Astra: Autonomous Serverless Analytics with Cost-Efficiency and QoS-Awareness

Jananie Jarachanthan*, Li Chen*, Fei Xu†, Bo Li‡
*University of Louisiana at Lafayette, †East China Normal University,
‡Hong Kong University of Science and Technology
{jananie.jarachanthan1, li.chen}@louisiana.edu, fxu@cs.ecnu.edu.cn, bli@cse.ust.hk

***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 with the difficulty in coordinating computation across different stages and provisioning resources in a large configuration space. This paper presents our design and implementation of *Astra*, which configures and orchestrates serverless analytics jobs in an autonomous manner, while taking into account flexibly-specified user requirements. *Astra* 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 optimal job execution. We deploy *Astra* in the AWS Lambda platform and conduct real-world experiments over three representative benchmarks with different scales. Results demonstrate that *Astra* can achieve the optimal execution decision for serverless analytics, by improving the performance of 21% to 60% under a given budget constraint, and resulting in a cost reduction of 20% to 80% without violating performance requirement, when compared with three baseline configuration algorithms.**

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

I. 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.*

The research was supported in part by grant from Louisiana Board of Regents under the contract LEQSF(2019-22)-RD-A-21, NSFC grant under No. 61972158, RGC RIF grant R6021-20, and RGC GRF grants under the contracts 16207818 and 16209120.

However, when adapting data-intensive analytics applications (*e.g.,* MapReduce, Spark jobs) in 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]–[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 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) or incurring extra billing cost which could have been avoided by a better configuration. Essentially, users still encounter with the critical challenge of serverless provisioning for big data analytics: *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*

1530-2075/21/$31.00 ©2021 IEEE
DOI 10.1109/IPDPS49936.2021.00085

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 automatically and optimally configure and orchestrate the serverless analytics jobs, according to flexibly-specified requirements from users, which is the focus of this work.

We design a general framework, called *Astra*, which automatically configures and orchestrates lambda functions for data analytics jobs to navigate the tradeoff between performance and cost. *Astra* 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, 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. Building upon the model, *Astra* obtain 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 requirement, a configuration and job execution optimization is formulated with the objective of minimizing monetary cost.

We have implemented and deployed *Astra* on Amazon Lambda and evaluated its performance with real-world experiments on various workloads, including Wordcount with different input sizes, Sort, and Query over the Uservisits dataset [12]. Upon the submission of an analytics job, *Astra* calculates the best configuration for resource allocation and task assignment by solving an optimization problem towards a specific objective. Extensive experimental results have demonstrated that *Astra* 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, *Astra* achieves the performance improvement of about 50% to 60% for Wordcount benchmarks with three different input sizes, up to 21% for Sort, and at least 50% improvement for Query benchmark. With respect to cost, a reduction up to 80% is achieved for Wordcount, up to 21% reduction for Sort, and at least 20% reduction for Query benchmark. As evidenced, *Astra* successfully navigates the tradeoff between job completion time and monetary cost according to flexible requirements, outperforming existing solutions which are either suboptimal or incomplete.

The rest of the paper is organized as follows. Sec. II presents the background of serverless analytics and examines the intricate interplay among cost and performance factors. Sec. III models the performance and monetary cost for a MapReduce job in the serverless platform. Sec. IV designs algorithms for *Astra* to optimize job completion time or monetary cost. Sec. V implements *Astra* and demonstrates its advantages over three baselines with real-world experiments. Sec. VI discusses the related work and Sec. VII presents concluding remarks.

II. Background and Motivation

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

A. 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 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 [13], software compilation [14], machine learning [15], 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 [16]. Moreover, it has also been shown that data analytics applications which require concurrent handling of massive data are feasible in serverless computing [17].

B. 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 1000 lambdas, 512MB of temporary storage and 900 seconds of timeout [18], 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

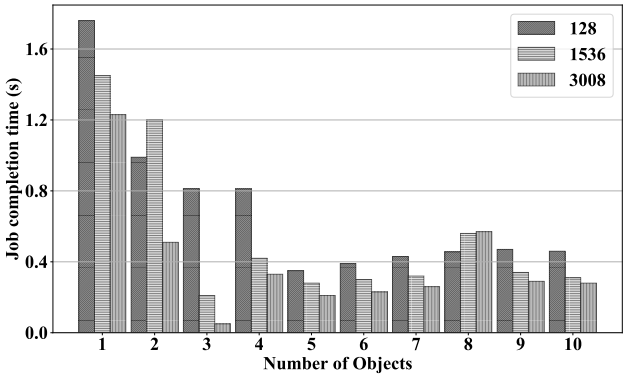

Fig. 1: Job completion time with the number of objects processed per lambda in three types of memory allocation.

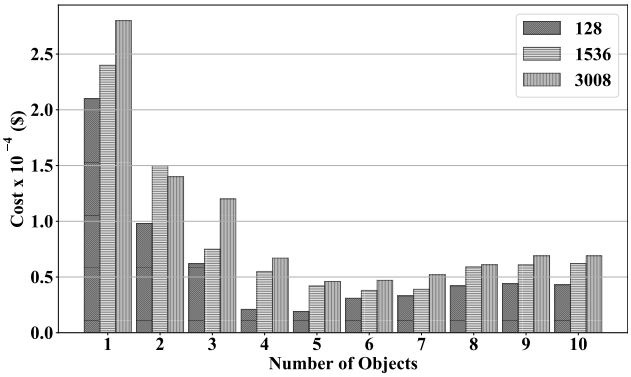

Fig. 2: Monetary cost with the number of objects processed per lambda in three types of memory allocation.

mapping and reducing [19]. 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.

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.

C. 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 [19], with a total of 10 objects with 2MB 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 I presents five orchestration examples, when we vary the number of objects handled by each function.

TABLE I: 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)	-	-	-	-	-

For instance, as shown in the second column entry in Table I, 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. 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 [20].

Fig. 1 and Fig. 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 when the slowest mapper finishes, 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

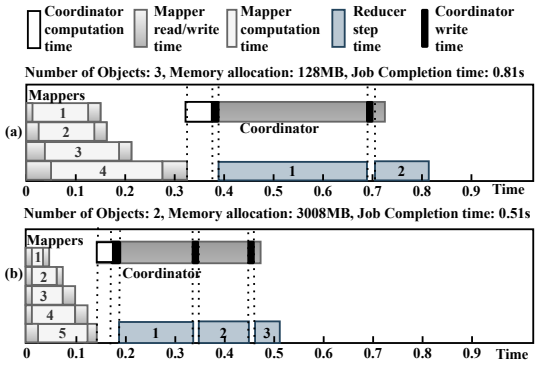

Fig. 3: Job timeline with two sample configurations.

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.

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 I, 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 Fig. 3(a) to Fig. 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 *Astra*, to automate the deployment of

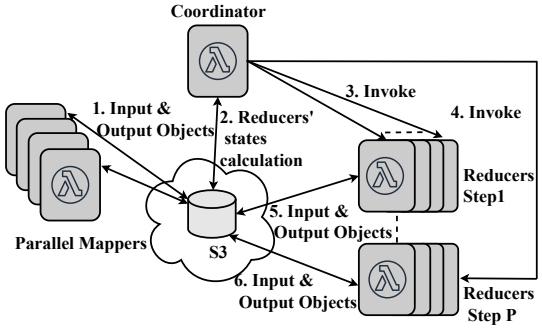

Fig. 4: Workflow of a serverless MapReduce job in AWS Lambda. The job uses AWS S3 as the intermediate storage.

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 inter-dependencies in the next section, and formulate the optimization problems with flexible objectives and constraints.

III. 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[1] and reduce, respectively, as shown in Fig. 4. Our modeling is easily adapted to a general serverless data analytics setting, to be discussed later.

A. Performance of Completion Time

1) Lifetime of Mappers: As illustrated by Fig. 3 in Section II, 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 $(d_m^j + e_m^j)/B$.

[1]An alternative to the coordinate lambda is to use AWS step functions [21], 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 *Astra*.

TABLE II: 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

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 128MB to 3008MB in 64MB increments, which impacts its processing speed in proportion. We use binary variable $x_i, (i = 1, 2, \cdots, L)$ to specify whether the i-th type of memory is allocated (for mapper lambdas) out of the L categories. Intuitively, we have

$$x_i \in \{0, 1\}, \quad \forall i \in \{1, 2, \cdots, L\}; \qquad \sum_{i=1}^{L} x_i = 1, \qquad (1)$$

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

$$n_j \in \{0, 1\}, \quad \forall j \in \{1, 2, \cdots, \lambda_M\}; \quad \sum_{j=1}^{\lambda_M} n_j = 1. \qquad (2)$$

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

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

where u_i is the processing time of unit-size object given i-th 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:

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

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).

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 $p, (p = 1, 2, \cdots, P)$. 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 128MB to 3008MB, we use the binary variable $y_a, (a = 1, 2, \cdots, L)$ to specify whether the a-th memory allocation is chosen:

$$y_a \in \{0, 1\}, \quad \forall a \in \{1, 2, \cdots, L\}; \quad \sum_{a=1}^{L} y_a = 1. \qquad (5)$$

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 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:

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

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 II, 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, resulted 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 $z_s, (s = 1, 2, \cdots, L)$ to specify whether the s-th memory allocation is selected for reducer lambdas or not, which naturally has the following constraints:

$$z_s \in \{0, 1\}, \quad \forall s \in \{1, 2, \cdots, L\}; \quad \sum_{s=1}^{L} z_s = 1. \qquad (7)$$

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

$$w_g \in \{0, 1\}, \quad \forall g \in \{1, 2, \cdots, \lambda_M\}; \quad \sum_{g=1}^{\lambda_M} w_g = 1. \qquad (8)$$

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 II. 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:

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

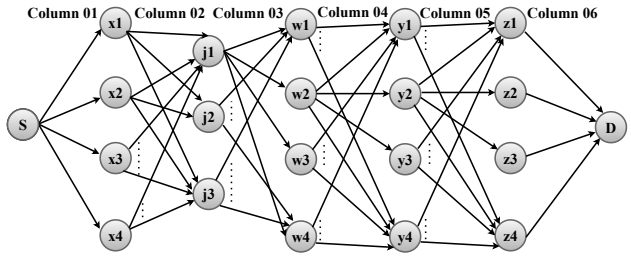

Fig. 5: DAG for performance optimization (Eq. (16)) or cost minimization (Eq. (20)), depending on the associated edge weights from Column 01 to Column 06.

where u_s is the processing time of unit-size object given s-th 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:

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

B. 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.

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 [22] of $\$0.005$ per 1000 Put requests (F) and $\$0.004$ per 10000 Get requests (G) in S3, we have the following expressions of S3 requests cost for each phase:

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

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:

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

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 [20] 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.

IV. 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.

A. 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:

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

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

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

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

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 5TB) 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 [23]. To solve this problem, we propose a strategy based on graph theory [24], 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 [25].

B. 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.

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

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

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

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).

Algorithm 1 Astra: 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.

```
procedure FIND-OPTIMAL-PATH(G(v, E))
    P ← Dijkstra(G, W, F)
    cost ← 0, storage ← 0, u ← S̄
    while u ≠ D̄ do
        cost ← cost + C(u, v), storage ← storage + F(u, v)
        if cost ≥ budget then
            E ← E − E[v][u]
            P ← Find-optimal-path (G(v, E))
        else
            u ← v
    return P
```

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 resulted 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, Alg. 1 can be used to identify the shortest path as the optimal solution.

V. Performance Evaluation

In this section, we present the design and implementation of *Astra*, and evaluate its performance with real-world experiments.

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

Experimental Setup. 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 64MB increments [18] in AWS Lambda. Each lambda has a limit of 900 seconds for execution.

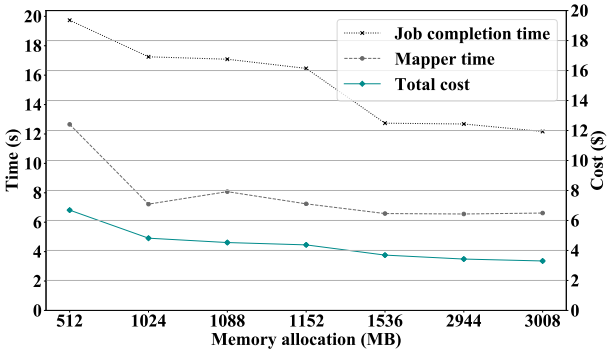

Fig. 6: The change of completion time, mapper phase time and monetary cost with memory allocation.

There are three baselines implemented to compare with *Astra*, 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, and the number of objects per mapper is set as 1 to realize the maximum degree of parallelism for mappers. We randomly allocate the number of objects per reducer as 2. Baseline 2 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 1. The third baseline has a hybrid consideration of both performance and cost. It follows the same setting as Baseline 2 for parallel mappers, each with 128 MB to process one object. For the reducing phase, Baseline 3 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 three different workloads. The big data benchmarks include: i) Query over the uservisits dataset [12] with the size of 25.4 GB, stored in S3 as 202 objects. The dataset has 155 million individual rows, each consisting of sourceIP, visitDate, adRevenue, userAgent, countryCode, languageCode, searchWord and duration. ii) Wordcount, with the input sizes of 1 GB, 10 GB and 20 GB, respectively. iii) Sort, with 100 GB.

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

Fig. 7 presents the completion time achieved by *Astra*, in comparison with the three baselines, for different workloads. The budget constraints and the resulted costs (by *Astra*) are shown with 2-tuples above the bar groups for each benchmark. As clearly shown in Fig. 7, *Astra* outperforms all the three baselines in terms of reducing the completion time for all the workloads, without exceeding budgets. Baseline 1, 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 *Astra*. More specifically, for the Wordcount benchmark with increasing scales of 1GB, 10GB, and 20GB, *Astra* achieves performance improvement over Baseline 1 of 46.27%, 42.14%, and 54.73%, respectively. Similarly, for Query and Sort, *Astra* outperforms Baseline 1 by at least 49.45% and 9.36%, respectively. Considering all the three baselines, *Astra* achieves $42 - 68\%$ improvement for Wordcount in all scales, up to 21% for Sort, and 57% for Query benchmark, respectively. To further illustrate how *Astra* works and analyze its advantages, Table III presents the budget-constrained performance-optimal resource provisioning in *Astra*, 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.

TABLE III: The resource allocations achieved by *Astra*, for the three benchmarks when optimizing job performance.

	Wordcount (1GB)	Wordcount (10GB)	Wordcount (20GB)	Sort (100GB)	Query (25.4GB)
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

For the Query benchmark, the number of objects per mapper is allocated as 1 by *Astra*, 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. *Astra* 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. *Astra* 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, *Astra* sets 8

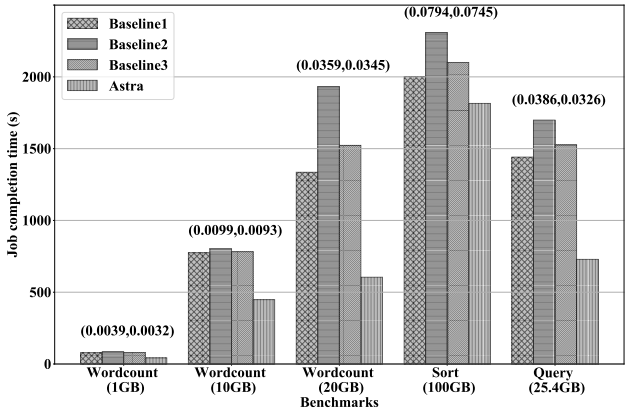

Fig. 7: Job completion time achieved by *Astra* and three baselines, for Wordcount (3 scales), Sort and Query workloads, given a budget constraint.

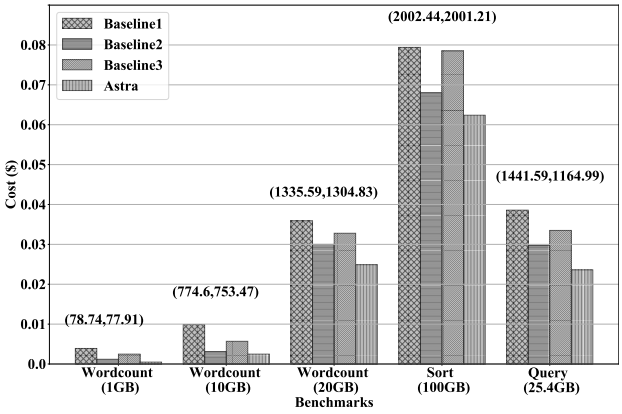

Fig. 8: Monetary cost achieved by *Astra* and three baselines, for Wordcount (3 scales), Sort and Query workloads, given a completion time threshold.

objects per reducer, each with 1024 MB, to finish within 1 step, as the outcome from optimizing completion time given the budget.

Similarly, when aiming at minimizing the monetary cost incurred by the job given a particular threshold of job completion time, *Astra* 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 *Astra*. It is easily verified that without exceeding the threshold, *Astra* results in the smallest cost for each benchmark. Baseline 2 is intuitively designed for cost saving, and thus results in a smaller cost than the other two baselines for all the workloads. Still, *Astra* achieves nearly 58%, 19%, and 17% cost reduction over Baseline 2, for the three Wordcount benchmarks. For Sort and Query benchmarks, about 8% and 20% cost savings are exhibited, compared to Baseline 2. In summary, compared with the three baselines, *Astra* achieves the cost reduction of at least 20%, up to 87% for those three different benchmarks.

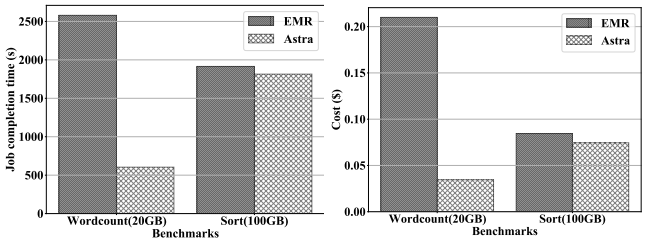

Fig. 9: Job completion time and cost achieved with EMR and *Astra*, for Wordcount (20 GB) and Sort (100 GB) benchmarks, respectively.

Finally, we compare *Astra* with the VM-based solution. We use the Amazon Elastic MapReduce (EMR) with three `m3.xlarge` on-demand VM instances, and the number of concurrent mapping tasks is 100. The workloads include Wordcount with 20GB input and Sort with 100GB. As shown in the left side of Fig. 9, *Astra* outperforms the VM-based solution by 76.56% and 5.22% for Wordcount and Sort benchmarks, respectively. The right side of Fig. 9 shows that *Astra* minimizes the cost, with 64.52% and 11.83% cost savings over the EMR solution, for the same two benchmarks, respectively.

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 *Astra* sees more types of workloads, the modeling and DAG construction could be dynamically adjusted and refined to achieve better accuracy. The overhead of *Astra* 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@2.20GHz×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, *Astra* can be adapted to Google Functions and Azure Functions by using their respective platform quotas and pricing mechanisms. *Astra* relies on S3 for the exchange of intermediate data. When other types of data storage are considered for intermediate data, such as serverless databases (AWS Aurora) or serverless in-memory data storage (AWS ElasticCache), our modeling needs to be adjusted by analyzing the characteristics and cost of the particular storage. Function-to-function communication is also an open topic to be explored. Finally, *Astra* is suitable for other data analytics workloads which are directly in or convertible to the MapReduce form. This is evidenced by our preliminary experiments with WordCount and SQL (aggregation query) workloads in Spark, where *Astra* achieves at least 92% cost reduction without performance degradation over VM-based vanilla Spark.

VI. Related Works

Function as a service (FaaS) is popular at present where users can deploy and run their applications without worrying about the infrastructure. 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], [15].

PyWren [10] presents a prototype to execute MapReduce jobs with lambda functions, using S3 as intermediate data storage. Similarly, Flint [26] enhances the PySpark MapReduce framework in the serverless environment, and leverages the Amazon Simple Queue Service (SQS) [27] for the shuffling of intermediate data. Extending the idea of Elastic MapReduce (EMR) [28], Amazon AWS presented a serverless architecture [19] 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.

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. Locus [6] leverages a small number of expensive fast ElastiCache (Redis) [8] instances combined with the much cheaper S3 service. Gadepalli, *et al.* [29] applied serverless computing at the edge at near-native speed, while having a small memory footprint and optimized invocation time. InfiniCache [30] presents the first in-memory object cache for serverless functions to improve I/O performance. Amoeba [31] switches between the IaaS-based and serverless-based deployment by monitoring loads and predicting the CPU and memory usage of these platforms. Different from all the existing works, we present a framework to automatically configure and orchestrate the MapReduce job in serverless environment towards flexibly specified objectives. 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 [32], that operate within the serverless platform from the provider perspective.

VII. Concluding Remarks

This paper presents an optimization framework, *Astra*, to navigate the cost-performance tradeoff for serverless analytics jobs. *Astra* relies on the modeling of completion time performance and monetary cost of a job to formulate optimization problems towards user-specified objectives. *Astra* 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 *Astra* in AWS Lambda. Our experimental results with three representative benchmarks have demonstrated the effectiveness of *Astra* in optimal resource provisioning: *Astra* achieves 21% to 60% performance improvement without exceeding the budget constraint, and 20% to 80% cost reduction without violating the QoS objective.

References

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

[2] (2020) Cloud Functions. [Online]. Available: https://cloud.google.com/functions

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

[4] S. Fouladi and *et al.*, "Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[5] (2020) AWS IoT Greengrass. [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 *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

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

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

[9] A. Klimovic, Y. Wang, and *et al.*, "Pocket: Elastic Ephemeral Storage for Serverless Analytics," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[10] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the Cloud: Distributed Computing for the 99%," in *Proceedings of the 2017 Symposium on Cloud Computing*, 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 Generation Computer Systems*, vol. 97, pp. 259–274, 2019.

[12] (2014) Big Data Benchmark. [Online]. Available: https://amplab.cs.berkeley.edu/benchmark/

[13] B. Carver, J. Zhang, A. Wang, and Y. Cheng, "In Search of a Fast and Efficient Serverless DAG Engine," in *Fourth International Parallel Data Systems Workshop (PDSW)*. IEEE, 2019, pp. 1–10.

[14] S. Fouladi, F. Romero, and *et al.*, "From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers," in *USENIX Annual Technical Conference (USENIX ATC)*, 2019.

[15] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: a Serverless Framework for End-to-end ML Workflows," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 13–24.

[16] S. Nastic, T. Rausch, and *et al.*, "A Serverless Real-Time Data Analytics Platform for Edge Computing," *IEEE Internet Computing*, vol. 21, no. 4, pp. 64–71, 2017.

[17] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless Computing: A Survey of Opportunities, Challenges and Applications," *arXiv:1911.01296v3*, 2020.

[18] (2020) AWS Lambda Limits. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html

[19] B. Liston. (2016) Ad Hoc Big Data Processing Made Simple with Serverless MapReduce. [Online]. Available: https://aws.amazon.com/blogs/compute/ad-hoc-big-data-processing-made-simple-with-serverless-mapreduce/

[20] (2020) AWS Lambda Pricing. [Online]. Available: https://aws.amazon.com/lambda/pricing/

[21] (2020) AWS Step Functions. [Online]. Available: https://aws.amazon.com/step-functions/

[22] (2020) Amazon S3 pricing. [Online]. Available: https://aws.amazon.com/s3/pricing/

[23] M. Conforti, G. Cornuéjols, and G. Zambelli, *Integer Programming*. Springer, 2014, p. 455.

[24] B. Goldengorin, *Optimization Problems in Graph Theory*. Springer, 2018.

[25] A. A. Zoobi, D. Coudert, and N. Nisse, "Space and Time Trade-Off for the k Shortest SimplePaths Problem," Ph.D. dissertation, Inria & Université Cote d'Azur, CNRS, I3S, Sophia Antipolis, France, 2020.

[26] Y. Kim and J. Lin, "Serverless Data Analytics with Flint," in *11th International Conference on Cloud Computing (CLOUD)*. IEEE, 2018.

[27] (2020) Amazon Simple Queue Service. [Online]. Available: https://aws.amazon.com/sqs/

[28] (2014) Amazon EMR. [Online]. Available: https://aws.amazon.com/emr/

[29] P. K. Gadepalli, G. Peach, and *et al.*, "Challenges and Opportunities for Efficient Serverless Computing at the Edge," in *38th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2019.

[30] A. Wang, J. Zhang, and *et al.*, "InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache," in *18th USENIX Conference on File and Storage Technologies (FAST)*, 2020.

[31] Z. Li and *et al.*, "Amoeba: QoS-Awareness and Reduced Resource Usage of Microservices with Serverless Computing," in *34th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020.

[32] T. Rausch, A. Rashed, and S. Dustdar, "Optimized Container Scheduling for Data-intensive Serverless Edge Computing," *Future Generation Computer Systems*, vol. 114, pp. 259–271, 2021.