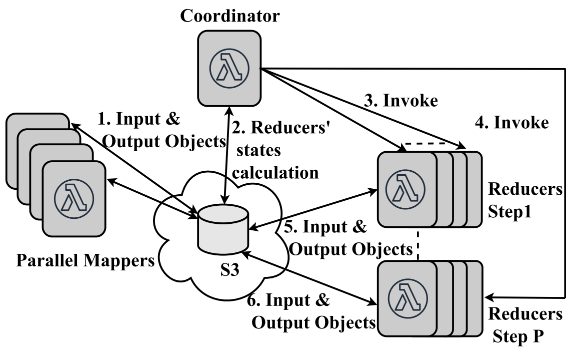

Coordinator
1. Input & Output Objects
2. Reducers' states calculation
3. Invoke
4. Invoke
Reducers Step1
5. Input & Output Objects
Parallel Mappers
S3
6. Input & Output Objects
Reducers Step P

Fig. 4. Workflow of a serverless MapReduce job in AWS Lambda. The job uses AWS S3 as the intermediate storage.

optimization problems towards cost minimization and performance optimization goals. Our work is for the provisioning of a single job from the user perspective given a serverless platform, orthogonal to the job schedulers, such as Skippy [41], that operate within the serverless platform from the provider perspective.

3 Modeling Serverless Data Analytics

In this section, we present our modeling of job completion time and monetary cost for serverless data analytics, with a MapReduce job implemented in Amazon Lambda as an example. More specifically, we consider a job processing N number of input objects with D size, which are stored in AWS S3. The job consists of three types of lambda functions, to map, coordinate[2] and reduce, respectively, as shown in Fig. 4. Our modeling is easily adapted to a general serverless data analytics setting, to be discussed later.

3.1 Performance of Completion Time

3.1.1 Lifetime of Mappers

As illustrated by Fig. 3 in Section 2, a number of identical mapper lambdas will be launched in parallel, each performing computation for $k_M \leq N$ objects. As there are N objects in total, the number of mapper lambdas can be represented as N/k_M, which has a maximum value of λ_M. For a mapper lambda, the lifetime is determined by both the S3 requests and the computation. Specifically, the time it takes to get and put objects in S3 depends on the data transfer time between lambda and S3, which is determined by the network bandwidth B and the sizes of objects to read and write. Intuitively, if we have a larger number of mapper lambdas, the data size in transmission of each mapper will be smaller. Given a total of j mappers, we use d_m^j and e_m^j to denote the input size and output size, respectively, for the mapper m, and the output size is proportional to the input size. Thus, the time associated with S3 requests of this mapper is represented as $(d_m^j + e_m^j)/B$.

The computation time of a mapper lambda relies on the computation workload and the processing power of the particular lambda. In AWS Lambda, we can allocate memory for a lambda from 128 MB to a maximum limit in 64 MB increments[1]. The CPU capacity of a lambda function is proportional to the allocated function memory [35]. We use binary variable $x_i, (i = 1, 2, \dots, L)$ to specify whether the ith type of memory is allocated (for mapper lambdas) out of the L categories. Intuitively, we have

$$x_i \in \{0, 1\}, \quad \forall i \in \{1, 2, \dots, L\}; \qquad \textstyle\sum_{i=1}^{L} x_i = 1, \tag{1}$$

which indicates that only one category of memory allocation can be assigned by nature. We further use n_j to specify whether we launch j lambdas as mappers. In a similar vein, we have

$$n_j \in \{0, 1\}, \quad \forall j \in \{1, 2, \dots, \lambda_M\}; \quad \textstyle\sum_{j=1}^{\lambda_M} n_j = 1. \tag{2}$$

The computation workload for mapper m given j mappers is determined by the input size d_m^j. Therefore, we can express its computation time as

$$c_m^{j,i} = d_m^j \textstyle\sum_{i=1}^{L} x_i u_i, \qquad \forall i, j. \tag{3}$$

where u_i is the processing time of unit-size object given ith resource allocation.

The completion time of the mapping phase, denoted as $T_1^{j,i}$, is determined by the slowest mapper, which can be represented as the maximum computation time among all the concurrent ones as follows:

$$\begin{aligned} t_m^{j,i} &= (d_m^j + e_m^j)/B + c_m^{j,i}, \\ T_1^{j,i} &= \textstyle\sum_{j=1}^{\lambda_M} n_j (\max_{m \in \{1,2,\dots,j\}} t_m^{j,i}), \qquad \forall i, j. \end{aligned} \tag{4}$$

According to Equation (4), the mapping phase completion time is dependent on the number of lambdas running in parallel (j) and the type of lambda in terms of the memory allocation (i).

3.1.2 Lifetime of Coordinator

After the mapping phase, a coordinator lambda will be launched to determine the number of reducing steps, denoted as P, and the number of reducers to be called in each step, denoted as g_p for each step $p, (p = 1, 2, \dots, P)$. In each step, the coordinator stores a reducer state object of size l in S3, which contains the count of reducers and the information about intermediate objects to be used by the reducers. Intuitively, the state object has the same size for all the steps.

Similar to the memory allocation for mapper lambdas among L types within the range of 128 MB to 3008MB[1], we use the binary variable $y_a, (a = 1, 2, \dots, L)$ to specify whether the ath memory allocation is chosen:

$$y_a \in \{0, 1\}, \quad \forall a \in \{1, 2, \dots, L\}; \quad \textstyle\sum_{a=1}^{L} y_a = 1. \tag{5}$$

For the coordinator lambda, the lifetime is determined by its computation time before the beginning of the reducer phase, the data transfer time before each reducer step to write state information, and the sum of lifetime of the first $P - 1$ reducer steps, as shown in the timeline in Fig. 3. The total data transfer time incurred by S3 put requests across P steps can be represented as $P * l/B$, given the network

2. An alternative to the coordinate lambda is to use AWS step functions [42], which allows the coordination of multiple services into serverless workflows. As step function involves state transaction cost, we choose to use a coordinate lambda which is more flexible and cost-efficient for *Astrea*.

Authorized licensed use limited to: East China Normal University. Downloaded on December 15,2022 at 03:36:29 UTC from IEEE Xplore. Restrictions apply.

TABLE 3
The Calculation Derived by the Coordinator for the Number of Objects Processed by Each Step, the Number of Reducers in Each Step, and the Sizes of the Input and Output Objects at Each Step

Step	No. of Objects	No. of Reducers	Input Get (MB)	Output Put (MB)
1	j	$g_1 = j/k_R$	$q_0 = S$	q_1
2	g_1	$g_2 = g_1/k_R$	q_1	q_2
....				
P	g_{P-1}	g_P	$q_{(P-1)}$	q_P

bandwidth B. The computation time of the coordinator will be determined by the computation power and workload, which are impacted by the lambda memory type and the total number of objects as input for the reducing phase. As the lifetime of the P-1 reducers will be included in the reducing phase in the next subsection, we denote $T_2^{g,a}$ as the lifetime of the coordinator phase which excludes the overlapping time with reducers

$$t_2^{g,a} = c_2^{g,a} + P_{j,g} * l/B,\ T_2^{g,a} = \sum_{a=1}^{L} y_a t_2^{g,a},\ \forall a, j, g. \tag{6}$$

3.1.3 *Lifetime of Reducers*

The reducing phase will be executed in P steps and each lambda will handle the same amount of objects. The calculations of each step are shown in Table 3, including the total number of objects to be handled, the total number of reducer lambdas to be launched, the total size of input objects to retrieve, and the total size of output objects to store.

In the first step of the reducing phase, a total number of j objects, resulting from j mappers, need to be read as input. Given the number of objects, denoted as $k_R \leq j$, to be handled by each reducer, we need to launch $g_1 = j/k_R$ lambdas in this step, which read the total amount of data ($q_0 = S$) generated from the mapping phase and write q_1 amount of data for further processing of the next step. In a general step p, the number of reducers is denoted as g_p. The total number of objects is equal to the total number of reducers from the previous step, g_{p-1}, and the size of the total input objects (Get) is equal to the size of the total output objects (Put) from the previous step, q_{p-1}. We represent the total number of reducers as g, which can be derived as $\sum_{i=1}^{P} g_p$.

Given the same set of memory allocations in L types, we use binary variable $z_s, (s = 1, 2, \ldots, L)$ to specify whether the sth memory allocation is selected for reducer lambdas or not, which naturally has the following constraints:

$$z_s \in \{0, 1\},\ \ \forall s \in \{1, 2, \ldots, L\};\ \ \ \sum_{s=1}^{L} z_s = 1. \tag{7}$$

Similarly, we use w_g to specify whether or not we launch g lambdas as reducers.

$$w_g \in \{0, 1\},\ \ \forall g \in \{1, 2, \ldots, \lambda_M\};\ \ \ \sum_{g=1}^{\lambda_M} w_g = 1. \tag{8}$$

The lifetime of the reducing phase depends on the total data transfer time and the lambda computation time. Given a total of g reducers in P steps, let Q_P^g denote the total input object size, which can be expressed as $\sum_{i=0}^{P-1} q_p$, according to Table 3. Similarly, the total output object size $\sum_{i=1}^{P} q_p$ is denoted by R_P^g. The total computation time in the reducing phase can be represented as

$$o_P^{g,s} = Q_P^g \sum_{s=1}^{L} z_s u_s,\ \ \forall s \in \{1, 2, \ldots, L\},$$

where u_s is the processing time of unit-size object given sth resource allocation. The data transfer time of the reducing phase, denoted as $d_3^{g,s}$, can be expressed as

$$d_3^{j,g} = (Q_P^g + R_P^g)/B,$$

which depends on the total size of input data, output data, and the bandwidth B. Finally, we obtain the lifetime of the reducing phase, $T_P^{g,s}$, as the summation of the total computation time and data transfer time

$$T_P^{g,s} = \sum_{g=1}^{\lambda_M} w_g (d_3^{j,g} + o_P^{g,s}),\ \ \ \ \forall s, g. \tag{9}$$

3.2 Monetary Cost

The monetary cost for the serverless analytics job is incurred by the Get and Put requests from S3, the storage of the input and intermediate objects, the invocation of lambdas and their execution times.

3.2.1 *Requests Cost of Lambdas*

Given j mappers in the mapping phase, the cost for S3 requests, U_1^j, is determined by k_M Get requests and one Put request from each mapper. The cost for coordinator, denoted as $U_2^{j,g}$, is incurred by writing reducer state object in S3, determined by the total number of the reducer steps. $U_P^{j,g}$ denotes the requests cost for the reducing phase, incurred by each reducer getting k_R number of objects and putting one object in S3. With the standard pricing [43] of \$0.005 per 1,000 Put requests (F) and \$0.004 per 10,000 Get requests (G) in S3, we have the following expressions of S3 requests cost for each phase:

$$\begin{aligned} U_1^j &= j(k_M * G + 1 * F),\quad U_2^{j,g} = Pj, g * P,\\ U_P^{j,g} &= g(k_R * G + 1 * P). \end{aligned} \tag{10}$$

3.2.2 *Storage Cost of Objects*

Apart from the cost incurred by Put and Get requests, the storage of objects in S3 depends on the size of data and the duration for storage. In our considered serverless MapReduce job, the input objects will be stored in S3 until the completion of the job. In addition to the storage cost for input objects, the coordinator and reducers will generate storage cost for intermediate objects. We denote the storage costs for the three phases as $V_1^{j,i}$, $V_2^{g,a}$ and $V_P^{g,s}$, respectively, given j mappers, g reducers and the lambda resource types (i for mappers, a for the coordinator and s for reducers). The size of objects handled by the coordinator is denoted as S, and the unit price for storage (per unit size and unit time) is represented as H. Thus, we have

$$\begin{aligned} V_1^{j,i} &= DT_1^{j,i}H,\quad V_2^{g,a} = T_2^{g,a}(D + S + Q_P^g)H,\\ V_P^{g,s} &= T_P^{g,s}(D + S + R_P^g)H. \end{aligned} \tag{11}$$

Authorized licensed use limited to: East China Normal University. Downloaded on December 15,2022 at 03:36:29 UTC from IEEE Xplore. Restrictions apply.

3.2.3 Runtime Cost of Lambdas

The runtime cost of lambdas for the MapReduce job consists of the invocation cost and the computation cost. The invoking price for a lambda is \$0.20 per 1 million requests [24] and represented as E. Let us denote the invoking costs for the three phases as I_1^j, $I_2^{j,g}$ and $I_3^{j,g}$ respectively, each of which depends on the number of lambdas of the particular phase, expressed as

$$I_1^j = j * E, \quad I_2^{j,g} = 1 * E, \quad I_3^{j,g} = g * E. \tag{12}$$

The computation cost of each lambda is determined by the price of the allocated memory and the duration it runs. We use v_i, v_a and v_s to represent the price of the particular type of lambda. Intuitively, the runtime costs of lambdas for the three phases, denoted as $W_1^{j,i}$, $W_2^{g,a}$ and $W_P^{g,s}$, respectively, can be expressed as follows:

$$W_1^{j,i} = \textstyle\sum_{i\in\mathcal{L}} v_i x_i T_1^{j,i} + I_1^j \tag{13}$$

$$W_2^{g,a} = \textstyle\sum_{a\in\mathcal{L}} v_a y_a (T_2^{g,a} + T_{P-1}^{g,s}) + I_2^{j,g} \tag{14}$$

$$W_P^{g,s} = \textstyle\sum_{s\in\mathcal{L}} v_s z_s T_P^{g,s} + I_3^{j,g}. \tag{15}$$

With the comprehensive modeling for completion time and monetary cost of a serverless analytics job, we are now ready to formulate optimization problems according to particular objectives in the next section.

4 Optimizations for Performance Enhancement and Cost Reduction

In this section, we formulate optimization problems according to user requirements, and design solutions based on graph theory to navigate the cost-performance tradeoff.

4.1 Performance Optimization Given a Budget

Constrained with a particular budget, we aim to optimize application performance, which means minimizing the job completion time, formulated as follows:

$$\min_{\boldsymbol{x,y,z,n,w}} \quad \boldsymbol{f} = T_1^{j,i} + T_2^{g,a} + T_P^{g,s} \tag{16}$$

$$\text{s.t.} \quad Eq.\ (1), (2), (5), (7), (8) \tag{17}$$

$$D + S + Q_P^g \le O, \quad j \le R \tag{18}$$

$$U_1^j + U_2^{j,g} + U_P^{j,g} + V_1^{j,i} + V_2^{g,a} + V_P^{g,s} + W_1^{j,i} + W_2^{g,a} + W_P^{g,s} \le J. \tag{19}$$

In this optimization problem, the objective of job completion time is the summation of the mapping phase duration $T_1^{j,i}$, the total coordinator time between reducing steps, $T_2^{g,a}$, and the total lifetime of reducing steps, $T_P^{g,s}$, of which the expressions have been derived in the previous section. Constraint (17) regulates the nature of the binary variables. Constraint (18) indicates the limits in AWS Lambda for maximum storage size (O, which is currently 5 TB) and for the maximum number of requested lambdas (R). The budget limit (J), represented by Constraint (19), can be flexibly specified by the user.

This problem has binary variables and is intuitively NP-hard [44]. To solve this problem, we propose a strategy

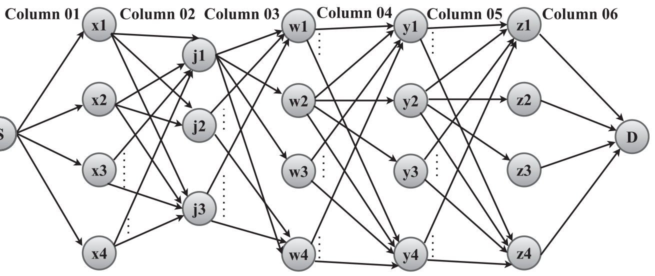


Fig. 5. DAG for performance optimization (Eq. (16)) or cost minimization (Eq. (20)), depending on the associated edge weights from Column 01 to Column 06.

based on graph theory [45], an effective tool in resource allocation and scheduling. Our problem formulation naturally maps to the shortest path problem. We construct a directed acyclic graph (DAG) as shown in Fig. 5. The vertices of the graph represent the resource allocation for lambda functions along the workflow, and the edge weights are the resulted completion times for particular phases.

With this graph, a flow starting from the source node $\overline{S}$ will go through five nodes to reach the destination $\overline{D}$. Each of the five nodes along the flow path represents an allocation of a particular resource. In particular, the five columns of vertices along the DAG in order represent the memory allocation for mapper lambdas, the number of mappers, the number of objects per reducers, the memory allocation for coordinator lambda and the memory allocation for reducer lambdas.

The edge weights are set as the completion times that are associated with the resource allocations specified by the connected vertices. For the first set of edges between the first two columns of vertices, the edge weight represents the resulted mapper completion time (Eq. (4)). For example, the weight of the edge between vertices x_1 and j_3 means the completion time of each mapper, if there are j_3 lambdas allocated as mappers, each with x_1 type of memory allocation. Similarly, weights of the second set of edges are specified as the aggregation of the data transfer time of the coordinator and the reducing phases ($d_2^{j,g} + d_3^{j,g}$). For the third set of edges, weights are assigned as the coordinator phase computation time ($c_2^{g,a}$). Finally, weights of edges between the fourth and fifth column of vertices represent the computation time of the reducing phase (Eq. (9)).

With such an edge weight assignment, optimizing job completion time is equivalent to finding the shortest path. We develop Algorithm 1 to find the optimal resource allocation towards minimized job completion time, based on the shortest path algorithm [46].

4.2 Cost Minimization With Quality-of-Service

We next consider the following cost minimization problem, given a threshold for the purpose of meeting Quality-of-Service (QoS) requirement.

$$\min_{\boldsymbol{x,y,z,n,w}} \quad \boldsymbol{h} = U_1^j + U_2^{j,g} + U_P^{j,g} + V_1^{j,i} + V_2^{g,a} + V_P^{g,s} + W_1^{j,i} + W_2^{g,a} + W_P^{g,s} \tag{20}$$

$$\text{s.t.} \quad Eq.\ (1), (2), (5), (7), (8),\ \text{and}\ (18) \tag{21}$$

$$T_1^{j,i} + T_2^{g,a} + T_P^{g,s} \le E. \tag{22}$$

Authorized licensed use limited to: East China Normal University. Downloaded on December 15,2022 at 03:36:29 UTC from IEEE Xplore. Restrictions apply.

The objective is the total monetary cost that needs to be paid for the running of the serverless job, including the requests costs, storage costs and runtime costs incurred by mappers, coordinator and reducers. Similar to the previous optimization problem, we have the constraints for binary variables and for resource upper limits. Constraint (22) regulates that the job performance should satisfy the QoS objective, which means that the job completion time does not exceed a user-specified threshold (E).

To construct the DAG for cost minimization, the vertices are the same with the completion time optimization, as we have the same set of resource allocation variables. The edges of the DAG are associated with the weights that denote the monetary costs of the three phases resulting from the corresponding resource allocation. Specifically, the weights of the first set of edges in Fig. 5 represent the costs of the mapper phase ($U_1^j + V_1^{j,i} + W_1^{j,i}$). The second set of edges gives the aggregation of requests costs and invoking costs during the coordinator and reducer phase ($U_2^{j,g} + U_P^{j,g} + I_2^{j,g} + I_3^{j,g}$). The next set of edges is associated with coordinator storage cost ($V_2^{g,a}$) and lambda cost ($C_2^{g,a}$). The last set of edges represents the aggregation of the storage and lambda costs ($V_P^{g,s} + W_P^{g,s}$) of the reducing phase. Similarly, Algorithm 1 can be used to identify the shortest path as the optimal solution, with minimal changes.

4.3 Multi-Round Extension

When we extend our consideration to a large-scale data analytics job with heavy workloads, Algorithm 1 may no longer be able to generate a feasible deployment plan. The reason is that a larger job requires more concurrent lambda functions to meet the performance requirement, while serverless platforms have resource limitations [13] on the maximum concurrency (i.e., the maximum number of lambdas requested by a job) which will be violated. For example, consider a MapReduce job with 2,500 input objects. Each of the objects is of large size, so that 2,500 mapper lambdas are required to handle each object in parallel, exceeding the maximum concurrency of 1,000 allowed by the AWS Lambda platform. In this case, it is impossible for Algorithm 1 to generate a feasible serverless deployment solution. To accommodate a large job, an intuitive way is to deploy it across multiple rounds, each with a smaller number of concurrent functions within the concurrency limit. In addition, another type of circumstance is that the size of temporary storage for the objects handled by a lambda goes beyond the platform limit (512 MB on AWS Lambda), which needs to be identified and resolved carefully. Therefore, motivated by the aforementioned challenges of directly applying *Astrea* for a large-scale job, we next present our extended design of *Astrea* for multi-round deployment of large jobs to satisfy the maximum concurrency limit and maximum temporal storage limit of serverless platforms.

The essential idea of our extended design is to carefully apply Algorithm 1 with relaxed constraints across rounds, when the maximal required concurrency is greater than the platform limit. As presented in Algorithm 2, when no feasible configuration exists, the constraint in Algorithm 1 is relaxed by adding the value of maximum concurrency limit R_0 (`line 4`), until a feasible solution is identified. From the feasible solution P, we can obtain the set of mappers $\mathcal{M}$ (`line 6`), where each element contains the information of the input objects assigned to the particular mapper. If the total number of mappers, $|\mathcal{M}|$, violates the platform limit for maximum concurrency R_0, we split the workload across multiple rounds so that the lambda functions required in each round do not overwhelm the serverless platform and are judiciously configured (`lines 7-20`).

In particular, the number of rounds is calculated by dividing the total number of mappers with R_0 (`line 8`). For each round, the set of input objects, $\mathcal{D}_r$, to be processed is obtained from $\mathcal{M}_r$, the set of mappers assigned in this round (`lines 10-15`). We can consider each round as running a subjob which processes a partition of the input dataset (i.e., $\mathcal{D}_r$). For the subjob, we can again use Algorithm 1 to obtain the optimal configuration based on an updated problem formulation (`lines 16-17`). Given the solution P_r, we can obtain the set of mappers, $\mathcal{M}_r^*$, which includes the input object information per mapper (`line 18`). Accordingly, we can then check the temporary storage size limit for each mapper and switch object assignments between mappers if necessary to avoid exceeding the limit (`line 19`). Such configurations will be added to $Paths$ round by round, which will be eventually returned for *Astrea* to enforce the execution accordingly.

It is worth noting that Algorithm 1 is a special case of Algorithm 2 without temporary storage size checking. In other words, our extended design of *Astrea* enables fault-tolerance for temporary storage violation, and allows multi-round deployment to accommodate large-scale jobs. Intuitively, executing a job in multiple sequential rounds would prolong the job completion. If a job is more stringent on completion time, complementary directions of leveraging faster storage service or designing better intermediate data management strategy to further improve job performance (at the expense of extra billing cost) could be explored, to be further elaborated in the following sections.

Algorithm 1. *Astrea*: Finding the Optimal Resource Allocation (by minimizing the completion time)

```
Input: $W(u,v)$: time, $C(u,v)$: costs (edge constraint from
        Equation (19)), $F(u,v)$: storage, Source $\overline{S}$, Destination $\overline{D}$
Output: Best performance path with acceptable cost.
 1: Initialize: DAG $G$ with edges $E$ and vertices $(u,v)$
 2: $P \leftarrow$ Apply Dijkstra algorithm for the Graph G
 3: $u \leftarrow \overline{S}$
 4: while $P$ is a path and $u \neq \overline{D}$ do
 5:   $cost \leftarrow 0, stor \leftarrow 0$
 6:   while $u \neq \overline{D}$ do
 7:     $cost \leftarrow cost + C(u,v), stor \leftarrow stor + F(u,v)$
 8:     if $cost \geq budget$ then
 9:       Remove that edge from the edges $E$
10:       $P \leftarrow$ Apply Dijkstra algorithm for the Graph G
11:       $u \leftarrow \overline{S}$
12:       break
13:     else
14:       $u \leftarrow v$
15: return $P$
```

5 Performance Evaluation

In this section, we present the design and implementation of *Astrea*, and evaluate its performance with extensive real-

Authorized licensed use limited to: East China Normal University. Downloaded on December 15,2022 at 03:36:29 UTC from IEEE Xplore. Restrictions apply.

world experiments over a wide array of Big Data analytics benchmarks and machine learning workloads. Our purpose is to show the effectiveness of *Astrea* and its multi-round extension when increasing the problem scale beyond the platform limit for one-round deployment.

Algorithm 2. *Astrea-extended*: Finding the Optimal Resource Allocation for a Very Large Job through Multiple Rounds

```
Input: $R_0$: Maximum allowed current lambdas, $\mathcal{D}$: The set of
       input objects, Algorithm 1
Output: $Paths$
 1: Initialize: $round \leftarrow 1$, $R \leftarrow R_0$
 2: $P \leftarrow$ execute Algorithm 1 with limit $R$
 3: while $P$ is not a feasible path do
 4:     $R \leftarrow R + R_0$
 5:     $P \leftarrow$ execute Algorithm 1 with limit $R$
 6: Obtain the set of mappers, $\mathcal{M}$, from $P$
 7: if $|\mathcal{M}| > R_0$ then
 8:     $round \leftarrow \lceil \frac{|\mathcal{M}|}{R_0} \rceil$
 9:     ForEach $r \in [1, round]$ do
10:         if $r < round$ then
11:             $\mathcal{M}_r \leftarrow$ Get $R_0$ mappers from $\mathcal{M}$
12:             $\mathcal{M} \leftarrow \mathcal{M} - \mathcal{M}_r$
13:         else
14:             $\mathcal{M}_r \leftarrow \mathcal{M}$
15:         Obtain input object set $\mathcal{D}_r$ from $\mathcal{M}_r$
16:         Update problem formulation for Algorithm 1
17:         $P_r \leftarrow$ execute Algorithm 1 with limit $R_0$
18:         Obtain the set of mappers, $\mathcal{M}_r^*$, from $P_r$
19:         Check temporary storage size (and update $\mathcal{M}_r^*$)
20:         Append $[P_r, \mathcal{M}_r^*]$ to $Paths$
21: else
22:     Check temporary storage size (and update $\mathcal{M}$)
23:     Append $[P, \mathcal{M}]$ to $Paths$
24: return $Paths$
```

5.1 Prototype Implementation and Experimental Setup

Astrea is designed and implemented in AWS Lambda. When a user submits a data analytics job, *Astrea* will model the performance and cost for the job using `Performance Predictor` and `Cost Predictor` modules. With the modeling and the user-specified requirement, *Astrea* provisions resources based on the algorithm described in Section 4, to navigate the cost-performance tradeoff. Finally, *Astrea* deploys the user code according to the best orchestration and configuration plan, and the job will be executed accordingly in the serverless environment.

In the two modules of performance modeling, we use the following settings. The reducer state object written by the coordinator to S3 before each reducer step normally has one line to specify the number of reducers and the number of objects. Thus, it is assumed as 1 MB in size. The computation time of each lambda is proportional to its memory size, which ranges from 128 MB to 3008 MB, with 64 MB increments [13] in AWS Lambda. Each lambda has a limit of 900 seconds for execution.

Provisioning Baselines. There are three baselines for serverless provisioning implemented to compare with *Astrea*,

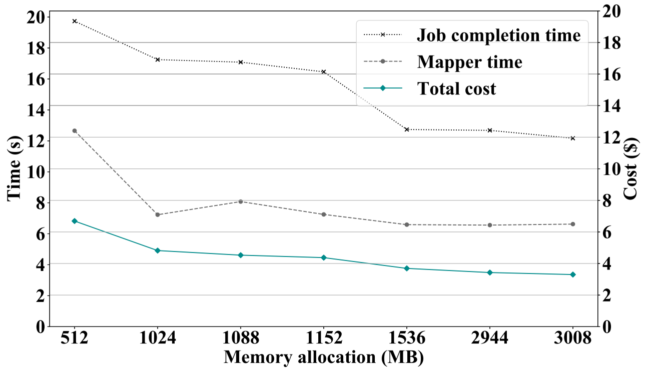


Fig. 6. The change of completion time, mapper phase time and monetary cost with memory allocation.

based on the experimental observation of serverless Wordcount as shown in Fig. 6. Without modeling the intricate inter-dependencies among various factors, this observation simply provides some vague sense about the general behavior of performance and cost with respect to memory allocation, serving as a rough guideline for the baselines. Since the memory blocks greater than 1536 MB are not showing much improvement in completion time, as observed in Fig. 6, 1536 MB is allocated for all lambdas in Baseline 1 (called *max_performance*), and the number of objects per mapper is set as 1 to realize the maximum degree of parallelism for mappers. We allocate the number of objects per reducer as 2 which is empirically good. Baseline 2 (called *min_cost*) is implemented with a setting from the cost point of view. With a preference for cost saving, the lambdas are naively allocated with the smallest memory block 128 MB, and the objects allocations are maintained the same as baseline *max_performance*. The third baseline has a hybrid consideration of both performance and cost. It follows the same setting as *min_cost* for parallel mappers, each with 128 MB to process one object. For the reducing phase, Baseline 3 (called *hybrid*) allocates 1536 MB to three reducer lambdas in two steps, and the two reducers in the first step each process half of the total objects.

Workloads. We have conducted our experiments under five different workloads listed as follows:

- Big Data Benchmark [14]:
  - Query (selection) over Rankings dataset, with the size of 6.38 GB, which lists websites and their page ranks. The dataset has 90 Million rows, each including pageURL, pageRank, and avgDuration.
  - Query (aggregation) over two Uservisits datasets, with the size of 25.4 GB and 126.8 GB, respectively, stored in S3 as 202 objects. The datasets have 155 million and 775 million individual rows, respectively, each consisting of sourceIP, visitDate, adRevenue, userAgent, countryCode, languageCode, searchWord, and duration.
- Wordcount, with the input size of 1 GB, 10 GB, and 20 GB, respectively.
- Sort, with the input size of 100 GB.
- Laptop pricing dataset, with the size of 10 GB, used for K-Nearest Neighbours algorithm. Each row consists of Manufacturer, IntelCore, IntelCoreGen processing

Authorized licensed use limited to: East China Normal University. Downloaded on December 15,2022 at 03:36:29 UTC from IEEE Xplore. Restrictions apply.

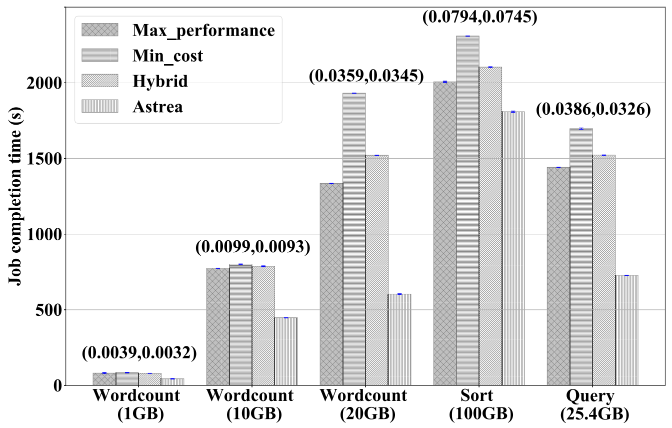


Fig. 7. Job completion time achieved by *Astrea* and three baselines, for Wordcount (3 scales), Sort and Query workloads, given a budget constraint.

speed, Ram, HDD, SSD, Graphics, ScreenSize, and Price.

- Seeds dataset, with the size of 10 GB, used for K-Means Clustering. Each row consists of area, perimeter, compactness, length of kernel, width of kernel, and asymmetry coefficient length of kernel groove.

In our evaluations, we illustrate the performance and monetary cost with error bars of standard deviation by repeating the experiments for three times. In the following subsections, we present the results and analysis of four groups of experiments, briefly described as follows:

- We compare *Astrea* with the three provisioning baselines aforementioned over various workloads with increasing scales.
- We compare *Astrea* with Apache Spark [15] deployed on industrial platforms of EC2 [16] and SageMaker [17], over various Big Data analytics and machine learning workloads.
- We compare *Astrea* with Amazon EMR [18], the built-in industrial VM-based MapReduce platform, for two different workloads.
- We evaluate our extended multi-round *Astrea*, in comparison with Apache Spark over various workloads of which the scales increase beyond platform limits for one-round deployment.

5.2 *Astrea versus* Provisioning Baselines

To begin with, we evaluate the behavior of *Astrea* in identifying the optimal resource configuration and orchestration for performance optimization, given a cost budget.

5.2.1 Wordcount, Sort and Query Workloads

Fig. 7 presents the completion time achieved by *Astrea*, in comparison with the three baselines, for different workloads. The budget constraints and the resulted costs (by *Astrea*) are shown with 2-tuples above the bar groups for each benchmark. As clearly shown in Fig. 7, *Astrea* outperforms all the three baselines in terms of reducing the completion time for all the workloads, without exceeding budgets. Baseline *max_performance*, with the highest memory allocation for lambdas, outperforms the other two baselines with shorter completion times for all the workloads, but is still far from competitive with *Astrea*. More specifically, for the Wordcount benchmark with increasing scales of 1 GB, 10 GB, and 20 GB, *Astrea* achieves performance improvement over *max_performance* of 46.20%, 42.51%, and 54.78%, respectively. Similarly, for Query and Sort, *Astrea* outperforms *max_performance* by at least 49.31% and 10.08%, respectively. Considering all the three baselines, *Astrea* achieves $42 - 69\%$ improvement for Wordcount in all scales, up to 21% for Sort, and 57% for Query benchmark, respectively.

TABLE 4
The Resource Allocations Achieved by *Astrea*, for the Three Benchmarks When Optimizing Job Performance

	Wordcount (1 GB)	Wordcount (10 GB)	Wordcount (20 GB)	Sort (100 GB)	Query (25.4 GB)
Map., Co., Red. memory	256, 256, 1024	128, 1024, 1024	256, 1024, 1024	256, 256, 1024	128, 256, 1024
Obj. (map.)	2	8	4	4	1
Obj. (red.)	2	11	2	8	11
Mappers	10	3	10	50	202
Reducers	11	1	11	7	22
Red. steps	4	1	4	1	4

To further illustrate how *Astrea* works and analyze its advantages, Table 4 presents the budget-constrained performance-optimal resource provisioning in *Astrea*, for the three workloads with different scales. Specifically, the resource allocation includes specifying the memory type for mapper and reducer lambdas, the number of objects processed per mapper and the number of objects processed per reducer, which can further determine the number of (mapper or reducer) lambdas and the number of reducer steps.

For the Query benchmark, the number of objects per mapper is allocated as 1 by *Astrea*, resulting in 202 mappers with a maximum degree of parallelism. If the number of objects per mapper is more than one, there will be fewer mappers, each processing more input data, and the data transfer time will be longer, impacting the job completion time. 128 MB memory is allocated for each mapper, which is sufficient to process one object and cost-effective. For the reducing phase, if the number of objects per reducer is too small, then a large number of reducers will be required in the first step, followed by a relatively large number of subsequent steps, which prolongs the job completion. On the other hand, if the number of objects per reducers is more than 15, there will be one reducer in the second step that needs to handle all the objects from the first step, which incurs large data transfer time and increases the completion time. *Astrea* judiciously sets the number of 11, resulting in 22 reducers within 4 steps, and allocates 1024 MB memory, to speed up the job.

For the Sort benchmark, the total size of the input is 100 GB, and each of the 200 objects is as large as 500 MB. If the number of objects increases to 5 or more, then the size of the objects processed by each lambda will be larger, which thus increase the data transfer time. *Astrea* sets 4 objects per mapper, each with 256 MB memory, to achieve a good balance between computation time decrease (per mapper) and transfer time increase. Similarly, for the reducer phase, *Astrea* sets 8 objects per reducer, each with 1024 MB, to finish within 1 step, as the outcome from optimizing completion time given the budget.

Authorized licensed use limited to: East China Normal University. Downloaded on December 15,2022 at 03:36:29 UTC from IEEE Xplore. Restrictions apply.

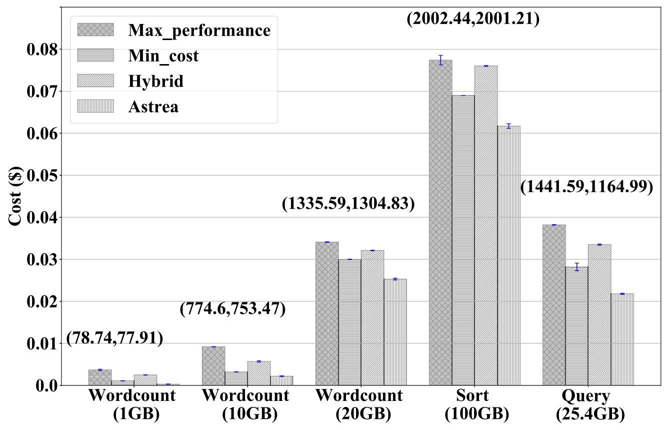


Fig. 8. Monetary cost achieved by *Astrea* and three baselines, for Wordcount (3 scales), Sort and Query workloads, given a completion time threshold.

Similarly, when aiming at minimizing the monetary cost incurred by the job given a particular threshold of job completion time, *Astrea* manages to find the optimal resource configurations to orchestrate lambda functions, as demonstrated in Fig. 8. For each workload, the job completion time threshold (for QoS purpose) is indicated by the first element in the 2-tuple above each workload bar group, where the second element represents the actual job completion time with *Astrea*. It is easily verified that without exceeding the threshold, *Astrea* results in the smallest cost for each benchmark. *min_cost* is intuitively designed for cost saving, and thus results in a smaller cost than the other two baselines for all the workloads. Still, *Astrea* achieves nearly 71%, 33%, and 17% cost reduction over *min_cost*, for the three Wordcount benchmarks. For Sort and Query benchmarks, about 11% and 23% cost savings are exhibited, compared to *min_-cost*. In summary, compared with the three baselines, *Astrea* achieves the cost reduction of at least 20%, up to 87% for those three different benchmarks.

5.2.2 Selection and Aggregation Queries

We continue to perform evaluations with selection and aggregation queries over datasets at different scales in the Big Data Benchmark. Selection queries in the PageRank algorithm are over the Rankings dataset, with 90 million rows and the size of 6.38 GB. More specifically, Scan1a and Scan2a are selections queries that correspond to two different conditions of pageRank>1000 and pageRank>100, respectively. Fig. 9 presents a comprehensive performance comparison of *Astrea* with the three provisioning baselines when aiming at minimizing job completion time of four query jobs under their respective budget limits. Similar to Fig. 7, for each of the four query workloads, the 2-tuple above its bar group in Fig. 9 represents its budget limit and the resulted monetary cost with *Astrea*. As illustrated in the figure, for Scan1a and Scan2a queries, *Astrea* gained 29% and 31% performance improvement compared to *max_performance* which, with the maximum parallel lambdas, yields better performance than the other two baselines.

Aggregation queries, referred to as Aggregation2a and Aggregation2b in Fig. 9, are evaluated using two sets of input objects with 25.4 GB and 126.8 GB, respectively. These input

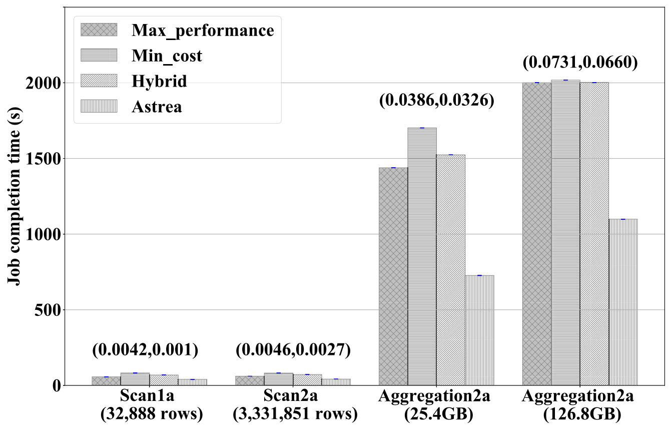


Fig. 9. Job completion time achieved by *Astrea* and three baselines, for BigData Benchmark queries, given a budget.

data are both from the Uservisits dataset with 155 million rows and 775 million rows, respectively. Similarly, *max_performance* resulted in the best completion time performance among the three baselines, while *Astrea* achieved 57% and 45% performance improvement over it for Aggregation2a and Aggregation2b, respectively, as shown in Fig. 9.

With respect to cost minimization without violating QoS requirement, Fig. 10 compares the monetary cost incurred by *Astrea* and the three provisioning baselines given a performance threshold, for each of the same four query jobs. Similar to Fig. 8, the QoS threshold with respect to job completion time is indicated by the first element in the 2-tuple above the bar group, while the job completion time achieved by *Astrea* is represented by the second element. As observed in Fig. 10, *Astrea* led to 82% and 84% cost reductions for Scan1a and Scan2a queries, respectively, when compared to *min_cost* which incurred the minimum cost among the three baselines. For the two aggregation queries, *Astrea* also outperformed *min_cost*, reducing the cost by 23% and 55%, respectively. To summarize, *Astrea* has been demonstrated to consistently outperform the provisioning baselines under various workloads at different scales.

5.3 *Astrea versus* Apache Spark

In this group of experiments, we focus on demonstrating the capabilities of *Astrea* to accommodate the Spark workloads. We evaluate how *Astrea* performs over Big Data analytics

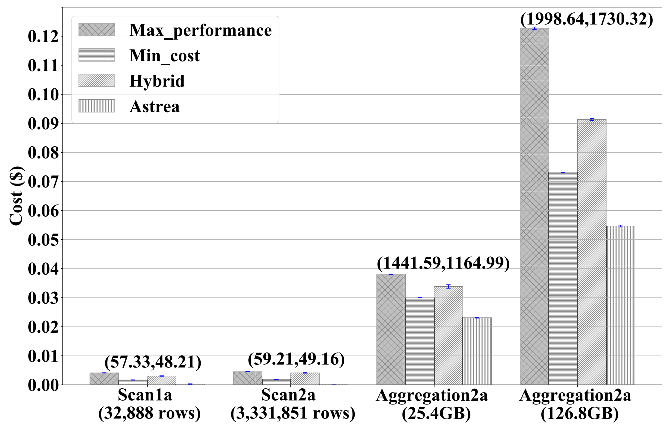


Fig. 10. Monetary cost achieved by *Astrea* and three baselines, for BigData Benchmark queries, given a completion time threshold.

Authorized licensed use limited to: East China Normal University. Downloaded on December 15,2022 at 03:36:29 UTC from IEEE Xplore. Restrictions apply.

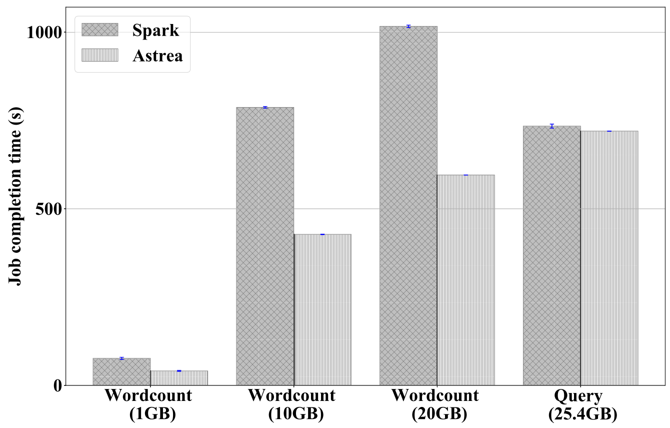


Fig. 11. Job completion time achieved by *Astrea* and Apache Spark, for Wordcount workloads with three different scales and Query benchmark.

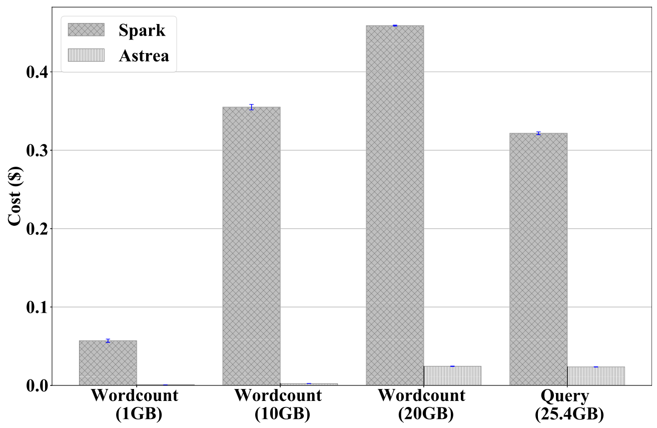


Fig. 12. Total cost achieved by *Astrea* and Apache Spark, for Wordcount workloads with three different scales and Query benchmark.

and machine learning workloads, in comparison with Apache Spark deployed on commercial platforms, i.e., EC2 [16] and SageMaker [17]. In this set of evaluation, without loss of generality, *Astrea* is configured in the cost-minimization setting, with the QoS threshold reasonably specified as the job completion time resulting from a baseline provisioning.

5.3.1 Wordcount and SQL Workloads

We first compare *Astrea* with Spark over traditional Big Data analytics workloads: Wordcount and SQL queries. Since a lambda function has limit on the temporary storage size and direct bulk transfer across functions is not feasible, *Astrea* deploys the workloads in the MapReduce form, without benefiting from in-memory computation as Spark. Despite such a disadvantage, we will show how the delicate resource provisioning enables *Astrea* to achieve comparable or even superior performance to Spark. To run the Spark Wordcount and Spark SQL, we provision a cluster of two `c4.4xlarge` (32 GB memory) VM instances on EC2 to deploy Apache Spark (version 3.0.1). We evaluate *Astrea* against the Spark baseline on three Wordcount jobs of different sizes and the query job over the 25.4 GB Uservisits dataset. The performance (job completion time) and the cost achieved by *Astrea* and Spark over the four jobs are presented in Figs. 11 and 12.

As observed in Fig. 11, for Wordcount jobs of three different scales and SQL query job, *Astrea* consistently outperformed Spark with respect to job completion time. Note that the metric of job completion time in the Spark cluster does not account for the cluster setup time (including loading application libraries) which is about 43 seconds. Considering the Wordcount jobs, *Astrea* achieved 41-50% performance improvement over Spark for the three scales, while the Query job in *Astrea* exhibited a slight improvement.

With respect to the monetary cost, in the Spark setting, it depends on the number of on-demand EC2 instances, the unit instance price [16] ($0.796 per hour in our setup), the storage price (negligible compared to the computation price), and the completion time of the Spark job. As illustrated in Fig. 12, *Astrea*, configured to minimize cost as aforementioned, resulted in at least 93% cost reduction compared to Spark. With increasing input scale of Wordcount, the cost saving of *Astrea* over Spark slightly decreases from 99% to 95%.

From Figs. 11 and 12, we can easily observe that *Astrea* manages to finish Big Data analytics jobs at a faster speed, and also with a much lower cost. Compared to the Spark setting with very limited concurrency and high instance cost, *Astrea* leverages light-weighted lambda functions and judiciously configures and coordinates them, so that a higher degree of concurrency can be achieved for job execution with the maximal cost-efficiency. For example, for the Query workload, about 200 mapper functions (each with 2 vCPU) concurrently work on the large input dataset in *Astrea*, to achieve comparable performance at a lower cost.

It is worth noting that our purpose in this set of experiments is to demonstrate the promise that the same analytics workload deployed on Lambda with *Astrea* can achieve comparable or superior performance to a reasonably configured Spark EC2 deployment. Although fine-grained Spark configuration is not our focus, we explore a few more settings for the 20 GB Spark Wordcount as follows, in addition to our previous two-`c4.4xlarge` configuration called *SparkVM_1*. We first set up a cluster of five `c4.large` instances (called *SparkVMcon_1*), each with 2vCPUs and less memory than *SparkVM_1*. This cluster, with a larger number of smaller instances, showed $\sim$26% performance improvement and reduced the cost by $\sim$48% over *SparkVM_1*. *Astrea* showed nearly 21% performance and 90% cost reduction over *SparkVMcon_1*. We then investigate different finer-grained configurations in the original two-`c4.4xlarge` cluster. In the first setting (called *Sparkcon_1*), each node has two executors with 4 cores (per executor), resulting in $\sim$15% performance improvement and $\sim$18% cost saving over *SparkVM_1*, which uses one executor by default. However, *Sparkcon_1* cannot beat *Astrea* in that the latter showed nearly 31% performance and 93% cost reduction. Another setting called *Sparkcon_2* configures each node to hold 8 executors with single core. Still, *Astrea* exhibited nearly 34% performance gain and 95% cost reduction over *Sparkcon_2*.

5.3.2 Machine Learning Workloads

We further evaluate the performance and cost of *Astrea* in comparison with Spark for two machine learning workloads elaborated as follows.

The first one is the K-nearest neighbors algorithm for classification problems which has been widely adopted in

Authorized licensed use limited to: East China Normal University. Downloaded on December 15,2022 at 03:36:29 UTC from IEEE Xplore. Restrictions apply.

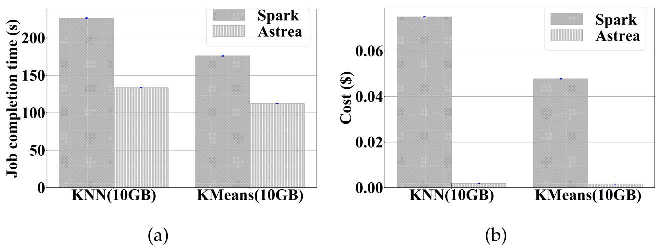


Fig. 13. Job completion time and Monetary cost achieved by *Astrea* and Apache Spark, for two different machine learning jobs.

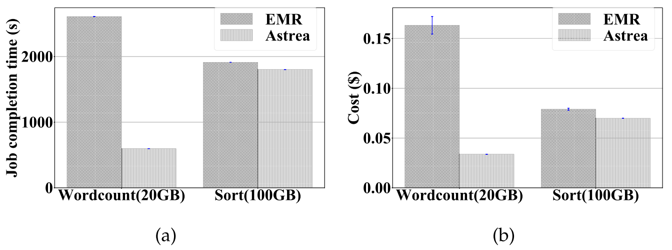


Fig. 14. Job completion time and cost achieved with EMR and *Astrea*, for Wordcount (50 GB) and Sort (25 GB) benchmarks, respectively.

industry. This supervised classification algorithm classifies unlabeled data from labeled inputs by comparing the k numbers of known classified neighbors. The similarities can be calculated based on euclidean distance, Manhattan distance, or Hamming distance. We implement the k-nearest neighbors in *Astrea* following the MapReduce paradigm. More specifically, mappers compute the euclidean distance, and reducers sort the distances and predict using k-closest neighbors. The K-nearest neighbors algorithm is trained on a 10 GB dataset to predict the price of a laptop given its configuration.

The second one is the K-means clustering, an unsupervised machine learning algorithm that discovers patterns by grouping similar data points into K clusters. *Astrea* implements it in the MapReduce form. Mappers accept data and a list of centers (global constant), compute the nearest center for each data, and finally store centers and their data points. Reducers take the former outputs from mappers and compute the new centers based on distance calculation. Each iteration is transformed into a series of reducer steps. We apply K-means clustering on a 10 GB seeds dataset.

The Spark counterparts of these two jobs are implemented in Apache Spark 3.0.1 and deployed on Amazon SageMaker, a commercial machine learning platform. SageMaker provides easy coordination for Spark machine learning workloads, enabling a faster response time and a smaller cost compared to the manual cluster deployment on EC2 instances. In particular, we use an instance-based notebook (`ml.t2.medium` with 2vCPUs hosting a Spark executor) in SageMaker for Spark K-nearest neighbors and Spark K-means clustering. The cost depends on the instance price, data processing cost and storage cost.

As shown in Figs. 13a and 13b, for the K-nearest neighbors algorithm (referred to as KNN in the figure), *Astrea* achieved 41% performance improvement and 97% cost reduction compared to Spark, when aiming at cost minimization given a completion time threshold (186.5 s, while the actual resulted completion time was 153.7 s). In a similar vein, for the K-means clustering (referred to as Kmeans in the figure) workload, *Astrea* led to 36% performance improvement and 97% cost saving compared to Spark, when minimizing cost without violating a performance threshold (148.31 s, while the actual completion time was 120.74 s). Note that these thresholds are specified based on the corresponding job completion time results under a provisioning baseline as aforementioned, which should be fairly reasonable. To summarize, through this set of experiments, we have demonstrated the advantages of *Astrea* in faster execution of machine learning jobs at lower cost, compared to Spark on SageMaker.

5.4 *Astrea versus* Amazon EMR

We further compare *Astrea* with another VM-based solution. We use the Amazon Elastic MapReduce (EMR), a cloud Big Data platform that easily runs and scales frameworks like Hadoop, Hive, *etc.* EMR (version 5.35.0) settings in our evaluations use Hadoop 2.10, three `m3.xlarge` on-demand VM instances (each with 4vCPU, hosting 2 executors), and the number of concurrent mapping tasks is 100. The workloads we used include Wordcount with 20 GB size and Sort with 100 GB. EMR-setting incurs on-demand instance cost, and the resulted latency depends on the setup time of the instances and the execution time of the job till the instance termination. As shown on the left side of Fig. 14, *Astrea* outperforms the VM-based solution by 77.09% and 5.72% for Wordcount and Sort benchmarks, respectively.

The cost comparison plot on the right side of Fig. 14 shows that *Astrea* minimizes the cost, with 79.33% and 11.52% cost savings over the EMR solution, for the same two benchmarks, respectively.

5.5 Multi-Round *Astrea*

While the previous experiments demonstrate the advantages of *Astrea* over various workloads with increasing scales, this subsection further evaluates how our solution accommodates to even larger scales of workloads beyond the platform limits for one-round deployment. More specifically, we evaluate our multi-round extension of *Astrea*, i.e., Algorithm 2 in Section 4.3, under two scenarios: *maximum concurrency limit* and *maximum temporal storage limit*.

In the first evaluation for a 50 GB Wordcount, Algorithm 1 gave feasible solution when the limit of maximum concurrent lambdas is doubled. The first round of Algorithm 2 resulted in the configuration decisions of 1450 mappers and four objects per mapper. *Astrea* suggested two rounds to finish the job, since the current platform concurrency limit is 1000. 69% of objects followed the first round configurations and the remaining followed the second one. For cost minimization with performance limit, the performance threshold was divided for two rounds based on the sizes of objects of particular rounds. Solutions were obtained by solving optimization problems for the subjobs, and finally *Astrea* executed the job on AWS Lambda platform over two rounds accordingly. As shown in Figs. 15a and 15b (first bar group), when minimizing the cost subject to the performance threshold 1824 s (the completion time resulting from a baseline provisioning which splits the objects by half), *Astrea* exhibited 89% cost reduction and 3% performance improvement compared to Apache Spark, which followed the EC2 deployment in Section 5.3.1.

Authorized licensed use limited to: East China Normal University. Downloaded on December 15,2022 at 03:36:29 UTC from IEEE Xplore. Restrictions apply.

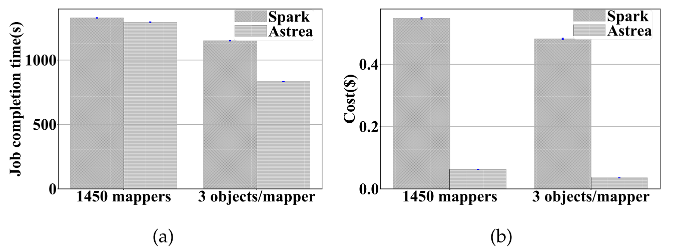


Fig. 15. Job completion time and cost achieved with Spark and *Astrea*, for Wordcount jobs with 50 GB and 25 GB input data, respectively.

To evaluate the second case when the temporary storage required by one or more lambdas exceeded the platform limit, we used a 25 GB Wordcount job. *Astrea* generated the setting of 34 mappers, each with three objects. Depending on the sizes of the objects, there were three lambdas that required extra storage beyond the platform limitation. To address this issue, Algorithm 2 swapped objects across mapper lambdas based on their sizes to reach a better match. The second bar groups in Figs. 15a and 15b present the results of job completion time and monetary cost achieved by *Astrea* in comparison with Spark, when minimizing the cost under the performance threshold of 1087 s (from a baseline provisioning). As clearly observed, *Astrea* outperformed with 28% performance improvement and 92% cost reduction.

5.6 Discussion

Intuitively, if DAG fully characterizes the choices, dependencies and impacts of the configuration sequences along the job workflow, deriving the shortest path results in the optimal execution and configuration with the objectives of the minimum job completion time or the cost. As *Astrea* sees more types of workloads, the modeling and DAG construction could be dynamically adjusted and refined to achieve better accuracy. The overhead of *Astrea* is incurred by our algorithm to solve the constrained optimization problem formulated given user requirements, which is within a few seconds on a laptop (Intel®Core™i7-8750H CPU@2.20 GHz×12, 2×8GiB memory). It is expected that the running time is negligible (in milliseconds) on a more powerful commodity server. Though implemented in AWS Lambda, *Astrea* can be adapted to Google Functions and Azure Functions by using their respective platform quotas and pricing mechanisms.

Astrea relies on a general modeling which accounts for the completion time (as well as the monetary cost) of each function resulting from both computation and I/O (communication), no matter which resource is more intensively demanded. This brings one of *Astrea*'s advantages: the ease of usage without the need of categorizing applications according to their resource intensiveness. On the other hand, *Astrea* currently relies on S3 for the exchange of intermediate data, which is cheaper but slower compared to other commercial caching services (such as AWS ElastiCache) or the academic prototyped far-memory systems (such as Pocket [9] and Jiffy [29]). When an application is explicitly considered as shuffle heavy, we could extend our modeling and implementation beyond S3, analyzing the characteristics and costs of alternative storage/caching services, to judiciously coordinate a mixture of services with different performance-cost tradeoffs. Building high-throughput low-latency far-memory systems for serverless analytics is an orthogonal direction. Direct function-to-function communication is also an open topic to be explored. By adopting finer-grained task-level scheduling approaches such as NIMBLE [28] and Wukong [47], we may expect further performance enhancement of *Astrea*. Finally, *Astrea* is suitable for other data analytics workloads which are directly in or convertible to the MapReduce form. This is evidenced by our experiments with Wordcount, SQL and machine learning workloads in Spark, where *Astrea* achieves 92% cost reduction without performance degradation over VM-based vanilla Spark.

6 Conclusion and Future Work

This paper presents an optimization framework, *Astrea*, to navigate the cost-performance tradeoff for serverless analytics jobs. *Astrea* relies on the modeling of completion time performance and monetary cost of a job to formulate optimization problems towards user-specified objectives. *Astrea* identifies the optimal solutions of resource configuration and Lambda function orchestration based on graph theory, to either minimize the job completion time with a budget limit, or minimize monetary cost with a performance threshold. We have implemented and deployed *Astrea* in AWS Lambda. Our experimental results with three representative benchmarks have demonstrated the effectiveness of *Astrea* in optimal resource provisioning: *Astrea* achieves 21% to 60% performance improvement without exceeding the budget constraint, and 20% to 80% cost reduction without violating the QoS objective. The optimality of *Astrea* has also been demonstrated in the comparison with Apache Spark: *Astrea* exhibits almost 92% cost reduction without degrading performance for serverless data analytics benchmarks. For machine learning analytics, *Astrea* achieves a cost reduction up to 98% with at least 37% performance improvement. Our extended multi-round design and implementation of *Astrea* also performs better than Apache Spark with 92% cost reduction.

In the future, we plan to explore the directions of incorporating better intermediate data transfer strategy and exploiting optimal task level orchestration for large-scale data analytics jobs, as discussed in Section 5.6. For example, a better scheduler can be designed to invoke functions of different stages in a pipelined manner to speed up the workflow. In a more general background, serverless computing has been envisioned to be promising as the next phase of cloud computing to revolutionize cloud programming [48]. There are future directions on stateful serverless [49] like improving and inventing logging mechanisms [50] for the shared log records and fault-tolerance tools [51] to identify bugs in serverless applications easily. On the other hand, with the transformative effects of Internet-of-Things (IoT), Blockchain and Artificial Intelligence on the cloud [52], it remains open to fulfill the potential of serverless computing to support a broader range of cloud applications. More specifically, in the paradigm of IoT, research challenges include where to place serverless functions (across edge, fog, and cloud layers), how to offload and dispatch functions, how to mitigate cold start to reduce latency for mission-critical applications, and *etc.* Last but not least, aligned with the concept of "sky computing" [53] and to avoid vendor lock-in, developing

Authorized licensed use limited to: East China Normal University. Downloaded on December 15,2022 at 03:36:29 UTC from IEEE Xplore. Restrictions apply.

solutions for cross-provider aggregation of serverless offerings, such as [54], could be an interesting direction to pursue.

References

[1] AWS Lambda, 2020. [Online]. Available: https://aws.amazon.com/lambda/

[2] Cloud functions, 2020. [Online]. Available: https://cloud.google.com/functions

[3] Azure functions, 2020. [Online]. Available: https://azure.microsoft.com/en-us/services/functions/

[4] S. Fouladi *et al.*, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in *Proc. 14th USENIX Symp. Networked Syst. Des. Implementation*, 2017, pp. 363–376.

[5] AWS IoT greengrass, 2020. [Online]. Available: https://aws.amazon.com/greengrass/

[6] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in *Proc. 16th USENIX Symp. Networked Syst. Des. Implementation*, 2019, pp. 193–206.

[7] Amazon S3, 2020. [Online]. Available: https://aws.amazon.com/s3/

[8] Amazon ElastiCache, 2020. [Online]. Available: https://aws.amazon.com/elasticache/

[9] A. Klimovic *et al.*, "Pocket: Elastic ephemeral storage for serverless analytics," in *Proc. 13th USENIX Symp. Oper. Syst. Des. Implementation*, 2018, pp. 427–444.

[10] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *Proc. ACM Symp. Cloud Comput.*, 2017, pp. 445–451.

[11] V. Giménez-Alventosa, G. Moltó, and M. Caballer, "A framework and a performance assessment for serverless MapReduce on AWS lambda," *Future Gener. Comput. Syst.*, vol. 97, pp. 259–274, 2019.

[12] M. Yu, Z. Jiang, H. C. Ng, W. Wang, R. Chen, and B. Li, "Gillis: Serving large neural networks in serverless functions with automatic model partitioning," in *Proc. IEEE 41st Int. Conf. Distrib. Comput. Syst.*, 2021, pp. 138–148.

[13] AWS lambda limits, 2020. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html

[14] Big data benchmark, 2014. [Online]. Available: https://amplab.cs.berkeley.edu/benchmark/

[15] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf. Hot Topics Cloud Comput.*, 2010, Art. no. 10.

[16] Amazon EC2 on-demand pricing, 2020. [Online]. Available: https://aws.amazon.com/ec2/pricing/on-demand/

[17] Amazon SageMaker pricing, 2020. [Online]. Available: https://aws.amazon.com/sagemaker/pricing/

[18] Amazon EMR, 2014. [Online]. Available: https://aws.amazon.com/emr/

[19] B. Carver, J. Zhang, A. Wang, and Y. Cheng, "In search of a fast and efficient serverless DAG engine," in *Proc. IEEE/ACM 4th Int. Parallel Data Syst. Workshop*, 2019, pp. 1–10.

[20] S. Fouladi *et al.*, "From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 475–488.

[21] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A serverless framework for end-to-end ML workflows," in *Proc. ACM Symp. Cloud Comput.*, 2019, pp. 13–24.

[22] S. Nastic *et al.*, "A serverless real-time data analytics platform for edge computing," *IEEE Internet Comput.*, vol. 21, no. 4, pp. 64–71, Jul. 2017.

[23] B. Liston, "Ad hoc Big Data processing made simple with serverless MapReduce," 2016. [Online]. Available: https://aws.amazon.com/blogs/compute/ad-hoc-big-data-processing-made-simple-with-serverless-mapreduce/

[24] AWS lambda pricing, 2020. [Online]. Available: https://aws.amazon.com/lambda/pricing/

[25] Y. Kim and J. Lin, "Serverless data analytics with flint," in *Proc. IEEE 11th Int. Conf. Cloud Comput.*, 2018, pp. 451–455.

[26] Amazon simple queue service, 2020. [Online]. Available: https://aws.amazon.com/sqs/

[27] W. Zhang, V. Fang, A. Panda, and S. Shenker, "Kappa: A programming framework for serverless computing," in *Proc. 11th ACM Symp. Cloud Comput.*, 2020, pp. 328–343.

[28] H. Zhang, Y. Tang, A. Khandelwal, J. Chen, and I. Stoica, "Caerus: NIMBLE task scheduling for serverless analytics," in *Proc. 18th USENIX Symp. Networked Syst. Des. Implementation*, 2021, pp. 653–669.

[29] A. Khandelwal, Y. Tang, R. Agarwal, A. Akella, and I. Stoica, "Jiffy: Elastic far-memory for stateful serverless analytics," in *Proc. 17th Eur. Conf. Comput. Syst.*, 2022, pp. 697–713.

[30] P. K. Gadepalli, G. Peach, L. Cherkasova, R. Aitken, and G. Parmer, "Challenges and opportunities for efficient serverless computing at the edge," in *Proc. 38th Symp. Reliable Distrib. Syst.*, 2019, pp. 261–2615.

[31] I. Müller, R. Marroquín, and G. Alonso, "Lambada: Interactive data analytics on cold data using serverless cloud infrastructure," in *Proc. ACM Int. Conf. Manage. Data*, 2020, pp. 115–130.

[32] A. Wang *et al.*, "InfiniCache: Exploiting ephemeral serverless functions to build a cost-effective memory cache," in *Proc. 18th USENIX Conf. File Storage Technol.*, 2020, pp. 267–281.

[33] Z. Li and *et al.*, "Amoeba: QoS-Awareness and reduced resource usage of microservices with serverless computing," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2020, pp. 399–408.

[34] H. Wang, D. Niu, and B. Li, "Distributed machine learning with a serverless architecture," in *Proc. IEEE Conf. Comput. Commun.*, 2019, pp. 1288–1296.

[35] F. Xu, Y. Qin, L. Chen, Z. Zhou, and F. Liu, "λDNN: Achieving predictable distributed DNN training with serverless architectures," *IEEE Trans. Comput.*, vol. 71, no. 2, pp. 450–463, Feb. 2022.

[36] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Batch: Machine learning inference serving on serverless platforms with adaptive batching," in *Proc. IEEE Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2020, pp. 972–986.

[37] J. Jarachanthan, L. Chen, F. Xu, and B. Li, "AMPS-Inf: Automatic model partitioning for serverless inference with cost efficiency," in *Proc. 50th Int. Conf. Parallel Process.*, 2021, Art. no. 14.

[38] J. Jiang *et al.*, "Towards demystifying serverless machine learning training," in *Proc. ACM Int. Conf. Manage. Data*, 2021, pp. 857–871.

[39] J. Thorpe *et al.*, "Dorylus: Affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads," in *Proc. 15th USENIX Symp. Oper. Syst. Des. Implementation*, 2021, pp. 495–514.

[40] A. Bhattacharjee, A. D. Chhokra, Z. Kang, H. Sun, A. Gokhale, and G. Karsai, "Barista: Efficient and scalable serverless serving system for deep learning prediction services," in *Proc. IEEE Int. Conf. Cloud Eng.*, 2019, pp. 23–33.

[41] T. Rausch, A. Rashed, and S. Dustdar, "Optimized container scheduling for data-intensive serverless edge computing," *Future Gener. Comput. Syst.*, vol. 114, pp. 259–271, 2021.

[42] AWS step functions, 2020. [Online]. Available: https://aws.amazon.com/step-functions/

[43] Amazon S3 pricing, 2020. [Online]. Available: https://aws.amazon.com/s3/pricing/

[44] M. Conforti, G. Cornuéjols, and G. Zambelli, *Integer Programming*, Berlin, Germany: Springer, 2014, Art. no. 455.

[45] B. Goldengorin, *Optimization Problems in Graph Theory*, vol. 139, Berlin, Germany: Springer, 2018.

[46] A. A. Zoobi, D. Coudert, and N. Nisse, "Space and time trade-off for the k shortest SimplePaths problem," in *Proc. 18th Int. Symp. Exp. Algorithms*, 2020, Art. no. 13.

[47] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng, "Wukong: A scalable and locality-enhanced framework for serverless parallel computing," in *Proc. ACM Symp. Cloud Comput.*, 2020, pp. 1–15.

[48] J. Schleier-Smith *et al.*, "What serverless computing is and should become: The next phase of cloud computing," *Communication ACM*, vol. 64, no. 5, pp. 76–84, 2021.

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

[50] Z. Jia and E. Witchel, "Boki: Stateful serverless computing with shared logs," in *Proc. ACM 28th Symp. Oper. Syst. Princ.*, 2021, pp. 691–707.

[51] K. Alpernas, A. Panda, L. Ryzhyk, and M. Sagiv, "Cloud-scale runtime verification of serverless applications," in *Proc. ACM Symp. Cloud Comput.*, 2021, pp. 92–107.

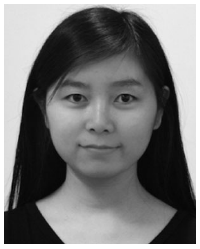
[52] S. S. Gill *et al.*, "Transformative effects of IoT, blockchain and artificial intelligence on cloud computing: Evolution, vision, trends and open challenges," *Internet Things*, vol. 8, pp. 100–118, 2019.

[53] I. Stoica and S. Shenker, "From cloud computing to sky computing," in *Proc. Workshop Hot Topics Oper. Syst.*, 2021, pp. 26–32.

[54] A. F. Baarzi, G. Kesidis, C. Joe-Wong, and M. Shahrad, "On merits and viability of multi-cloud serverless," in *Proc. ACM Symp. Cloud Comput.*, 2021, pp. 600–608.

Authorized licensed use limited to: East China Normal University. Downloaded on December 15,2022 at 03:36:29 UTC from IEEE Xplore. Restrictions apply.

Jananie Jarachanthan is currently working towards the PhD degree with the School of Computing and Informatics, University of Louisiana at Lafayette. Her research interests include cloud computing, distributed systems, resource allocation and scheduling.



Li Chen (Member, IEEE) received the BEngr degree from the Department of Computer Science and Technology, Huazhong University of Science and Technology, China, in 2012, the MASc degree, in January 2015, and the PhD degree from the Department of Electrical and Computer Engineering, University of Toronto, in July 2018. She is an assistant professor with the Department of Computer Science, School of Computing and Informatics, University of Louisiana at Lafayette. Her research interests include Big Data analytics, machine learning systems, cloud computing, datacenter networking, resource allocation and scheduling in networked systems.

Fei Xu (Member, IEEE) received the PhD degree in computer science and engineering from the Huazhong University of Science and Technology, Wuhan, China, in 2014. He received Outstanding Doctoral Dissertation Award in Hubei province, China, and ACM Wuhan & Hubei Computer Society Doctoral Dissertation Award, in 2015. He is currently an associate professor with the School of Computer Science and Technology, East China Normal University, Shanghai, China. His research interests include cloud computing and datacenter, virtualization technology, and distributed systems.

Bo Li (Fellow, IEEE) received the BEng (summa cum laude) in the computer science from Tsinghua University, Beijing, China, and the PhD degree in the electrical and computer engineering from University of Massachusetts at Amherst. He is a chair professor with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology. He held a Cheung Kong Visiting chair professor in Shanghai Jiao Tong University between 2010 and 2016. He was an adjunct researcher with the Microsoft Research Asia (MSRA) (1999-2006) and with the Microsoft Advanced Technology Center (2007-2009). He made pioneering contributions in multimedia communications and the Internet video broadcast, in particular the Coolstreaming system, which was credited as first large-scale Peer-to-Peer live video streaming system in the world. It attracted significant attention both from industry with substantial VC investment, and from academia in receiving the Test-of-Time Best Paper Award from IEEE INFOCOM (2015). He received 6 best paper awards from IEEE including INFOCOM (2021). He has been an editor or a guest editor for over a two dozen of IEEE and ACM journals and magazines. He was the co-TPC chair for IEEE INFOCOM 2004.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: East China Normal University. Downloaded on December 15,2022 at 03:36:29 UTC from IEEE Xplore. Restrictions apply.