ACTS: Autonomous Cost-Efficient Task Orchestration for Serverless Analytics

Jananie Jarachanthan[†*], Li Chen[*], Fei Xu[‡]
[†]*University of Jaffna*, [*]*University of Louisiana at Lafayette*, [‡]*East China Normal University*

***Abstract*—Serverless computing has become increasingly popular for cloud applications, due to its compelling properties of high-level abstractions, lightweight runtime, high elasticity and pay-per-use billing. In this revolutionary computing paradigm shift, challenges arise when adapting data analytics applications to the serverless environment, due to the lack of support for efficient state sharing, which attract ever-growing research attention. In this paper, we aim to exploit the advantages of task-level orchestration and fine-grained resource provisioning for data analytics on serverless platforms, with the hope of fulfilling the promise of serverless deployment to the maximum extent. To this end, we present *ACTS*, an autonomous cost-efficient task orchestration framework for serverless analytics. *ACTS* judiciously schedules and coordinates function tasks to mitigate cold-start latency and state sharing overhead. In addition, ACTS explores the optimization space of fine-grained workload distribution and function resource configuration for cost efficiency. We have deployed and implemented *ACTS* on AWS Lambda, evaluated with various data analytics workloads. Results from extensive experiments demonstrate that *ACTS* achieves up to 98% monetary cost reduction while maintaining superior job completion time performance, in comparison with the state-of-the-art baselines.**

***Index Terms*—serverless computing, cost-efficiency, data analytics, cloud resource provisioning**

I. Introduction

Serverless architectures facilitate Function-as-a-Service (FaaS) in cloud computing, bringing salient features of lightweight runtime, ease of management, high elasticity, and fine-grained billing for cloud users. With the wide adoption of and increasing support for serverless platforms by cloud providers, such as Amazon Lambda [1], Google Cloud Functions [2], and Microsoft Azure Functions [3], serverless computing becomes a promising paradigm that will revolutionize cloud programming [4]. The benefits of serverless computing have been driving the paradigm shift for a wide variety of application workloads, including real-time video encoding [5], machine learning [6] and data analytics [7]. By design, serverless computing is suitable for an application that is easily decomposed to short-lived stateless functions. However, data analytics jobs, such as MapReduce jobs, typically consist of sequential execution stages, decomposable to functions that are parallel within an individual stage but stateful across consecutive stages. As such, it remains challenging to fulfil the promise out of the serverless shift for data analytics applications. More specifically, the challenges arise from the following coupled consideration: *lowering the overhead of cross-stage state sharing, requesting function resources wisely, and scheduling functions for performance-oriented or cost-oriented objectives*.

To address these challenges, a number of ephemeral state sharing solutions [7], [8] [9] have been proposed for serverless analytics, which rely on far-memory [10]systems for high-throughput and low-latency intermediate data transfer across function tasks. These solutions, however, sacrifice cost efficiency for application performance, incurring extra cost to build the far-memory systems or subscribe to faster storage services. On the other hand, several existing efforts focus on the orthogonal perspectives of function scheduling [11] [12] or fine-grained resource configurations [13] for job completion time and cost efficiency objectives.

In this paper, we propose our solution based on a comprehensive consideration of all the challenging issues stated above. In particular, we leverage the benefit of REST API for function-to-function data transfer, pipeline function tasks with the exploit of warm lambdas, and allocate function resources judiciously, to offer an optimal end-to-end execution plan for serverless analytics. We articulate the benefits to motivate our design with an example of a MapReduce job to be executed in the serverless environment. Vanilla analytics frameworks usually launch tasks of a stage only when all the tasks of the previous stage have finished, which is called a *lazy* approach. When deploying the MapReduce job on a serverless platform, the *lazy* schedule would not invoke the reducer functions until all the mappers outputs are ready, *i.e.*, written into the storage such as Amazon S3 [14] as the naive design. The execution timeline of the example job with three mappers (of different size) and one reducer is shown in Fig. 1a.

Now if we keep a mapper lambda alive and reuse it for the reducing task, the invocation and initialization time of the reducer can be saved, as depicted in Fig. 1b. In addition to mitigating the negative effect of function cold start [15], the approach in Fig. 1b also reduces the storage read/write overhead due to the data locality at `Map3`, in comparison with Fig. 1a. We can further eliminate the storage read/write of intermediate data by facilitating direct data transfer across

Corresponding author: Li Chen (li.chen@louisiana.edu). This work was supported in part by the BoRSF under Grants LEQSF(2019-22)-RD-A-21 and LEQSF(2021-22)-RD-D-07, in part by the NSF under Awards OIA-2019511 and CNS-1650551, in part by the NSFC under Grant 61972158, and in part by the Science and Technology Commission of Shanghai Municipality under Grants 20511102802 and 22DZ2229004.

979-8-3503-9973-8/23/$31.00 ©2023 IEEE

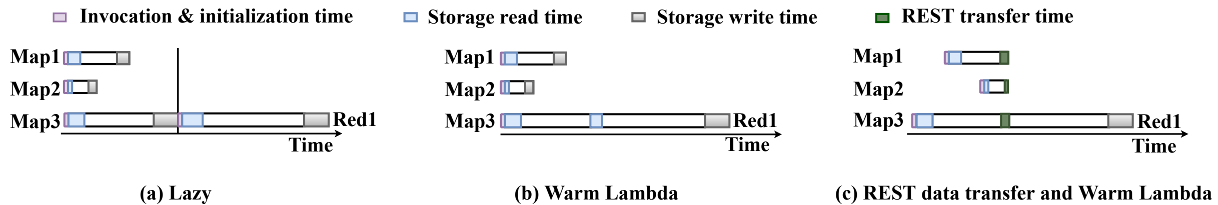


Fig. 1: Execution timeline of a MapReduce job, with three mappers (of different sizes) and a reducer, under different settings in the serverless environment: (a) Default *lazy* approach, state sharing through storage service, (b) Warm lambda for the reducer, state sharing through storage service, and (c) Warm lambda for the reducer, state sharing through REST API.

tasks through REST APIs. The challenge, however, is to schedule the execution of functions so that the functions invoking and invoked will not have to wait for each other for data transfer. For example, if `Map3` invokes `Map1` and `Map2` at the start of its execution and fetches data after its mapping finishes, similar as the execution order in Fig. 1b, `Map1` and `Map2` will be largely prolonged until the reducer finishes data reading, which increases the monetary cost. Thus, a better scheduling is desired as in Fig. 1c, to fully utilize the advantages of direct transfer and optimize the cost efficiency.

Another challenge, as well as an opportunity, is the fine-grained workload distribution and resource configuration for multi-stage analytics jobs, subject to platform limitations on function resources and data transfer size. For cloud users, there is little general guidance on serverless provisioning among the large configuration space, including the per-stage parallelism, per-function invocation scheduling and resource allocation, *etc.* Existing frameworks (*e.g.*, Pywren [16] and Caerus [12]) for serverless analytics typically have coarse-grained task orchestration and resource provisioning, which may easily result in a suboptimal deployment, incurring extra billing costs for cloud users.

Having identified all the aforementioned potential and challenges, in this paper, we present ***ACTS***, an ***A***utonomous ***C***ost-efficient ***T***ask orchestration framework for ***S***erverless analytics. To mitigate the adverse effect of function cold start and reduce the overhead of function state sharing (*i.e.*, intermediate data transfer across functions), *ACTS* schedules and coordinates function tasks in a mode similar to fork-join. In particular, in each computation stage, a group of functions will invoke other functions through REST following the scheduling decisions and receive their states upon their completion. Leveraging the benefits of warm containers, this group of functions will continue for the next stage processing, avoiding cold-start latency and state sharing overhead if new functions were invoked. The scheduling of function invocations will be judiciously determined in *ACTS* to avoid function idle time waiting for data transfer. *ACTS* further explores the space of fine-grained workload distribution and function resource configuration for cost efficiency. Based on a cost model, *ACTS* solves an optimization problem to minimize the monetary cost of a serverless data analytics job, without violating the platform limitations such as the function temporary storage size and the data transfer size.

Finally, we have implemented *ACTS* and deployed it on AWS Lambda for extensive performance evaluation. Real-world experiments have been conducted with various data analytics workloads at different scales, including three types of queries over Uservisits and Rankings datasets, sort, and two different machine learning workloads. Results demonstrate the effectiveness of *ACTS* in automatically configuring and orchestrating lambda functions for data analytics towards monetary cost minimization. Compared with the state-of-the-art baselines, *ACTS* consistently exhibits its advantages of cost saving, as well as performance improvement: it achieves cost reduction up to 98% for the queries workloads, at least 49% for the machine learning jobs, and up to 45% for the sort job, while maintaining superior job completion time performance with 6% to 62% improvement.

The rest of the paper is organized as follows. Sec. II presents the background of serverless analytics and motivates our design of *ACTS* with an intuitive example. Sec. III presents our proposed approach on task-level orchestration in detail. Sec. IV implements *ACTS* and demonstrates its advantages over the baselines with real-world experiments. Sec. V discusses the related work, and Sec. VI presents concluding remarks.

II. Background and Motivation

In this section, we present a brief background of serverless computing, particularly focused on data analytics. We then identify the limitations in current serverless analytics and motivate our design with an intuitive example.

A. Serverless Data Analytics

Serverless computing has emerged as a promising paradigm with increasing popularity because of its ability to simplify the code deployment with one-click upload and lightweight execution. With serverless computing, users no longer need to handle the complexities of cluster provisioning and management. The fine-grained pay-per-use billing model and high-elasticity in serverless computing also benefit a user in terms of cost efficiency, since the user does not have to pay for idle virtual machines during workload dynamics as in traditional cloud computing. Due to these compelling features, the past several years have witnessed the computing paradigm shift among a wide array of real-world applications ([5]–[7], [17]–[19] *etc.*), including data analytics (*e.g.*, [7]).

However, such a paradigm shift is nontrivial for data analytics applications, which typically proceed in interdependent computation stages. Although parallel tasks within a single

stage can be directly mapped to stateless serverless functions, the intermediate data transfer across consecutive stages poses challenges in the serverless environment, as direct communications between serverless functions are not supported by cloud providers [4]. Existing works thus turn to commercial storage or caching services (such as Amazon S3 [14], ElastiCache [20]) or build dedicated far-memory systems (Pocket [8], Jiffy [9], *etc.*) for state sharing at extra expense. The choice of data transfer mechanism would impact the end-to-end job completion time, as well as the total monetary cost. Another performance factor with serverless architecture is the function cold-start latency, which is the time taken for allocating an ephemeral container and initializing function modules when the function is launched for the first time. Thus, leveraging warm containers when planning task executions offers an intuitive optimization perspective for job completion time performance. Finally, cloud users have more flexibility for their jobs on serverless platforms, as resources are allocated and charged at function-level granularity. With such flexibility, the promise is that a user could plan its end-to-end job execution, by launching functions at just the right time and allocating function resources with just the right amount, to optimize his cost-efficiency. However, there lacks general guidance on fine-grained function scheduling and resource configuration, making it hard for cloud users, especially nonexperts in distributed computing or programming, to navigate the large design space with coupled decision variables. Therefore, in this paper, we are motivated to tackle these challenges in serverless analytics, to fulfil its promise on cost efficiency to an maximum extent.

B. Motivation Experiment

We consider an example MapReduce job for Wordcount to motivate our desgin for serverless data analytics. As a sample execution setting, without loss of generality, eight mappers are launched concurrently, followed by three reducing stages, as shown in Fig. 2. More specifically, reducer `s1r1` in the first reducing stage reads and handles the outputs from mappers `m1`, `m2`, and `m3`. Similarly, reducers `s1r2` and `s1r3` process the outputs of `m4-m6` and `m7-m8`, respectively. In the next reducing stage, reducer `s2r1` is executed to handle the outputs from reducers `s1r1` and `s1r2` in the previous stage. The final reducer `s3r1` processes the intermediate data from `s2r1` and `s1r3`. All the intermediate data are communicated through standard storage service. The memory size for each lambda function is 256 MB. Fig. 2 presents the execution timeline of the MapReduce job on AWS Lambda, under different schedule of function tasks, respectively. The block length for each function task represents the function running time (or the task execution/completion time), including the storage reading and writing time.

In Fig. 2a, tasks are scheduled with the *Lazy* approach: a task will not be started until all its upstream tasks, *i.e.*, the tasks with the outputs it needs to read, have finished. As illustrated, `s1r1` is scheduled when the slowest mapper `m1`, among all the upstream tasks `m1-m3`, finishes. Following such a scheduling,

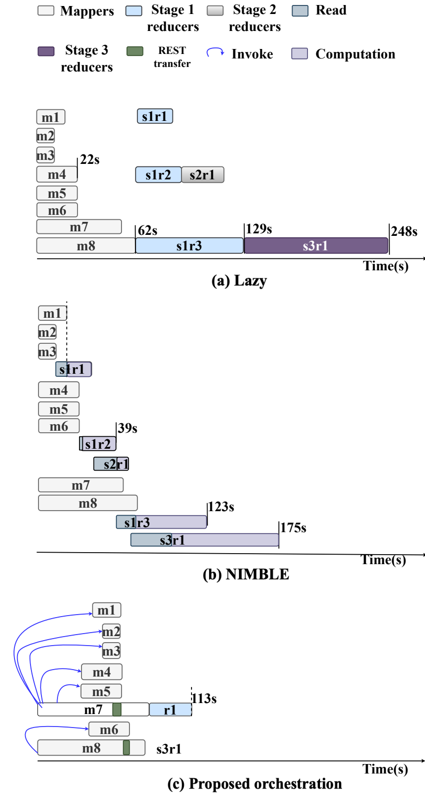


Fig. 2: Execution timeline of a MapReduce job for Wordcount under three different scheduling (and different state sharing) of function tasks in the serverless environment: (a) *lazy* scheduling as the default, (b) *NIMBLE* [12] scheduling as the state-of-the-art, (c) *ACTS* scheduling as our approach.

the job completes when the last task `s3r1` finishes, yielding a job completion time of 248 seconds. Fig. 2b presents the timeline when tasks are scheduled with *NIMBLE* [12], the state-of-the-art task-level scheduling algorithm for serverless analytics. The general idea of *NIMBLE* is to optimally overlap the downstream reading with upstream computation, so that the non-pipelineable computation in the downstream task can start as soon as possible for fastest job completion while the pipelineable data reading takes the shortest possible time for minimum monetary cost. As shown in Fig. 2b, each reducer task can be split into read and computation phases, which are pipelineable and non-pipelineable, respectively. For reducer `s1r1`, its computation phase starts almost immediately when the last mapper it relies on (`m1`) finishes, as the data reading phase is ideally scheduled and overlapped with the upstream mappers. Such a pipelining way of reading and computation continues until the final stage, resulting in a faster job completion time of 175 seconds compared with the lazy schedule.

Next, we present our proposed idea of task-level function orchestration, applied to the same Wordcount workload. As previously illustrated by Fig. 1, warm lambdas and REST data transfer will be leveraged in our framework named *ACTS*. Fig. 2c presents the execution of function tasks with *ACTS*, where mappers `m7` and `m8` invoke `m1-m5` and `m6`, respectively, at appropriate time instants through REST API. When the

Algorithm 1 *Orch_Heuristic*

Input: $\mathcal{H}$: The list of input object size, O, Q
Output: Orchestration j

1: Define $N \leftarrow 0$, boolean $stagecheck$=TRUE, Orchestration j
2: **while** $stagecheck$ **do** //Create stage s
3: $N \leftarrow N+1$, $warmcheck \leftarrow FALSE$
4: Define list $Lambdas \leftarrow \lambda_s^1, \lambda_s^2, ...\lambda_s^{P_s}$, list $output$
5: **for** $i \leftarrow Lambdas$ **do**
6: Create a list $warmset_{s,i}$
7: Allocate object size $f_{i,s}^j$ from $\mathcal{H}$ with λ_s^i
8: Append the binary label warm $W_{i,s}=1$ to λ_s^i
9: Append λ_s^i to $warmset_{s,i}$
10: Define $Size \leftarrow f_{i,s}^j + f_{i,s}^j/z$
11: Remove λ_s^i from $Lambdas$
12: **for** $l \leftarrow Lambdas$ **do**
13: $Size \leftarrow Size + f_{l,s}^j/z$
14: Allocate object size $f_{l,s}^j$ from $\mathcal{H}$ with λ_s^l
15: **if** ($Size >= O$ or $f_{l,s}^j/z >= Q$) **then** Break
16: **else**
17: Append the binary label warm $W_{i,s}=0$ to λ_s^l
18: Append λ_s^l to $warmset_{s,i}$
19: warmcheck=TRUE
20: Remove λ_s^l from $Lambdas$
21: Append output size $(Size - f_{i,s}^j)/z$ to $output$
22: Append $warmset_{s,i}$ to Orchestration j for stage s
23: $\mathcal{H} \leftarrow output$
24: **if** length of $\mathcal{H}$==1 **then** $stagecheck$=FALSE
25: **if** warmcheck==FALSE **then** Set Orchestration [j] as NULL
26: Break
27: Return Orchestration j

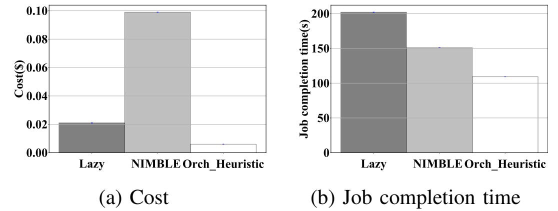
functions m1-m5 finish computing, they will transfer their data back to the function m7 which invokes them. Upon receiving all the data through REST, function m7 will be kept alive and proceed to the next stage processing, which is the reducing. The same applies to function m8. All the functions have the same memory allocation as in the previous settings. Compared with the previous two approaches in Fig. 2, our approach *ACTS*, with resource configurations yet to be further optimized, manages to reduce the job completion time by about 35% over the state-of-the-art *NIMBLE* scheduling, as evidence by the completion time in Fig. 2c. Moreover, with respect to the monetary cost for executing the same analytics, *ACTS* achieves at least 63% cost saving compared to *lazy* and *NIMBLE* scheduling. Such an improvement is due to multi-dimensional factors, including better execution plan, workload distribution, and the advantages of warm functions and REST data transfer.

III. Design of *ACTS* for Cost-efficient Serverless Analytics

Based on the observations from our motivation experiment in Section 2, we present the detailed design of our serverless analytics framework in what follows.

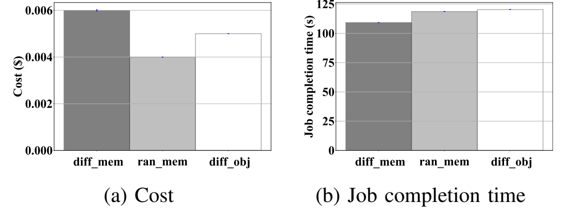
A. Task Orchestration

Our two-phase job modeling consists of mapper and reducer phases. The reducer phase has more than one stage. We consider a particular orchestration j. Each stage s can have a total of $P_s (s \in \{1, 2, , N\})$ number of lambdas. The warm lambdas in each stage are labeled as $W_{i,s}=1$. In each stage, a lambda processes the input data of size $f_{i,s}^j$. Each lambda will



(a) Cost (b) Job completion time

Fig. 3: Job completion times and monetary costs of three different settings: *Lazy*, *NIMBLE*, and *Orch_Heuristic*.



(a) Cost (b) Job completion time

Fig. 4: Job completion times and monetary costs of three different resource configuration settings.

output the objects with smaller size as $f_{i,s}^j/z$, where z is the assumed deviation of lambda processed output. For example, in the first stage, the first lambda will read the objects with a total size of $f_{1,1}^j$. We consider this lambda λ_1^1 as a warm lambda. Consider that the next stage's first lambda reads λ_1^1's output. So, λ_1^2's input is $f_{1,2}^j = (f_{1,1}^j/z + f_{2,1}^j/z + f_{3,1}^j/z)/z$. Here $f_{2,1}^j$ and $f_{3,1}^j$ are the inputs of the first stage lambdas that are attached to the warm lambda λ_1^1. The orchestration j can have flexible resource allocation and provisioning, including the number of stages, the number of lambdas in each stage, any number of warm lambdas in each stage, and any lambdas that can be attached to a warm lambda $W_{i,s}=0$.

We propose a heuristic called *Orch_Heuristic* to distribute computation workloads to the lambdas, select and label warm lambdas, and append a set of lambdas to the warm lambdas. Our Algorithm 1 develops an orchestration using the heuristic considering temporary storage limitation O and REST data transfer limitation Q with a serverless platform. Algorithm 1 creates a particular orchestration for a given set of object sizes $\mathcal{H}$. The loop (`line 5`) runs the stages to create an orchestration j until the final output of that stage is empty. In each stage, the algorithm labels warm lambdas as $W_{i,s}=1$, attaches objects sizes, and creates a set of lambdas ($W_{i,s}=0$) called $warmset_{s,i}$ for each warm lambda (`lines 7-9`). The algorithm checks the need for temporary storage and data transfer size of each warm lambda when appending the other lambdas to $warmset_{s,i}$ over the limitations O and Q (`lines 15-20`). For each stage, when each set of lambdas $W_{i,s}=0$ or 1 is ready, the $warmset_{s,i}$ is appended to the jth orchestration (`lines 22`). At the end of each stage, the outputs are stored in the list $\mathcal{H}$. If $\mathcal{H}$ has only one output then it will be the final output (`line 24`). Otherwise it will go to the next stage and continue the loop. We check whether there are lambdas $W_{i,s}=0$ attached to warm lambdas to avoid creating more stages like other existing approaches with fixed orchestrations (`line 25`). Finally, Algorithm 1 returns the orchestration j.

We implement Algorithm 1 in Python, execute it for a 10 GB two-phase (MapReduce) Wordcount job, and find the

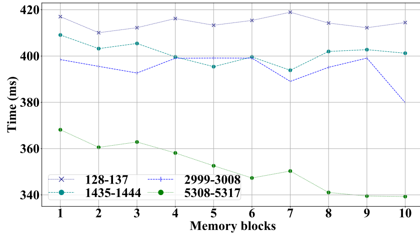


Fig. 5: Processing time of a lambda function for the same workload when allocated with different memory blocks.

orchestration for one input object size per lambda. Then we run the orchestration on the AWS Lambda platform. We execute the same job using *lazy* and *NIMBLE* approaches. All three settings are allocated the same randomly selected memory of 1536 MB. We present the results of job completion time and total cost in Fig. 3 to evaluate our algorithm against the other two. Fig. 3b shows 28% performance improvement for *Orch_Heuristic* compared to the *NIMBLE* approach and Fig. 3a shows at most 71% cost reduction for *Orch_Heuristic* over other settings. The *NIMBLE* approach uses two EC2 instances whereas *lazy* uses S3 for the data transfer. *Orch_Heuristic* gives 65% cost reduction compared to the *NIMBLE* approach without incurring the EC2 instance cost. The reason behind the results of *Orch_Heuristic* is that it carefully packs the objects into warm lambdas and runs in two stages, while other approaches complete the job through four stages. Also, Algorithm 1 reduces the lambdas' initialization times by aggregating tasks in warm lambdas, and reduces the cost as well as time by REST data transferring.

B. Resource Configuration

In AWS Lambda, we can allocate the memory for a lambda from 128 MB to 10,240 MB in unit MB increments. The CPU capacity of a lambda function is proportional to the allocated function memory. We elaborate on the benefits of appropriate resource allocations and provisioning by executing a 10 GB Wordcount job on AWS Lambda in three different settings. Setting 1 (called *ran_mem*) and setting 2 (called *diff_mem*) follow Algorithm 1 for one object per lambda. *ran_mem* is allocated with 1536 MB for all stages, whereas different randomly selected memories for each stage (512 MB, 1024 MB) are allocated for *diff_mem*. The setting 3 *diff_obj* allocates two objects per lambda and different memories for different stages (1024 MB, 2048 MB). Fig. 4a and Fig. 4b show the monetary costs and job completion times, respectively, for the three settings. There is 8% completion time reduction in *ran_mem* over *diff_mem*. Since *diff_obj* allocates more objects than other settings, the number of lambda invocations is smaller. However, *diff_mem* shows at least 20% cost reduction over the other two settings. Thus, a number of factors, including memory allocations, the number of lambdas, and object size allocations (*i.e.*, workload distributions), collectively impact the job performance and cost on serverless platforms. This sample experiment shows that our heuristic (Algorithm 1) can be coupled with variables of different dimensions to generate different orchestrations. In the upcoming subsections, we derive a cost model and formulate a cost optimization problem to navigate the design space of multi-dimensional variables such as memory allocation and function scheduling.

C. Cost Modeling

We define a binary variable $y_j(j \in 1, 2, , n)$ for each orchestration j, where n denotes the length of a complete set of possible orchestrations for a MapReduce job. In an orchestration j (generated by Algorithm 1), the total number of possible stages, including mapper and all reducer stages, is N. Each stage s can have a total of $P_s(s \in \{1, 2, , N\})$ lambdas. Each lambda $i(i \in P_s)$ in stage s appends the details such as whether it is warm or not (warm lambda is represented with $W_{i,s} = 1$), the lambdas that belong to a warm lambda ($l \in q_i$: q_i represents the set of lambdas which belong to the warm lambda i), and the set of objects' sizes f_i^j. We use the words step and stage interchangeably and will add explanation where used differently.

As mentioned in the previous subsection, in AWS Lambda, we can allocate the memory for a lambda within the 128 MB to 10,240 MB range. The CPU allocation of a lambda function is roughly proportional to the function memory range. AWS Lambda can allocate 2 to 6 vCPUs to a function. For example, when configuring memory within the 128 - 3008 MB range, a maximum of 2vCPUs can be allocated by the provider. Similarly, different ranges of memory blocks will lead to a different allocation of vCPUs. Fig. 5 presents the lambda processing time of a 1MB job (deployed on a single lambda) given four different ranges of memory allocations. All the four ranges show relatively small variations in lambda processing time within the ranges. The three ranges 128-137 MB, 1435-1444 MB, and 2999-3008 MB are associated with the CPU allocation of 2 vCPUs while the 5308-5317 MB setting has 4 vCPUs. Hence, the gap of the processing times between the memory allocations of different CPU capacities is larger than between the settings of memory blocks associated with the same CPU allocation. The cost of a lambda function largely relies on its runtime and the price of its compute resource allocation (memory size). Based on the experimental observation in Fig. 5 and for problem tractability, we consider a total of L types of memory allocations, which evenly spread out the allowable range.

We use binary variable $x^s_{j,k}, (k = 1, 2, \cdots, L)$ to specify whether the k-th type of memory is allocated out of the L categories for lambdas of stage s. Intuitively, we have

$$x^s_{j,k} \in \{0, 1\}, \quad \forall k \in \{1, 2, \cdots, L\}; \qquad \sum_{k=1}^{L} x^s_{j,k} = 1, \quad (1)$$

which indicates that only one category of memory allocation can be assigned by nature. For an orchestration j from Algorithm 1 in AWS Lambda, REST data transfer is used between warm and other lambdas within an individual stage, whereas Amazon S3 is used between stages for intermediate data storage. Since Algorithm 1 compacts stages into a fewer number than that in the *lazy* approach, the need for S3 is very less or eliminated. The total cost of a particular lambda includes the S3 input read cost and its own processing cost. In addition, if it is a warm lambda, the total cost further

includes the processing cost of the outputs from other lambdas belonging to it and the S3 write cost. Otherwise, the total cost of a non-warm lambda includes the REST transfer cost. Data read request cost per second is denoted as D. The size of input read by the i th lambda is $(y_j f_i^j)$. The S3-Lambda bandwidth is denoted as B. The warm lambdas $(W_{i,s} = 1)$ write the output to the S3. Data write request cost per second is denoted as G. The warm lambda output is expressed as $F_{i,s}^j = [(f_{i,s}^j/z) + \sum_{l \in q_i}(f_l^j/z)]/z$. The lambdas $(W_{i,s} = 0)$ that belong to a warm lambda transfer their outputs to the warm lambda. Data transfer cost per MB size is denoted as E. We use $v_{j,k}$ and $u_{j,k}^s$ to represent the price and unit computation time of the particular type of lambda. Then we have the read, processing, write and transfer costs expressed as:

$$U_{read} = D(y_j f_{i,s}^j)/B \tag{2}$$

$$U_{process} = \sum_{k=1}^{L} x_{j,k}^s u_{j,k}^s [(y_j f_{i,s}^j) + W_{i,s}(y_j f_{i,s}^j)/z + W_{i,s} \sum_{l \in q_i}(y_j f_{l,s}^j)/z] \sum_{k \in \mathcal{L}} v_{j,k}^s x_{j,k}^s \tag{3}$$

$$U_{write} = (G/B) W_{i,s}(y_j F_{i,s}^j), U_{send} = E(y_j f_{l,s}^j)/z \tag{4}$$

The initialization time of a lambda is denoted as t_i, and the invoking cost is represented as I. Then the total cost of a lambda can be expressed as:

$$C_{i,s}^j = U_{read} + U_{process} + U_{write} + (1 - W_{i,s}) U_{send} + I + t_i \sum_{k \in \mathcal{L}} v_{j,k}^s x_{j,k}^s \tag{5}$$

D. Cost Minimization

The objective of our formulation is to minimize the monetary cost, which is $\sum_{j \in n} \sum_{s \in \mathcal{N}} \sum_{i \in \mathcal{P}_f} C_{i,s}^j$, over variables y_j and $x_{j,k}^s$, represented as follows.

$$\min_{\boldsymbol{x},\boldsymbol{y}} \quad F(x,y) = \sum_{j \in n} \sum_{s \in \mathcal{N}} \sum_{i \in P_s} C_{i,s}^j \tag{6}$$

$$\text{s.t.} \quad Eq.(1), y_j \in \{0,1\}, \quad \forall j \in \{1,2,\cdots,n\} \tag{7}$$

For a particular orchestration $j, y_j = 1$, the objective Eq. (6) can be written as $G(x, \bar{y})$. Now the formulation on variable x is:

$$\min_{\boldsymbol{x}} \quad G(\boldsymbol{x}, \bar{\boldsymbol{y}}) \tag{8}$$

$$\text{s.t.} \quad \text{Eq. (1)} \tag{9}$$

The formulation falls into the category of linearly-constrained zero-one quadratic program on $\boldsymbol{x}$, given any $\boldsymbol{y}$. The objective in Eq. (8) consists of constant items ($f_{i,s}^j, z, G, B,$ *etc.* from Eq. (6)). Replacing them with real numbers Q_k and P_k for a particular j, we can rearrange the formulation as:

$$\min_{\boldsymbol{x}} \quad \sum_{j=1}^{L} Q_j x_j x_j + \sum_{j=1}^{L} P_j x_j \tag{10}$$

$$\text{s.t.} \quad x_j \in \{0,1\}^L \tag{11}$$

To solve this problem, we build a quadratic convex reformulation using semidefinite relaxation [21]. As we do not have any equality constraint, replacing the product of $x_j x_j$ by a variable X_j yields:

$$\min \quad \sum_{j=1}^{L} Q_j X_j + \sum_{j=1}^{L} P_j x_j \tag{12}$$

$$\text{s.t.} \quad X_j = x_j x_j, x_j \in \{0,1\}^L \tag{13}$$

Using the semidefinite relaxation of the previous formulation, we can replace the constraints Eq. (15) and (16) with the linear matrix inequality $X = xx^t \geq 0$. From Schur's Lemma [22], the linear matrix is equivalent to $\begin{bmatrix} 1 & x^t \\ x & X \end{bmatrix} \geq 0$. Now the obtained form of SDP relaxation is:

$$\min \quad \sum_{j=1}^{L} Q_j X_j + \sum_{j=1}^{L} P_j x_j \tag{14}$$

$$\text{s.t.} \quad \begin{bmatrix} 1 & x^t \\ x & X \end{bmatrix} \geq 0, \quad x \in \mathbb{R}^L, X \in S^L \tag{15}$$

Here the S^L defines $L \times L$ symmetric matrix. Now we use the optimal solution to this SDP in order to build a quadratic reformulation. We introduce the QCR method [21] of reformulating the formulation by adding a combination of quadratic functions that can vanish on a feasible solution set X. For any $\mu \in \mathbb{R}^L$

$$G_\mu(x, \bar{y}) = \sum_{j=1}^{L} Q_j x_j x_j + \sum_{j=1}^{L} P_j x_j + \sum_{j=1}^{L} \mu_j (x_j^2 - x_j) \tag{16}$$

The function $G_\mu(x,y)$ is a reformulation since for all $x \in X$, $G_\mu(x,y)$ is equal to $G(x,y)$. And we have to find the μ such that $G_\mu(x,y)$ is convex. So, from the semidefinite relaxation, $G(x,y)$ is transformed into convex. We can solve the reformulated problem (16) using mixed-integer convex quadratic programming. It has already been proved in [21] that solving the above semidefinite relaxation SDP allows us to deduce optimal values for μ. The optimal value μ_j^* of $\mu_j; j \in \{1,2,\cdots,L\}$ will be given by the optimal values of the dual variables associated with constraint (20). The resulting quadratic convex reformulation is:

$$RQ_{conv} : \text{Min} \quad G_{\mu^*}(x,y) \tag{17}$$

$$\text{s.t.} \quad x_j \in \{0,1\}^L \tag{18}$$

The formulation Eq. (7)-(9) can be rewritten as,

$$\min_{\boldsymbol{x} \in \mathbb{R}^L, \boldsymbol{y} \in \mathbb{R}^n} \phi(X,y) : \sum_{k=1}^{L} \sum_{j=1}^{n} a_{k,j} X + \sum_{j=1}^{n} b_j y_j + \sum_{k=1}^{L} d_k x_k \tag{19}$$

$$\begin{bmatrix} 1 & x^t \\ x & X \end{bmatrix} \geq 0, Eq.\ (1) and (7) \quad x \in \mathbb{R}^L, X \in S^L \tag{20}$$

Here real number $a_{k,j}$, b_j, and d_k consists of constants ($f_{i,s}^j, z, G, B,$ *etc.*) from Eq. (6). We denote that ϕ_{min} and ρ are the optimal values of (19) and (8) for a particular y, respectively. We've shown that Eq. (6) formulation can be relaxed on any given y and it is clear that $\phi_{min} <= \rho$. So, the global minimum of y will give the optimal value for (6). The formulation (19) confirms that it is a binary quadratic problem. Hence any MIQP and BQP [23] [24] solvers can be used to solve this problem.

Algorithm 2 *ACTS*

Input: M: Number of objects, $\mathcal{H}$: The set of input objects sizes
Output: Orchestration and configuration
1: **for** $j \leftarrow [1,5]$ **do**
2: Create object size list $\mathcal{H}$ by summing sizes of j objects
3: Execute Algorithm 1 for input objects $\mathcal{H}$
4: **if** Orchestration [j]==NULL **then** Break
5: Define binary variables y and x
6: Calculate cost as in Eq. (6)
7: Define formulation as in Eq. (7)-(9)
8: (x,y) ←Execute the optimizer
9: Define S_MR from (x,y)
10: **for** $s \leftarrow 0, N$ **do** //For the S_MR
11: **for** $i \leftarrow 0, P_s$ **do**
12: Calculate launch time $\leftarrow (T_w - T_l)$, Update S_MR.
13: Execute S_MR on AWS Lambda

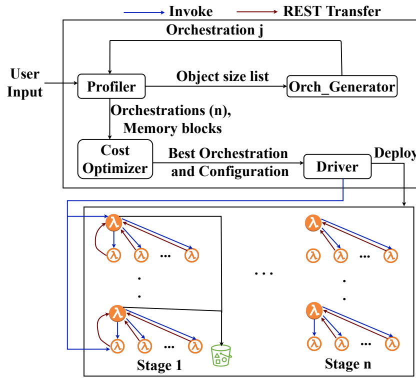


Fig. 6: Architecture overview of *ACTS* for serverless analytics with autonomous cost-efficient task orchestration.

E. Design of ACTS

We next develop Algorithm 2 to find the cost-minimal orchestration with the best resource configuration for a two-phase multi-stage job. First, it creates an input object sizes list. For example, if the number of objects per lambda is 2 ($j = 2$ in `line 1`), then the sum of each two objects will be inserted in the list $\mathcal{H}$. We assumed the number of repetitions as 5 for the loop to eliminate big object size summations ($\mathcal{H}$) and unnecessary time for limitations checking of Algorithm 1. As in `line 3`, it executes Algorithm 1 and finds the best orchestration for j, and if there is no set of lambdas attached to the warm, Algorithm 2 will break and re-enter the loop (`line 4`). If there is no orchestrations listed, because of limitations, then the Lazy approach solution will be followed with memory configurations [13]. Algorithm 2 creates the formulation in Eq. (7)-(9), executes the optimizer, and finds the best orchestration (S_MR) with appropriate resource provisioning plan (`line [7-9]`). For this best orchestration, it calculates the launch times of each lambda (`line [11-13]`), to be elaborated soon. Finally, *ACTS* deploys the orchestration with appropriate memory configurations using found resource provisioning plan on AWS Lambda.

Lambda Function Scheduling. In each stage s, warm lambdas are launched concurrently at the initiation of that stage. Other lambdas that belong to a particular warm lambda are invoked by the warm lambda. As shown in Fig. 7, the launch time of a non-warm lambda is $T_w - T_l$. Here T_w is

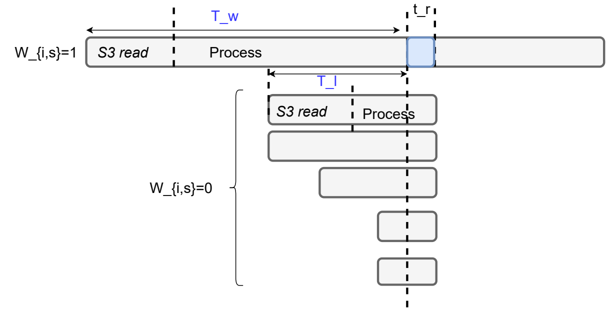


Fig. 7: Function launch time.

the summation of S3 read and processing time of the warm lambda's own input. T_l is the summation of S3 read and processing time of the lambda attached to the warm lambda. The following equations substitute lambda i with $warm$ for warm lambda and $nwarm$ for the other attached lambdas.

$$T_w = (y_j f^j_{warm,s})/B + \sum_{k=1}^{L} x^s_{j,k} u^s_{j,k} y_j f^j_{warm,s} \tag{21}$$

$$T_l = (y_j f^j_{nwarm,s})/B + \sum_{k=1}^{L} x^s_{j,k} u^s_{j,k} y_j f^j_{nwarm,s} \tag{22}$$

Now, we are ready to integrate each component into our automated prototype *ACTS* for the fine-grained task-level orchestration. As illustrated in Fig. 6, given the user input, the `Profiler` creates multiple orchestrations as in Algorithm 2 (`line[1-5]`) using the `Orch_Generator` component and passes the orchestrations and the selected memory blocks (as explained in Section 3.2) to the `Cost Optimizer`. Then the `Cost Optimizer` runs the solver and results in the best orchestration with resource provisioning plan and configurations as in Algorithm 2 (`line[6-10]`). Finally, the `Driver` deploys and executes the analytics job on the serverless platform accordingly.

IV. Performance Evaluation

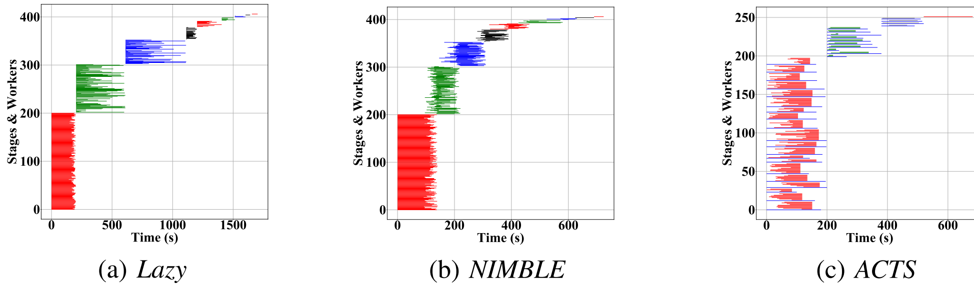
In this section, we present the implementation of *ACTS*, and evaluate its performance with real-world experiments.

A. Prototype Implementation and Experimental Setup

ACTS prototype is implemented to run on the AWS Lambda platform. The algorithm explained in the previous section is implemented in Python to run on the client machine. It includes the profiler and optimizer. After orchestration is ready, the prototype is deployed as Mapper and Reducer task zip files in the AWS Lambda platform, and input is uploaded to the AWS S3. *ACTS* utilizes REST API to handle the intermediate data transfer. We have implemented different baseline settings to compare with our prototype.

Baseline *Lazy:* The MapReduce style prototypes such as PyWren [16] and Locus [7] follow the *Lazy* scheduling as explained in previous sections. Our baseline has the multi-stage MapReduce style, and functions are allocated 3008 MB of memory. It consists of a number steps with 1 object per mapper and 2 objects per reducer. Here AWS S3 is used to store the input and intermediate outputs.

Baseline *NIMBLE:* *NIMBLE* baseline is implemented by following the state-of-the-art Caerus. Since it does not consider any technique to allocate memory or objects per lambda,



(a) *Lazy* (b) *NIMBLE* (c) *ACTS*

Fig. 8: Job completion time breakdown of Aggregation query workload under *Lazy*, *NIMBLE*, and *ACTS* settings.

we follow the same configuration as our baseline *Lazy*. We employed two m4.4xlarge instances to handle the intermediate outputs and AWS S3 to store input data.

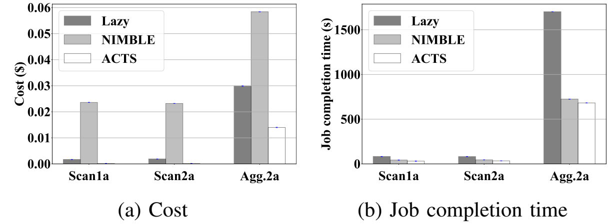
In our evaluations, we illustrate the performance and monetary cost with error bars of standard deviation by running each of the experiments three times. In the following subsections, we present the results and analysis of three groups of experiments. We compare our *ACTS* with the two baselines *Lazy* and *NIMBLE* for various data analytics workloads, including three different BigData Benchmark queries [25], two machine learning workloads, and MapReduce sort (100GB).

B. Results and Analysis

Big Data Benchmark. In our first set of evaluations, we run the three Big Data Benchmark queries in three different settings: *Lazy*, *NIMBLE* and *ACTS*. Here, the selection queries (Scan1a, Scan2a) of the PageRank algorithm are executed over the Rankings dataset, with 90 million rows and a size of 6.38 GB. More specifically, Scan1a and Scan2a are selection queries that correspond to two different conditions of pageRank>1000 and pageRank>100, respectively. Aggregation query, referred to as Aggregation2a, is evaluated on the Uservisits dataset with 155 million rows and 25.4 GB size.

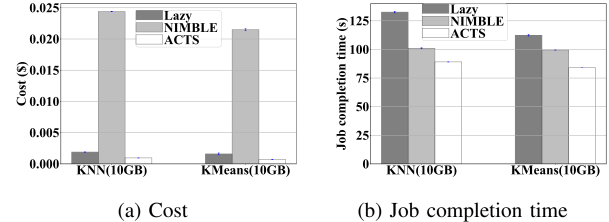
Fig. 9a presents the total monetary cost of our prototype, in comparison with the two baselines, over the three different queries stated above. Our *ACTS* shows 91%, 93%, and 53% cost reductions over *Lazy* approach. The task orchestration with *ACTS* results in 3 stages for two Scan queries and 4 stages to finish the Aggregation query, whereas the *Lazy* setting has 5 stages to finish the selection queries and 9 stages for the aggregation query. Leveraging warm lambdas and REST data transfer together lead to the reduction of intermediate storage cost and function invocation cost in *ACTS*, as compared to *Lazy*. Also, the coarse-grained sub-optimal memory allocations with *Lazy* is avoided by fine-grained appropriate configurations in *ACTS*, contributing to the reduction of the total monetary cost. Compared with the *NIMBLE* baseline, *ACTS* shows nearly 98%, 98%, and 82% reductions in total costs. If we do not account for the cost incurred by the EC2 instances used in *NIMBLE* for intermediate data sharing, *ACTS* still exhibits nearly 60%, 63%, and 45% cost savings.

While *ACTS* is demonstrated to optimize the monetary cost, it also exhibits advantages with respect to job completion time performance, achieving better or at least similar performance



(a) Cost (b) Job completion time

Fig. 9: Monetary costs and job completion times of different queries workloads under *Lazy*, *NIMBLE*, and *ACTS* settings.



(a) Cost (b) Job completion time

Fig. 10: Monetary costs and job completion times of machine learning workloads (KNN and KMeans) under *Lazy*, *NIMBLE*, and *ACTS* settings.

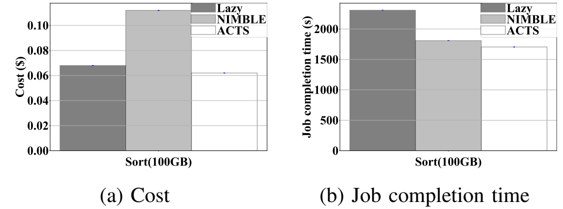
level compared to the baseline approaches. As evidenced by Fig. 9b, *ACTS* shows 62%, 58%, and 58% performance improvement compared to *Lazy*. Also, *ACTS* shows 28%, 25%, and 6% performance improvement over *NIMBLE*. The reasons for the performance improvement include the usage of warm lambdas (task aggregation), REST data transfer, appropriate stage level memory allocations, judicious association of lambdas with each warm lambda, and reduced number of stages, which contribute to the cost minimization as well. In a more intuitive way, Fig. 8 presents the stage-level timeline breakdown for the Aggregation 2a query job, to more clearly illustrate the job execution details under different settings. The *Lazy* and *NIMBLE* prototypes in Fig. 8a and Fig. 8b execute 202 lambdas concurrently in stage 1 since they don't follow any heuristic to check objects sizes and distribute workloads. In contrast, the heuristic with *ACTS* presented in Algorithm 1 carefully packs the input objects in stage 1 by considering the platform limitations like temporary storage and data transfer (for REST) sizes. Also, *ACTS* uses S3 as intermediate storage within stages for only three times. As shown in Fig. 8c, all the warm lambdas start the execution concurrently whereas others are launched at different times calculated by Algorithm 2. The *Lazy* approach starts all the lambdas concurrently in all stages when the outputs are ready for intake in each other stage. *NIMBLE*'s first stage lambdas execute concurrently and

continue launching the next stage when a part of the output is ready. Due to the appropriate workload assignment across lambdas through Algorithm 2, the total number of lambda invocations is fewer in *ACTS* compared to the other two baselines. These reasons behind the performance improvement of *ACTS* contribute to the cost saving as well. For example, fewer lambda launches, less usage of storage services, and fine-grained appropriate memory allocations also reduce the monetary cost of *ACTS* compared to the baselines.

Machine Learning Analytics. We further evaluate the monetary cost of *ACTS* in comparison with the baselines *Lazy* and *NIMBLE* for two machine learning workloads elaborated as follows. The first evaluation involves the K-nearest neighbors algorithm for classification problems. 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 execute the K-nearest neighbors in *ACTS*, *Lazy* , and *NIMBLE* prototypes 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. Fig. 10a and Fig. 10b present the evaluation results in terms of monetary cost and job completion time for two types of jobs, including the K-nearest neighbors job referred to as KNN. As clearly observed, *ACTS* results in 49% cost reduction compared to *Lazy*, and 96% cost reduction over *NIMBLE*.

The second job is the K-means clustering, an unsupervised machine learning algorithm that discovers patterns by grouping similar data points into K clusters. *ACTS*, *Lazy*, and *NIMBLE* implement 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. As shown in Fig. 10a, *ACTS* achieves 56% and 96% cost reductions over *Lazy* and *NIMBLE* approaches, respectively. When eliminating the cost of EC2 instances for *NIMBLE*, *ACTS* still shows 60% cost reduction over it. While minimizing the cost, *ACTS* also manages to achieve 33% and 15% performance improvement over *Lazy* and NIMBLE, respectively, as illustrated in Fig. 10b. Therefore, *ACTS* works effectively and maintains its advantages over the baselines for the MapReduce convertible machine learning workloads.

MapReduce Sort. Finally, we evaluate our prototype with a 100 GB sorting job, still in comparison with the two baseline settings. Fig. 11a and Fig. 11b present the total monetary cost and completion time of the job achieved by *ACTS*, in comparison with *Lazy* and *NIMBLE*. Again, *ACTS* shows 45% and 9% cost reductions over *NIMBLE* and *Lazy* baselines, respectively, while achieving 6% and 26% performance improvements. In summary, *ACTS* consistently exhibits its advantages over the state-of-the-art baselines, with respect to both the monetary cost and job completion time performance.



(a) Cost (b) Job completion time

Fig. 11: Monetary costs and job completion times of Sort (100GB) workloads under *Lazy*, *NIMBLE*, and *ACTS* settings.

C. Discussion

The running time overhead of *ACTS* incurred by the BQP solver 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 overhead can be negligible (in milliseconds) on a more powerful commodity server. Though implemented in AWS Lambda, *ACTS* can be adapted to Google Functions and Azure Functions with minimal modifications in our model and problem formulation. Extension to other platforms will be left as our future work. Our current study is focused on a two-phase (MapReduce style) multi-stage serverless analytics job. Extending it to a more general data analytics job, with a complex DAG (Directed Acyclic Graph) of stages, is nontrivial and remains open, which will be left as our future direction.

V. Related Works

On serverless platforms, users can deploy and run their applications without worrying about the infrastructure complexities, charged at the function invocation granularity. To deploy data analytics applications in the serverless environment, challenges arise due to the need for data analytics tasks to exchange intermediate data which is not well supported by serverless providers. To address the challenges, existing works either subscribe to commercial storage or caching services provided in the cloud (such as Amazon S3 [14], ElastiCache [20]) or manually build far-memory systems (Pocket [8], Jiffy [9], *etc.*) that are dedicated and optimized for serverless state sharing at extra expense. For example, PyWren [16], Serverless architecture [26], MARLA [27], and Astra [13] utilize S3 for the intermediate data transfer for MapReduce or query style analytics jobs. Lambada [28] and Flint [29] leverage SQS and/or DynamoDB to store and retrieve the data. Locus [7] extends Pywren and augments shuffle in MapReduce using expensive fast ElastiCache (Redis) [20] instances combined with the much cheaper S3 service. Pocket [8] presents a distributed far-memory system for high-throughput and low-latency intermediate data transfer of serverless analytics. Jiffy [9] further enables fine-grained far-memory sharing and multiplexing across concurrent serverless analytics jobs. The data-centric approach [30] provides a data bucket abstraction to hold the intermediate data generated by functions. Different from these efforts, *ACTS* leverages the advantages of REST API for function invocations and data transfer, aiming at cost efficiency for serverless analytics jobs.

Apart from the existing state sharing or intermediate data transfer solutions for serverless data analytics aforementioned, recently, task level scheduling for serverless analytics has increasingly gained research attention. Wukong [11] adopts decentralized scheduling and task clustering in its serverless framework that can reduce data movement over the network and improve cost-effectiveness. Orion [31] provisions VMs, bundles functions, and pre-warms at appropriate timings for serverless DAG applications according to a distribution-based latency model which requires repeated dry-runs. The state-of-the-art *Caerus* [12] theoretically defines and studies the serverless scheduling problem for data analytics as a new problem. It presents *NIMBLE* scheduling to efficiently pipeline task executions within a data analytics job, minimizing execution cost while being Pareto-optimal between cost and job completion time for serverless analytics. Compared to these efforts, *ACTS* incorporates the consideration of task level scheduling, while targeting a more complex problem with multi-dimensional decisions to make. As demonstrated in our evaluation, task-level orchestration, coupled with warm containers, REST data transfer, fine-grained resource provisioning, can be more cost-efficient.

VI. Conclusion

We present *ACTS* for serverless data analytics in this paper, which is an autonomous framework that generates and enforces cost-efficient task orchestration and resource configuration for data analytics jobs. *ACTS* relies on function invocation and state sharing in a mode similar to fork-join, exploiting warm containers and REST data transfer to reduce the overhead of cold-start latency and function state sharing. In addition to the careful scheduling of function invocations, it further explores the fine-grained resource configuration and workload assignment space to generate and enforce cost-minimal decisions. We have implemented *ACTS* and deployed it on AWS Lambda for extensive real-world experiments. Evaluated with various data analytics workloads, *ACTS* consistently exhibits better performance with respect to both monetary cost saving and job completion time reduction, in comparison with the state-of-the-art baselines.

References

[1] (2022) AWS Lambda. [Online]. Available: https://aws.amazon.com/lambda/

[2] (2022) Cloud Functions. [Online]. Available: https://cloud.google.com/functions

[3] (2022) Azure Functions. [Online]. Available: https://azure.microsoft.com/en-us/services/functions/

[4] J. Schleier-Smith and *et al.*, "What Serverless Computing is and should Become: The Next Phase of Cloud Computing," *Communications of the ACM*, vol. 64, no. 5, pp. 76–84, 2021.

[5] 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.

[6] J. Carreira and *et al.*, "Cirrus: a Serverless Framework for End-to-end ML Workflows," in *Proceedings of ACM Symposium on Cloud Computing (SoCC'19)*, Nov. 2019, pp. 13–24.

[7] Q. Pu and *et al.*, "Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

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

[9] A. Khandelwal and Y. T. W. Serverless, "Jiffy: Elastic Far-memory for Stateful Serverless analytics," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 697–713.

[10] A. Lagar-Cavilla, J. Ahn, and *et al.*, "Software-defined Far Memory in Warehouse-scale Computers," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 317–330.

[11] B. Carver and *et al.*, "In Search of a Fast and Efficient Serverless DAG Engine," in *Fourth International Parallel Data Systems Workshop (PDSW)*, 2019.

[12] H. Zhang and *et al.*, "Caerus: NIMBLE Task Scheduling for Serverless Analytics," in *Proceedings of 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)*, Apr. 2021, pp. 653–669.

[13] J. Jarachanthan and *et al.*, "Astra:Autonomous Serverless Analytics with Cost-Efficiency and QoS-Awareness," in *35th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021.

[14] (2022) Amazon S3. [Online]. Available: https://aws.amazon.com/s3/

[15] P. Vahidinia and *et al.*, "Mitigating Cold Start Problem in Serverless Computing: A Reinforcement Learning Approach," *IEEE Internet of Things Journal*, 2022.

[16] E. Jonas and *et al.*, "Occupy the Cloud: Distributed Computing for the 99%," in *Proceedings of the 2017 Symposium on Cloud Computing*, 2017.

[17] M. Yu, Z. Jiang, and *et al.*, "Gillis: Serving Large Neural Networks in ServerlessFunctions with Automatic Model Partitioning," in *41st IEEE International Conference on Distributed Computing Systems*, 2021.

[18] F. Xu and *et al.*, "λDNN: Achieving Predictable Distributed DNN Training with Serverless Architectures," *IEEE Transactions on Computers*, 2021.

[19] J. Jarachanthan, L. Chen, and *et al.*, "AMPS-Inf: Automatic Model Partitioning for Serverless Inference with Cost Efficiency," in *50th International Conference on Parallel Processing (ICPP)*, Aug., pp. 1–12.

[20] (2022) Amazon ElastiCache. [Online]. Available: https://aws.amazon.com/elasticache/

[21] A. Billionnet, S. Elloumi, and M.-C. Plateau, "Quadratic 0–1 Programming: Tightening Linear or Quadratic Convex Reformulation by Use of Relaxations," vol. 42, no. 2. EDP Sciences, 2008, pp. 103–121.

[22] F. Zhang, *The Schur Complement and Its Applications*. Springer Science & Business Media, 2006, vol. 4.

[23] (2008) GUROBI. [Online]. Available: https://www.gurobi.com/

[24] C. Bliek1ú, P. Bonami, and A. Lodi, "Solving Mixed-Integer Quadratic Programming Problems with IBM-CPLEX: a Progress Report," in *Proceedings of the twenty-sixth RAMP symposium*, 2014, pp. 16–17.

[25] (2014) Big Data Benchmark. [Online]. Available: https://amplab.cs.berkeley.edu/benchmark/

[26] (2022) MapReduce. [Online]. Available: https://aws.amazon.com/blogs/compute/ad-hoc-big-data-processing-made-simple-with-serverless-mapreduce/

[27] 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.

[28] I. Müller and *et al.*, "Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure," in *ACM International Conference on Management of Data (SIGMOD)*, 2020.

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

[30] M. Yu and *et al.*, "Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.

[31] A. Mahgoub, E. B. Yi, and *et al.*, "Orion and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.