WiseFuse: Workload Characterization and DAG Transformation for Serverless Workflows

ASHRAF MAHGOUB, Purdue University, USA
EDGARDO BARSALLO YI, Purdue University, USA
KARTHICK SHANKAR, Carnegie Mellon University, USA
ESHAAN MINOCHA, Purdue University, USA
SAMEH ELNIKETY, Microsoft Research, USA
SAURABH BAGCHI, Purdue University, USA
SOMALI CHATERJI, Purdue University, USA

We characterize production workloads of serverless DAGs at a major cloud provider. Our analysis highlights two major factors that limit performance: (a) lack of efficient communication methods between the serverless functions in the DAG, and (b) stragglers when a DAG stage invokes a set of parallel functions that must complete before starting the next DAG stage. To address these limitations, we propose WiseFuse, an automated approach to generate an optimized execution plan for serverless DAGs for a user-specified latency objective (SLO) or cost budget. We introduce three optimizations: (1) **Fusion** combines in-series functions together in a single VM to reduce the communication overhead between cascaded functions. (2) **Bundling** executes a group of parallel invocations of a function in one VM to improve resource sharing among the parallel invocations to reduce skew. (3) **Resource Allocation** assigns the right size to each VM hosting a function or a group of functions to reduce the latency and cost of invoking the serverless DAG. We implement WiseFuse and evaluate it experimentally using three serverless applications, namely, Video Analytics, Approximate SVD, and ML Analytics, which span different DAG structures, memory footprints, and intermediate data sizes. In comparison to competing approaches, WiseFuse shows significant improvements in E2E latency and cost. Specifically, for the ML pipeline, WiseFuse achieves a P95 latency that is 67% lower than Photons [SoCC-20], 39% lower than Faastlane [USENIX ATC-21], and 90% lower than Sonic [USENIX ATC-21], without increasing the $ cost.

CCS Concepts: • **Computer systems organization → Cloud computing**.

Additional Key Words and Phrases: DAG transformation; serverless; workload characterization

ACM Reference Format:
Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh Bagchi, and Somali Chaterji. 2022. WiseFuse: Workload Characterization and DAG Transformation for Serverless Workflows. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 2, Article 26 (June 2022), 28 pages. https://doi.org/10.1145/3530892

Authors' addresses: Ashraf Mahgoub, amahgoub@purdue.edu, Purdue University, 465 Northwestern Ave., West Lafayette, Indiana, USA, 47907; Edgardo Barsallo Yi, ebarsall@purdue.edu, Purdue University, 465 Northwestern Ave., West Lafayette, Indiana, USA, 47907; Karthick Shankar, karthick@cmu.edu, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, Pennsylvania, USA, 15213; Eshaan Minocha, eminocha@purdue.edu, Purdue University, 465 Northwestern Ave., West Lafayette, Indiana, USA, 47907; Sameh Elnikety, samehe@microsoft.com, Microsoft Research, 1 Microsoft Way, Redmond, Washington, USA, 98052; Saurabh Bagchi, sbagchi@purdue.edu, Purdue University, 465 Northwestern Ave., West Lafayette, Indiana, USA, 47907; Somali Chaterji, schaterji@purdue.edu, Purdue University, 225 South University Street, West Lafayette, Indiana, USA, 47907.

2476-1249/2022/6-ART26
https://doi.org/10.1145/3530892

1 Introduction

Serverless workflows are becoming increasingly popular for many applications as they are hosted on a scalable infrastructure with fine-grained resource provisioning and billing [29, 45, 49]. A serverless workflow contains two or more serverless functions orchestrated as a DAG (Directed Acyclic Graph). Commercial providers offer orchestration services to facilitate the design and execution of serverless DAGs (*e.g.*, AWS Step Functions, Azure Durable Functions, and Google Cloud Workflows). The serverless platform, however, executes each function in the DAG in a separate VM without making use of the DAG structure and the data transfer among functions, which leads to significant increases in the end-to-end (E2E) latency and cost [1, 30, 35].

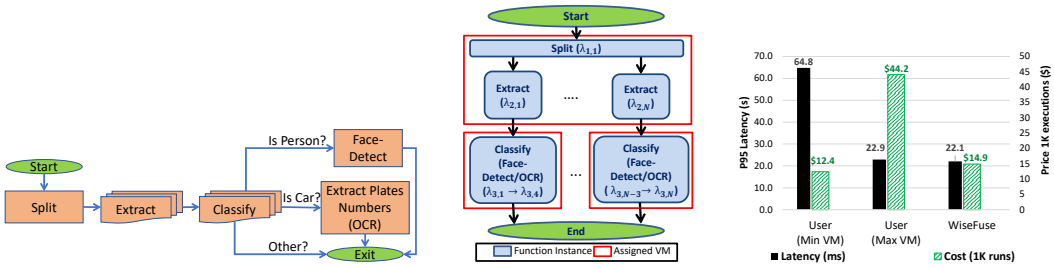
To understand the challenges of supporting serverless workflows, we study production workloads of serverless DAGs of *Azure Durable Functions* [5] over two weeks [1]. Our analysis shows that the vast majority of DAG executions are for recurring DAGs: the top 5% most frequent DAGs constitute 94.6% of all DAG invocations, with an invocation rate of at least 1.6K times per day.

With this high rate of invocations, the cloud provider can monitor the DAG execution parameters and quickly identify the bottlenecks to optimize the DAG execution. We identify two major performance bottlenecks: (1) *Communication latency between in-series functions*, which stems from the intermediate data that is typically passed through remote storage. (2) *Computation skew among in-parallel invocations of the same functions*, since each parallel invocation processes different content. Our analysis of the production DAGs shows that 46% of the DAGs have a high communication latency, and 48% of the DAGs have a high computation skew of 2× or more between parallel workers. Both factors are a direct result of the current state-of-practice where each function in the DAG runs in a separate VM.

Key Ideas: We propose WiseFuse, which is an automated approach to generate an optimized execution plan for serverless DAGs. We show an example of WiseFuse's optimizations in Figure 1, and the solution overview is schematically shown in Figure 2. Users provide WiseFuse with a DAG definition that includes individual functions as nodes and their data dependencies as edges, which is typical in today's commercial offerings . We introduce three optimizations. (1) **Fusion**: Combining in-series functions together as a single execution unit. (2) **Bundling**: Executing parallel invocations of the same function together in the same VM. (3) **Resource Allocation**: Assigning the best size to each VM hosting a function or a group of the DAG functions. First, we show that Fusion allows for optimized communication between cascaded functions due to better data locality between sending and receiving functions. Second, we show that Bundling mitigates execution skew among the colocated function invocations and therefore reduces latency. Finally, assigning the best sizes for each VM hosting one or more DAG functions allows WiseFuse to reduce the E2E latency and $ cost.

Challenges: Determining a good execution plan for a serverless DAG poses three technical challenges. *First*, serverless functions experience a high runtime variability, even when executed as standalone functions [1, 16, 29, 35, 39, 49]. This variability in runtime can be either in communication time (*i.e.*, while downloading/uploading data from remote storage), or in processing time. Hence, we need to model the communication time and processing time of individual functions as well as for the entire DAG as distributions, rather than as single-point estimates. *Second*, we also need to estimate the impact of Fusion or Bundling on the E2E latency distribution. The effect is non-monotonic, *e.g.*, bundling of a certain number of functions decreases E2E latency but excessive bundling increases it, and the same argument applies to Fusion. *Third*, the search space of all possible execution plans is massive, due to the large number of possible DAG widths and VM

[1] We release a subset of the production traces and this is available from https://github.com/Azure/AzurePublicDataset.



(a) Video Analytics pipeline: User-defined DAG.

(b) WiseFuse's execution plan.

(c) WiseFuse's gains.

Fig. 1: Example serverless DAG of Video Analytics application (1a) and the corresponding optimized execution plan generated by WiseFuse (1b). The gains in latency and cost over user-defined DAG, using either minimum or maximum VM size configurations (1c).

size choices, making exhaustive grid search infeasible. Therefore, we need to develop an efficient algorithm to optimize the execution plan.

Our Solution: To overcome these challenges, we make three technical contributions in WiseFuse. *First*, we create a distribution and correlation-aware performance model to capture the variability in performance for each function. Our model breaks down the function's runtime into download, processing, and upload components, while taking into account the correlation between in-series and in-parallel workers. By profiling the latency distributions of the three components for each function, we can identify stages that experience high communication latency (*i.e.*, high latencies in intermediate data download/upload) and hence can benefit from Fusion. We also identify stages with parallel invocations of the same function that experience execution skew (*i.e.*, long tail operation), and hence can benefit from Bundling. *Second*, we search for a good bundle size, and the right VM size to assign to each bundle. Using the performance model, we also estimate the impact of joint Fusion and Bundling of functions on the E2E latency and cost. *Finally*, we use the above two contributions to develop a searching strategy that performs a joint optimization of the combination of Fusion and Bundling operations.

To get a sense of the impact of these three contributions combined, consider the DAG for a video analytics pipeline in Figure 1a and WiseFuse's execution plan in Figure 1b. Compared to the user-defined DAG (without performing any Fusion or Bundling action), WiseFuse achieves 64% lower latency compared to allocating the Min VM size for each function, and achieves 66% lower cost over allocating the Max VM size for each function.

Evaluation: We evaluate WiseFuse using three popular serverless applications with different DAG structures and show significant improvements in E2E latency and cost compared to four approaches from recent work. For example, for the video analytics application, WiseFuse achieves 62% lower latency than Photons [13], which colocates parallel invocations together in the same VM to maximize memory utilization. WiseFuse also achieves 39% and 47% lower latency than communication optimization frameworks, Faastlane [30] and Sonic [35].

Contributions: We make the following contributions in this paper:

(1) We characterize production FaaS workloads for serverless workflows (*i.e.*, DAGs) of *Azure Durable Functions* over two weeks. From our analysis, we pinpoint two major performance bottlenecks: (a) high latency when transferring data between in-series functions. (b) Lack of resource sharing between in-parallel invocations of a function, leading to processing skew.

(2) We highlight two types of optimizations that can be performed by the cloud provider while

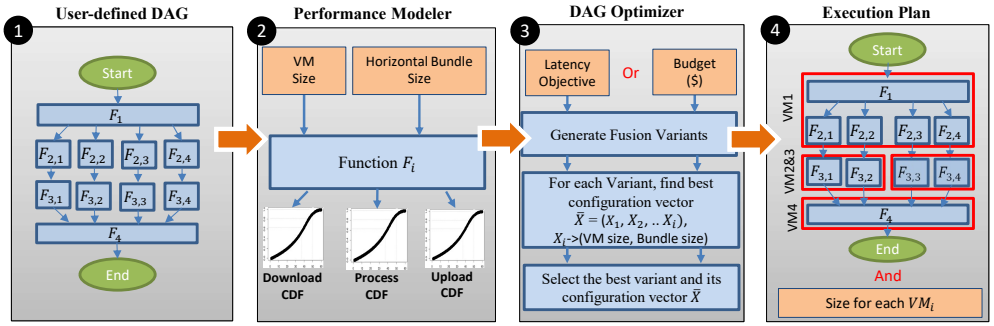


Fig. 2: Overview of WiseFuse design. WiseFuse optimizes the execution plan for the user-defined DAG, which includes combinations of Fusion (*vertical coupling of in-series stages*), Bundling (*horizontal coupling of parallel workers in a stage*) actions, the size of each VM host a function or a group of DAG functions.

executing a serverless DAG, *Fusion* and *Bundling*. We show that performing the two independently is counterproductive and we jointly determine a good combination of Fusion and Bundling actions. (3) We implement a DAG-aware execution plan optimizer that meets a user-given latency objective while reducing cost. The execution plan specifies which functions should be fused or bundled, as well as the appropriate VM sizes.

The rest of the paper is organized as follows: Section 2 motivates the optimizations in WiseFuse using workload characterization of Azure Durable Functions. Section 3 gives an overview of our design, encompassing Fusion (grouping of in-series functions), Bundling (grouping of parallel invocations of a function), DAG transformation, usage model, and finding the best configuration. Section 4 gives the implementation details for WiseFuse. In section 5, we evaluate WiseFuse on AWS Lambda against competing approaches — an approach that does a limited form of Bundling (Photons), and approaches that perform a variant of Fusion (Sonic and Faastlane), and the user-defined DAG baseline configured either with the largest VMs to obtain low latency or with the smallest VMs for lower cost. Further, we use three applications for the evaluation — Video Analytics, Approximate SVD, and ML Pipeline, and use microbenchmarks to evaluate the individual contribution of Fusion and Bundling. We evaluate WiseFuse on another cloud provider using the Approx SVD application to see if our techniques generalize. We discuss related Work in Section 6, and discuss several aspects of WiseFuse (*e.g.*, updating the performance model, scheduling, and security considerations) in Section 7. We present our conclusions in Section 8.

2 Workload Characterization

We analyze a subset of *Azure durable functions* production workloads over two weeks. Compared to a previous characterization of FaaS workloads [45], we focus here on characterizing serverless DAGs rather than single functions. Our analysis highlights important findings regarding serverless DAGs in the real world and motivates the optimizations of WiseFuse. We collected data on DAG executions for 14 days, between October 18th and October 31st, 2021, from six data centers, three in the US, two in Europe, and one in Asia. We are releasing publicly an anonymized subset of this data to spur research in serverless DAGs [6].

We give a general characterization of serverless DAGs in terms of invocation frequency and structure, then we provide specific characterizations to motivate the importance of Fusion and Bundling. Additionally, we use microbenchmarks to measure the data transfer latency for different data sizes at three major FaaS providers.

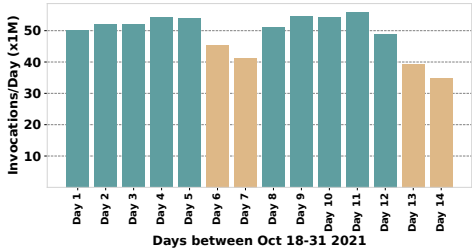


Fig. 3: Daily DAG invocations over 2 weeks. Weekend days are in yellow.

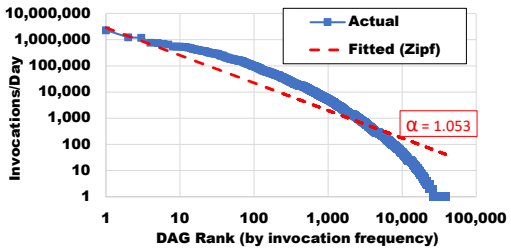


Fig. 4: Daily invocations per DAG vs DAG rank (by frequency). Top 5% DAGs have 94.6% of invocations.

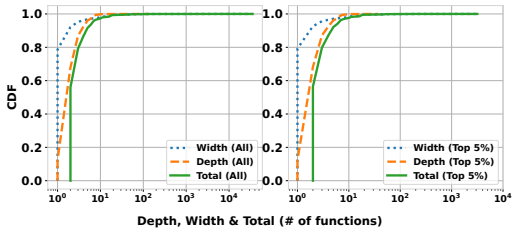


Fig. 5: Distribution of DAG depth (*i.e.*, stages), width (*i.e.*, degree of parallelism) and total nodes (left for all DAGs, right for top 5%).

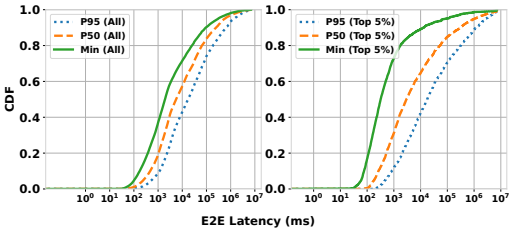


Fig. 6: CDFs for P95, P50, min of execution time per DAG (left for all DAGs, right for top 5%). P50 of all DAGs is 5.6 sec, whereas it is 3.1 sec for the top 5%.

2.1 General DAG Characterization

Definitions: A DAG is a sequence of stages S_1, S_2, ... S_d. The order of the stages in the DAG represent the order of execution (*i.e.*, stage S_i executes before S_{i+1}). A stage S is a set of functions that execute in parallel F_1, F_2, ..., F_n, and $n = 1$ for a stage with a single function. All functions within a single stage execute the same code, but can have different inputs (*e.g.*, scatter fanout stages) or same inputs but with different hyper-parameters (*e.g.*, broadcast fanout stages).

Daily Invocations: Figure 3 shows the daily DAG invocations collected data between October, 18^{th}–31^{st}, 2021. We notice that DAGs are consistently invoked at a high rate of 34M-55.8M invocations per day. By analyzing earlier data, we notice that the total number of DAG invocations per day has grown by 6× over the past 2.5 years, suggesting that this workload is growing rapidly.

Invocations per DAG: Now we study the frequency of invocations for each DAG. We collect the total invocations of each DAG across all 14 days and show the average daily invocations in Figure 4. We notice that the invocation frequency varies significantly, and the invocation frequency follows a Zipf distribution, which is well-known for modeling rank-frequency relations. For the fitted Zipf distribution, the goodness of fit Z-test achieves a p-value of 0.052 (*i.e.*, above 95% confidence interval).

Notice that the majority of DAG invocations are for recurring DAGs: The top 5% most frequent DAGs constitute 94.6% of all DAG invocations, and their invocation rate is ≥ 1.6K/day. Therefore, improving the performance and cost of these DAGs yields large gains for both users and the cloud provider.

DAG Structure: We now look at the structure of the DAGs. We collect the number of stages (*i.e.*, depth) and fanout degree (*i.e.*, width) in DAG executions. We use the maximum number of parallel functions across all fanout stages to determine the width of the DAG. Note that if the DAG is invoked K times, it contributes K data points for the width and depth distributions. Figure 5

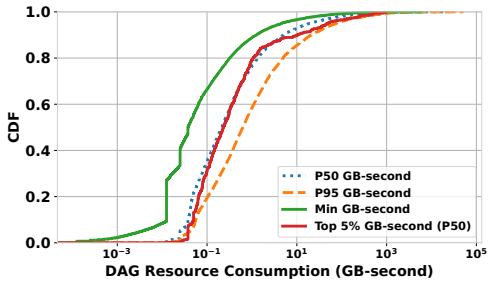


Fig. 7: For 50% of DAGs, the median invocation consumes more than 210 MB-sec. For top 5% most frequent DAGs, this number is lower, at 230 MB-sec.

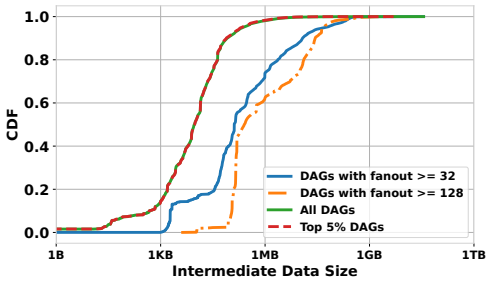


Fig. 8: CDF of total intermediate data files passed across the entire DAG. Higher intermediate data sizes are observed with the top 5% most frequent DAGs.

shows the CDFs for the DAG widths, depth, and total number of functions. The median number of functions in the DAG is 3, whereas the median decreases to 2 for the top 5% most frequent DAGs. We also notice that 60% of all DAGs have a width ≥1, for which Bundling can potentially be applied. For the top 5% most frequent DAGs, the ratio of DAGs that can be bundled decreases to 44% and the remaining 56% DAGs are linear chains. DAG depth grows faster than the width till a crossover point (≈80th percentile). At the tail, DAG depths are usually short and the max depth is 47, whereas DAG widths are longer and the max width is 10.9K. This is an important observation that we leverage in WiseFuse's design to reduce the search space, as described in Section 3.6.

E2E Latency for Serverless DAGs: Here we show the distribution of the latency of all DAGs invocations as well as for invocations from the top 5% DAGs. We plot the CDFs for Min, P50, and P95 latency in Figure 6. We notice that the median execution time for all DAGs (*i.e.*, median of medians) is 5.6 sec, whereas it decreases to 3.1 sec for the top 5% DAGs. We also notice that the variance in execution time between invocations of the same DAG is high (not captured in the figure). Specifically, the ratio between P95 and P50 is 3× on median, and the ratio between P50 and Min is 4.8× on median. Hence, it is essential to model the E2E latency of the DAG as a distribution to capture this variance. Additionally, we contrast the DAG invocation latency to the single function invocation latency from a prior study [45] that reported the median latency for serverless functions is 670 ms. Thus, serverless DAGs are longer running than individual functions, stressing the importance of optimizing DAG E2E latency and cost for users as well as for FaaS providers.

DAG Resource Consumption: The cost of single DAG invocation is calculated as the product of consumed resources (memory is the observable parameter) by the execution time. To analyze the amount of resources consumed per DAG, we capture the resource consumption of its invocations, providing a distribution per DAG. We plot percentiles of that distribution for each DAG in Figure 7 in GB-sec units. For half of all DAGs, the median invocation consumes more than 210 MB-sec, and the median P95 is higher than 630 MB-sec. The max consumed memory per DAG (not observed in the figure) is 1 GB-sec on median. Moreover, for the top 5% most frequent DAGs, a lower median of 230 MB-sec is observed. To understand how resource consumption impacts the $ cost, we give an example for pricing in Azure Functions, AWS Lambda, and Google Cloud Platform (GCP). For a DAG that consumes 1 GB-sec per invocation, the cost of 1M invocations is $16 on Azure Durable Functions, $16.6 on AWS Lambda, and $16.5 on GCP. This motivates the need for allocating the right resources for functions in the DAG to reduce its latency *and* cost.

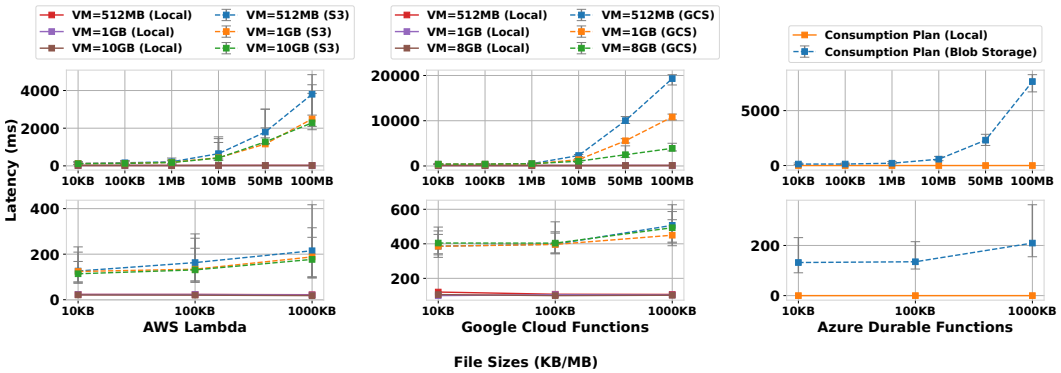


Fig. 9: Data exchange latency on AWS Lambda, Google, and Azure comparing remote storage to local communication. Markers represent the median, and error bars represent Min to P95 latency range.

2.2 Data Transfer Between Stages

Volume of Intermediate Data: Here we analyze the total volume of intermediate data being transferred between all stages in a DAG. We show the distribution of intermediate data sizes in Figure 8. We notice that 85% of the transferred intermediate data are of sizes ≥ 1KB and the median is 8KB. When we focus on DAGs with parallel stages, we notice that the size of intermediate data increases with higher fanout degrees. For example, the median data size for DAGs with stages of 32 workers or more is 710KB, which is 88× the median intermediate data size for all DAGs. Finally, the size of intermediate data for the top 5% DAGs is not significantly different from that for all DAGs, with a median of 8.4KB. Therefore, Fusion is essential to reduce latency and cost for all DAGs in general, and it is more beneficial for wider DAGs in particular.

Latency for Transferring Intermediate Data: We run a benchmark on AWS Lambda, GCP, and Azure Durable Functions to characterize the latency for exchanging data between serverless functions. We use a simple linear chain of a single sender and a single receiver function. The sender generates a random array of bytes of a particular size and uploads it as a file to remote storage (S3 for Lambda, Google Storage for GCP, and Blob Storage for Azure), then the receiver downloads the file from remote storage. We measure the E2E latency for the chain and subtract the generation and reconstruction times, hence the remaining time becomes the upload/download times. Figure 9 shows the median, Min, and P95 latency when exchanging different data sizes through remote storage. We vary the VM sizes for both sender and receiver functions between 512 MB and 10 GB (Lambda's Max) or 8 GB (GCP's Max), whereas we use the consumption plan for Azure which assigns resources automatically to the sender and receiver functions.

We make the following observations: (1) For AWS Lambda, increasing the VM size from 512 MB to 1 GB yields a significant reduction in data passing latency for file sizes > 1 MB. However, increasing the VM size further (> 1 GB) does not reduce the communication latency indicating that the network bandwidth saturates at this point. On the other hand, for GCP, increasing the VM size increases the network bandwidth but sub-linearly (16× increase in VM size causes a 4× speedup in data transfer). Hence Fusion becomes essential to reduce data exchange latency and cost in both platforms. (3) Data exchange latency has a high variability, with a ratio of 8× between the P95 and the median, and a ratio of 3.4× between the median and the minimum latency for AWS Lambda. For GCP and Azure, a lower variability is observed (ratio between P95 and median is between 18% and 80%), but with a lower network bandwidth compared to AWS Lambda. Hence, it is essential to represent the upload and download times as a distribution to capture that variability. (4) Using

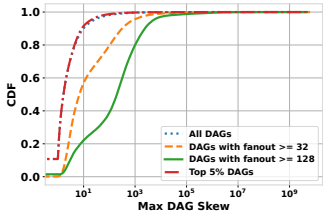


Fig. 10: CDF of DAG max skew for all DAGs, top 5%, for DAGs with fanout ≥ 32 & ≥ 128.

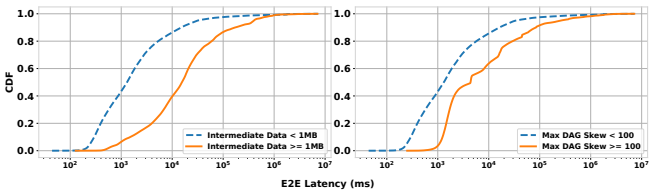


Fig. 11: CDFs of DAG latency per max skew (right) and intermediate data size (left). Longer E2E latencies are observed for DAGs with higher max skew and intermediate data size.

Fusion, the data exchange latency is reduced significantly for all file sizes and for all platforms (as observed by comparing *local* to *S3, GCS, or Blob Storage* variants.). Moreover, gains are higher for bigger file sizes making Fusion more beneficial for data intensive applications.

2.3 Skew among Parallel Workers

We analyze the degree of runtime skew observed among all parallel stages in the DAG. For each stage, we calculate the degree of skew as the ratio between the runtime of the *slowest* worker to that of the *fastest* worker. If the DAG has multiple parallel stages, we take the maximum skew across all parallel stages. Recall that the E2E latency of the stage (and the entire DAG) is dominated by the slowest worker in the stage. We show the distribution of DAG skew in Figure 10. We notice that the median skew is 1.9×. When we focus on DAGs containing parallel stages with higher fanouts, the median skew rises to 15× and 132× for DAGs containing stages with 32 and 128 parallel workers, respectively. For the top 5% DAGs, the median is close to 1 (no skew) but 32% of the top 5% DAGs have a skew greater than 2×. Thus, skew among parallel workers is significant and that motivates the need for efficient skew mitigation optimizations for serverless DAGs, which we achieve by Bundling.

2.4 Impact of DAG Skew and Intermediate Data Size on Latency

Finally, we analyze the impact of skew and intermediate data size on the distribution of DAG E2E latency. We split all DAGs into two groups with respect to their average skew (skew < 100 and skew ≥ 100). Similarly, we split all DAGs into two groups with respect to their average intermediate data size (size < 1MB, and size ≥ 1MB). We show the E2E latency distributions for the four groups in Figure 11. We notice that both skew and intermediate data size have a significant impact on the DAG E2E latency. Specifically, DAGs with skew ≥ 100 have 17× higher median latency than DAGs with skew < 100. Similarly, DAGs with intermediate data size ≥ 1MB have 9.5× higher median latency than DAGs with size < 1MB. This motivates our focus on these two factors to optimize through Bundling (reducing execution skew) and Fusion (reducing intermediate data passing latency).

3 Design

First, we give an overview of WiseFuse's design and its components. Second, we give the details for building a performance model for each function in the DAG, which we use to estimate the impact of Fusion or Bundling on the E2E latency and cost. Finally, we show how WiseFuse explores the vast search space of actions for Fusion, Bundling, and VM size allocation to find the best execution plan.

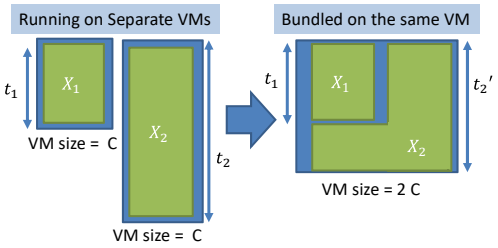


Fig. 12: Without Bundling, latency is dominated by the straggler (X_2) and is equal to t_2. With Bundling, X_2 gets more resources after X_1 executes, decreasing the stage's latency to t_2'.

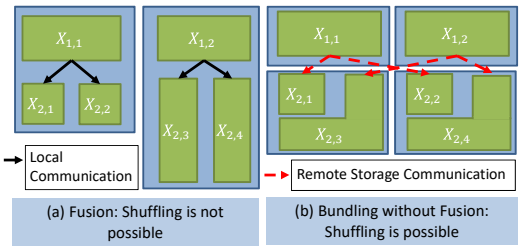


Fig. 13: Coupling between Fusion and Bundling. Fusion reduces data exchange latency but prevents shuffling, which is essential to break the locality of stragglers and reduce the E2E latency.

3.1 Overview

Fusion: Fusion combines two consecutive stages to reduce the data exchange latency between them. For example, performing Fusion of stages S_i and S_{i+1} combines *each* sending function in S_i with its receiving functions in S_{i+1}. Accordingly, Fusion needs to take into account the communication primitive between the two stages. For example, if the communication primitive between stages S_i and S_{i+1} is one-to-one (*i.e.*, each function in S_i is sending data to only one function in S_{i+1}), then the Fusion is performed in one-to-one manner and the resulting stage will have the same degree of parallelism (DOP) as both stages. However, if the communication primitive is one-to-many, many-to-one, or many-to-many, Fusion will generate a single stage with a DOP of $Min(DOP(S_i), DOP(S_{i+1}))$. Finally, if the communication primitive is all-to-all (*i.e.*, shuffling), Fusion combines all functions in stage S_i with all functions in stage S_{i+1}, producing a stage with a single function.

Bundling: Bundling is the operation of colocating two parallel workers together to run on the same VM, hence enabling resource sharing between them (Figure 12). Higher bundle sizes are achieved by recursively applying the same operation, leading to bundle sizes that are powers of 2. Bundling reduces execution time skew by allowing stragglers to access more resources (*e.g.*, CPU capacity and/or Memory) once other colocated workers finish execution. For Bundling to be useful, the function has to be scalable, i.e., given more resources it should be sped up (till a point). Notice that Bundling does *not* require knowledge/prediction of which invocation is the straggler. However, for Bundling to be efficient, stragglers need to be spread out over different bundles as much as possible, which WiseFuse achieves through shuffling (Figure 13).

Bundling also reduces data exchange latency in two cases. First, for broadcast stages, our bundling technique leverages the fact that all workers within a single bundle can read from the same local copy of the input data, and hence downloads the data once per bundle. Second, in case of a skew in the intermediate data sizes between the two parallel workers, Bundling allows for resource sharing so that the straggler gets a higher network bandwidth, speeding up its upload and download operations.

DAG Transformation: WiseFuse takes as input a user-defined DAG, with each function denoted as a separate node. By applying a sequence of Fusion and Bundling operations, the original DAG is *transformed* to a new DAG with fewer number of nodes. WiseFuse's target is to identify the best combination of Fusion and Bundling operations that transforms the DAG into the best performing DAG in terms of E2E latency and cost. The transformations do not need any additional effort or code changes from the user. Figure 1 shows an example of a user-defined serverless DAG (1a), WiseFuse's transformed DAG (1b), and the gains after the transformation (1c).

3.2 Workflow and Usage Model

We show the main design components of WiseFuse in Figure 2. First, similar to current commercial FaaS platforms, the user provides WiseFuse with the DAG definition, which includes the executable package for each function and the data dependencies between the functions in the DAG.

Second, we execute WiseFuse's `performance modeler`. We represent each function as a composition of three steps: download (input data), process, and upload (output data). We then perform per-function profiling and latency modeling to capture the variability in each of these three steps. This is essential for us to estimate the gains of fusing in-series functions together in a single VM, or Bundling in-parallel functions together in the same VM. For example, functions that have a high latency in the download or upload steps can benefit more from Fusion due to removing the data exchange latency between in-series functions. On the other hand, functions that experience latency skew in the process step can benefit more from Bundling due to resource sharing.

The `modeler` also estimates the degree of correlation between latencies of either in-parallel or in-series functions, which is essential to correctly estimate the impact of performing Fusion or Bundling. The output of the `modeler` is a performance model for each stage in the DAG that can estimate the latency distribution for each of the three steps (download, process, and upload) given a candidate VM size and bundle size.

Third, the `DAG optimizer` uses the generated performance models and explores the vast search space of Fusion, Bundling, and VM size allocations for the entire DAG. Next, it proposes the best set of transformations that leads to a new DAG which is given to the user for approval before deployment.

Usage model: In the typical usage model, the cloud provider deploys WiseFuse and applies it to serverless DAGs submitted by the clients. The cloud provider does the profiling runs needed for the `performance modeler`, without charging the client. In its profiling runs, the vendor cannot use content characteristics to enhance its model, due to data privacy limitations. It can however use metadata like the size of the input. Alternately, WiseFuse can be deployed by users as a user-side optimization tool. In this case, the users can provide hints of content to WiseFuse such as the input category (*e.g.*, for Video-Analytics application, input category can be: Sports, News, etc.). WiseFuse can use these content-aware hints to build a separate performance model for each input category and further improve the estimation accuracy of the model. Further, the user can apply WiseFuse at her end to do the DAG transformation, using Fusion and Bundling, on her original application DAG. She then submits the transformed DAG to the serverless platform of the cloud provider.

3.3 Per-function Performance Model

The first step is to build a performance model that maps the amount of resources for a given function to the expected latency distribution. Notice that modeling the latency distribution is important for both latency-sensitive and cost-sensitive applications. This is because of the pay-as-you-go cost model of serverless platforms, which bills the user proportionally to the product of the allocated resources and the function's runtime. We create this latency distribution separately for the three phases of a function's execution — data download, execution latency for the function itself, and data upload.

Profiling: To start off, WiseFuse profiles the latency distributions for 4 VM sizes, including Min and Max VM sizes. *Min implies the lowest size at which the function will execute and Max implies the largest VM size supported by the FaaS platform.* The value of the intermediate VM sizes vary for different FaaS platforms. For AWS Lambda, there are fine-grained VM sizes supported — 128 MB to 10 GB in step sizes of 1 MB. We do not wish to profile for all possible VM sizes and we leverage discontinuities that exist in the vendor offerings. Hence, we pick 1,024 MB as one intermediate

profiling point as that is the size at which AWS saturates the network bandwidth (*i.e.*, increasing memory beyond it does not provide any more network bandwidth as shown in Figure 9). We pick 1,792 MB as another intermediate profiling point as a VM with one full core gets assigned at that point. For GCP, there are only 7 VM sizes to pick from and here we can afford to profile for all sizes.
Interpolation: This initial profiling divides the configuration space into multiple regions. For example, for AWS Lambda, this divides the VM size space into 3 regions: Min-1024, 1024-1792, and 1792-Max. Afterward, WiseFuse performs percentile-wise linear interpolation to infer the CDF for the intermediate memory settings. For example, the P50 latency for 6 GB is estimated as the average between the P50 latency of 1.8 GB and of 10.2 GB. This generates a *prior* distribution for these intermediate memory settings. To verify the prediction accuracy in a region, WiseFuse collects a few test points using the midpoint memory setting in that region to measure its actual CDF (*i.e.*, the *posterior* distribution) and compares it with the *prior* distribution. If the error between the *prior* and *posterior* CDFs is high (*i.e.*, $\geq 15\%$) in any region, WiseFuse collects more data for the midpoint in that region and adds it to its profiled points, splitting that region further into two smaller regions. This process is repeated till saturation in accuracy is reached for all regions. In practice, we find that dividing the space into 5 regions is sufficient to achieve an error $\leq 15\%$ for all latency percentiles.

3.4 Estimating the Impact of Fusion

Here we show how WiseFuse estimates the impact of fusing two in-series functions together. We represent the latency of a function as a composition of three components: (1) Download-duration (*Down*): the time to read the input file(s) from remote storage. (2) Process-duration (*Proc*): the computation time. (3) Upload-duration (Up): the time to upload the output file(s) to remote storage. We model the function's latency PDF as a *convolution* between the three components:

$$P(latency = t) = P(Down = k, Proc = m, Up = t - k - m) \quad (1)$$

Notice that we still represent each component as the distribution of a random variable, rather than a constant, to capture the latency variability. By factorizing the latency into these three components, we can easily estimate the impact of fusing two functions on their combined latency and cost. For example, if we perform Fusion between `Extract` and `Classify` stages in Figure 1a, we remove the upload step from `Extract` and the download step from `Classify`. Thus, we estimate the combined latency as a convolution between *Down*(Extract), *Proc*(Extract), *Proc*(Classify), and Up(Classify) CDFs. Importantly, WiseFuse takes into consideration the correlation between the different components to efficiently estimate the convolution between them. For example, we estimate the Pearson correlation coefficient between all four components and find a strong correlation of 0.73 between *Proc*(Classify) and Up(Classify) CDFs. This is because it takes longer to perform object detection for frames with many objects; it also takes longer to crop and upload the detected objects. Hence, in WiseFuse, we use the marginal distributions for the statistically independent components, while we use the joint distribution for highly correlated components (*Proc*(Classify) and Up(Classify) in this case). We show the accuracy of WiseFuse's estimates for the impact of Fusion in Section 5.4.2.

3.5 Estimating the Impact of Bundling

Here we show how our model estimates the impact of bundling parallel invocations together on their combined latency distribution. By bundling parallel invocations, stragglers can leverage additional resources released by the fast executing workers (Figure 12). This allows for local resource sharing between the bundled workers and decreases their combined latency. However, Bundling can cause contention if the number of bundled invocations is high compared to the VM's size. Accordingly,

Algorithm 1 Get Bundled-Pair Latency

Input: NumIterations=N, PerfModel: Model, WorkerSize: C
Output: Latency CDF for bundled pair: CDF_{bundle}

```
## Use performance model to get the CDF for a single worker in the bundle with VM of size C
CDF_Single = Model(C)
## Use performance model to get speedup distribution for a single worker when executed on a VM of size 2C
CDF_Double = Model(2C)
## divide the percentiles of the CDF with double sized VM over the CDF with single sized VM
CDF_Speedup = CDF_Double/CDF_single
for i = 0− > N do
    ## Sample 2 latency points t1, t2 that where observed in the same run (to sustain the correlation)
    Sort the points t1<t2
    Estimate skew = t2-t1
    Set reduced-skew = CDF_Speedup(skew) * skew
    Add (t1 + reduced-skew ) to CDF_bundle
end for
return CDF_bundle
```

Fig. 14: Pseudo code for calculating the latency CDF for any sized bundle of functions in one stage

selecting the best bundle size is not trivial and it has to be carefully selected. Therefore, Bundling is beneficial and used when: (1) The DAG has a fanout stage since linear-chains do not benefit from Bundling. (2) Functions are scalable: The function can leverage additional resources when made available. (3) Input content skew: Stragglers experience longer execution times due to their input content, not variability due to infrastructure (*e.g.*, poor network bandwidth). (4) Stragglers can be spread out over different bundles, which we achieve through shuffling.

For simplicity of design, we consider that workers can be bundled in powers of two only. Further, all bundle sizes within one stage are equal. We give the steps for estimating the latency distribution of a bundle of workers in Algorithm 1. First, we use the performance model to estimate the latency distribution for a single function when assigned additional resources. For example, if the function is originally allocated a VM size (C) and then is bundled with another function in a VM with double the size ($2C$). The ratio between $\frac{CDF(2C)}{CDF(C)}$ shows the speedup due to additional resources, *i.e.*, the function's scalability. We then draw pairs of latency points from our profiles to capture the natural skew observed between the two workers. Then, we estimate the reduction of that skew (due to Bundling) using the speedup CDF. In summary, the algorithm considers two important factors: (1) The speedup distribution due to additional resources (2) The natural degree of skew observed between the two workers.

Notice that we represent the speedup as a distribution rather than a scalar value, which is essential to capture the reduction in latency for each latency percentile. For example, Figure 15 shows the two CDFs for the `Classify` function in our Video Analytics application, using 1 core VM vs 6 cores VM. We notice the speedup varies for different percentiles. For example, the speedup on the median is only 11%, whereas it becomes 47% for the P95. This is because invocations at higher percentiles experience higher skew and therefore reap greater benefit from additional resources. A stage's latency is estimated as the "Max" latency among all bundles (stage i with N bundles):

$$P(X_i \leq z) = P(X_{i,1} \leq z, X_{i,2} \leq z ... X_{i,N} \leq z) \tag{2}$$

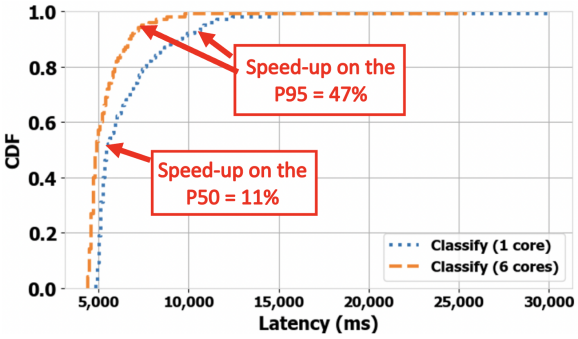


Fig. 15: Speed-up in `Classify` when executed on a 6-core VM vs a 1-core VM.

Similar to Eq 1, identifying the correlation between the parallel workers is essential to accurately estimate the stage's latency distribution. Specifically, in case of a low correlation between parallel workers, we can simplify Eq 2 as the multiplication of the marginal distributions, $P(X_i \leq z) = P(X_{i,1} \leq z)^N$. However, in case of a high correlation, this simplification can cause overestimation of the combined latency. In our applications, we observe a high enough correlation coefficient (0.4-0.6) between parallel workers, hence we use the joint distributions when estimating Eq 2.

3.6 Execution Plan Optimization

Here we show the steps for finding an optimized execution plan for a user-given latency objective.
Search Space: We calculate the size of DAG transformations, including Fusion, Bundling, and VM size selection. Let D be a DAG of N stages, denoted as $S_1, S_2, ..., S_N$. Since we can perform Fusion of any consecutive pair of stages S_i and S_{i+1}, we have $N-1$ consecutive pairs, and for each pair, we have a binary decision (to fuse or not to fuse). By performing this decision recursively, we actually explore *all* possible Fusion configurations between consecutive stages (not just pairwise Fusion). Hence, we have a total of $2^{(N-1)}$ possible combinations of Fusion actions. Since in practice N is not large (P50 depth is 2 and P99 is 12, Figure 5), this exhaustive search of Fusion space is feasible. Now, let the maximum degree of parallelism in any stage be 1K (recall that the actual observed P99 width is 84). Since we consider bundle sizes to be powers of 2 only, we have a maximum of $log_2(1000) \approx 10$ possible bundle sizes for each stage. Notice that after we perform Fusion for any pair of stages, they are represented as a single stage in the new DAG. Therefore, the total number of Fusion and Bundling actions is given by $\sum_{i=1}^{N} \binom{N-1}{i-1} \times 10^i$, where i represent number of stages after performing any combination of Fusion actions, and it varies between 1 (all stages are fused into one stage), and N (no Fusion). For a DAG of 8 stages and max degree of parallelism of 1K, we have 215.6M possible Fusion and Bundling actions that we can select from. Notice that this space does not include the possible actions of selecting the VM sizes, which can be as few as 7 different sizes for Google Cloud [22], or as many as any memory size (in steps of 1 MB) between 128 MB and 10.24 GB for AWS Lambda [3]. Thus, an exhaustive search to find the best plan is expensive and impractical.
Importance of E2E Optimization: Here we show the importance of selecting Fusion, Bundling, and VM allocation options for all stages in the DAG *jointly*, rather than optimizing each stage separately. Consider a DAG of 2 stages such that the first stage has only one function denoted as X_1. The second stage has k parallel invocations of a function, denoted as $(X_{2,1}, X_{2,2}, ... X_{2,k})$ and they receive their input files from X_1. Now, in order to achieve data locality between the two stages and minimize X_1's latency, we perform Fusion. This forces us to execute the sending function and

all receiving functions in the same VM. Thus, we force Bundling with a bundle size of k over the second stage. This can be harmful for large values of k where Bundling can cause contention in the VM and increase the E2E latency and cost.

Another cause of local optimization shortcoming is that Fusion removes the possibility to "shuffle" stragglers across bundles, which is needed in many cases to break the locality of stragglers. For example, consider the example shown in Figure 13a for a stage of two workers $X_{1,1}$ and $X_{1,2}$ and each of the them triggers two workers in the second stage. In the second stage, workers $X_{2,3}$ and $X_{2,4}$ are stragglers. If we perform Fusion between the two stages (which is the best local decision for workers $X_{1,1}$ and $X_{1,2}$), we will have to execute the two stragglers together in the same bundle and hence the locality of stragglers will remain. In contrast, if we do not perform Fusion between the two stages, we can spread the two stragglers over two separate bundles, which leads to a lower E2E latency for the bundles (Figure 13b). Accordingly, selecting the best Fusion and Bundling actions for each stage separately (*i.e.*, local optimization) is sub-optimal. WiseFuse avoids this pitfall by selecting the execution plan (which includes Fusion and Bundling options for all stages) that optimizes the E2E latency (or cost) for the entire DAG.

Search Strategy: As shown in Figure 2, the `DAG optimizer` takes as input either a latency objective (for latency-sensitive applications) or a budget (for cost-sensitive ones). The first step is to generate different Fusion variants from the user-defined DAG, and by default, we explore all Fusion variants. However, if exploring all Fusion variants is infeasible in a specific situation, we can select a subset that fuses stages with high data exchange (upload/download) volume. Afterward, a searching algorithm is executed for each variant *in parallel* to identify its best configuration vector $\bar{\mathbf{X}}$, which denotes the best VM sizes and bundle sizes for each stage in that variant. After the configuration vectors are generated for all Fusion variants, we select the vector that meets the latency objective with the lowest cost. Equivalently, for the budget objective, we select the vector that meets the budget objective and provides the lowest latency among the options.

Handling Different Input Sizes: The same serverless DAG can be executed with different input sizes, which can impact the execution time, memory footprint, or the fanout degree of its functions. Accordingly, different input sizes can be optimally executed with different execution plans. WiseFuse leverages polynomial regression to estimate the upload, process, and download CDFs for new (unseen) input sizes. The order of the polynomial is different for different applications and stages. For example, upload and download CDFs of `PCA` stage in our ML pipeline application have a linear relation while its process CDF has a quadratic relation with input size (PCA has quadratic compute complexity [50]). We evaluate WiseFuse's ability to handle different input sizes in Section 5.4.3.

Finding the best configuration vector: We show the pseudocode for WiseFuse's optimizer in Algorithm 2. First, we estimate the latency distribution for the fused stages as shown in Section 3.4. Afterward, we apply a Dynamic Programming (DP) search algorithm to select the best VM size and bundle size for each stage. We observe that our optimization problem can be reduced to the well known **Knapsack problem**. We set the knapsack's capacity to be our latency objective for the latency-sensitive application (equivalently, the capacity would be our $ cost objective for a cost-sensitive application). The items in the knapsack represent configurations of each stage (one configuration/stage). The weight of each item is the latency of that configuration and the cost of each item is the inverse of the $ cost of that configuration (since our target is to minimize the cost subject to meeting the latency objective). The algorithm proceeds as follows: For each stage in the DAG, we estimate the latency and cost for each feasible action, which includes all VM sizes (in steps of 128 MB) and all bundle sizes (in powers of 2). We divide the latency objective into equal-sized windows (10 ms) and actions that lead to the same latency window are considered equivalent. Hence only the action that has the least cost is saved in the DP table, while others

Algorithm 2 Pseudocode for WiseFuse's optimizer

Input: DAG = D with N stages, PerfModel = Model, Latency Target = T
Output: Execution Plan = ExecPlan

```
## From DAG D, get all m = 2^(N-1) Fusion variants DV_1, ... DV_m
## Initialize Set_X̄ as empty set
for i = 0 to m (in-parallel) do
    Set Vector X̄ = DP(DV_i, Latency Target T)
    Add X̄ to Set_X̄[i]
    Set Cost[i] = GetCost(DV_i, X̄, Model)
    Set Latency[i] = GetLatency(DV_i, X̄, Model)
end for
## Across all variants, find best variant index i_best
such that Latency[i_best] ≤ T and Cost[i_best] is minimum
return ExecPlan = (DV_{i_best}, Set_X̄[i_best])
```

are pruned. By doing so, we keep the number of solutions that gets transferred from one stage to the next bounded (at most A×T), leading to a much faster searching time. Specifically, the time complexity is O($S \times A \times T$), where S is the number of stages, A is the number of possible actions per stage, and T is the number of latency buckets (Latency Target/10 ms). For a DAG of 8 stages, 46 VM sizes, and 10 bundle sizes for each stage, we have 3.68K actions to explore using DP, compared to 215.6M for the exhaustive case.

3.7 Further Design Considerations

Interaction with Cold Starts: WiseFuse's performance modeler profiles the latency distribution for each function in the DAG using different inputs. During profiling, DAG invocations are performed serially and in quick succession to leverage warm VMs and to eliminate cold starts. Accordingly, the latency distribution of our model captures the variability that is due to input content only, not cold starts. One positive side effect of performing Fusion or Bundling is that they cause the DAG execution to use fewer VMs. Hence, they can reduce the incidence of cold starts.
Sharing Functions between Multiple DAGs: WiseFuse does not limit sharing of the same function between multiple DAGs. This is because both Fusion or Bundling are implemented during the deployment phase, whereas the functions' user-defined packages and source codes are still separately shared and edited by users to preserve modularity of the functions.
Handling Deeper DAGs: The optimizer shown in Algorithm 2 explores *all* Fusion variants for the input DAG, which can be a large pace to explore in case the DAG has many stages. Recall that for a DAG of N stages, we have $m = 2^{(N-1)}$ Fusion variants. Although the Fusion variants are explored in parallel, the computation cost for the optimizer can become problematic if the number of stages grow. To reduce the size of this search space, we can construct an ordered list of pairs of stages combined with the median data communication latency between them: $[(S_i, S_{i+1}, \mu_{comm})]$. Afterwards, we can limit the optimizer to explore Fusing the subset of pairs of stages that have the highest data communication latency between them, and for which Fusion will be most efficient.
Mitigating Infrastructure Skew WiseFuse 's Bundling objective is to mitigate input-based (demand-side) execution straggler. However, Bundling has a positive side effect of reducing the overall number of VMs/containers used to execute a stage and therefore can indirectly minimize the infrastructure-based (supply-side) variability as well. For example, with Bundling of broadcast fanout stages, each bundle downloads the input file once and hence the file is made available for all invocations at the same time. This removes any possible variability in download times (e.g. due to network fluctuations) between the invocations in case they were executed on separate VMs.

Using Homogeneous Bundle Sizes: Our solution considers homogeneous bundle sizes only. It might seem more beneficial to bundle parallel workers into heterogeneous bundle sizes according to their execution times. For example, a better solution can be executing straggler workers as standalone functions with high resources, while bundling short running workers together with low resources. Unfortunately, this approach requires identifying based on content, which workers are likely to become stragglers, something that providers are unable to do due to privacy regulations.

4 Implementation

We implement WiseFuse in C# with 2.7K LOC. The runtime of our search heuristic is 6 ms for Approx SVD, 535 ms for ML pipeline, and 779 ms for Video analytics. WiseFuse collects execution times for N invocations per function to build its performance model, and the value of N is user-defined. We find empirically that N=300 is sufficient to model P95 latency with error $\leq$ 15% for our three evaluation applications. For higher percentiles (*e.g.*, P98), we would need to profile more points. In general, more points need to be profiled for accurate predictions for higher percentile latencies (e.g. we need at least 1000 points to profile the P999 latency). The profiling cost with N=300 on AWS Lambda is $3.8 for Approx SVD, $0.9 for ML pipeline, and $1.7 for Video analytics, whereas it is $3.4 for Approx SVD on Google. In our evaluation on AWS Lambda, we use StepFunction [2] for function orchestration. However, because of data transfer quota limits in Google Cloud Functions (details in Section 5.5), we use remote storage triggers to orchestrate the functions in the DAG.
Fusion: We follow the same programming paradigm as used in current FaaS platforms, where developers identify a `handler` for each function, which serves as the function's entry point. When two functions are to be fused together, their execution packages and dependencies are combined and a single wrapper is added to simply call both function handlers in order. Then, we intercept function calls to remote storage API to redirect the communication to the next fused function through local memory. In case WiseFuse decides *not* to fuse the two stages, the upload and download commands are forwarded to the remote storage API and their latency is recorded. Thus our implementation abstracts away from the user whether Fusion is used, i.e., whether data is passed through remote storage or local memory.
Bundling: To perform Bundling, we rely on parallelism APIs supported by the runtime (*e.g.*, Python3's multiprocessing API) to invoke multiple parallel instances of the function's handler. Note that we only bundle invocations of the same function and hence the bundled invocations share the same execution packages and dependencies. Moreover, we bundle invocations that belong to the same DAG execution (and hence belong to the same user) for data privacy considerations. In case the bundled functions are data dependent on the same input file (*i.e.*, Broadcast fanout), we download their input file once per bundle. We expect the VM's OS to dynamically assign more resources (*e.g.*, more CPU capacity) to stragglers after shorter running workers complete their execution.

5 Evaluation

5.1 Baselines and Competing Approaches

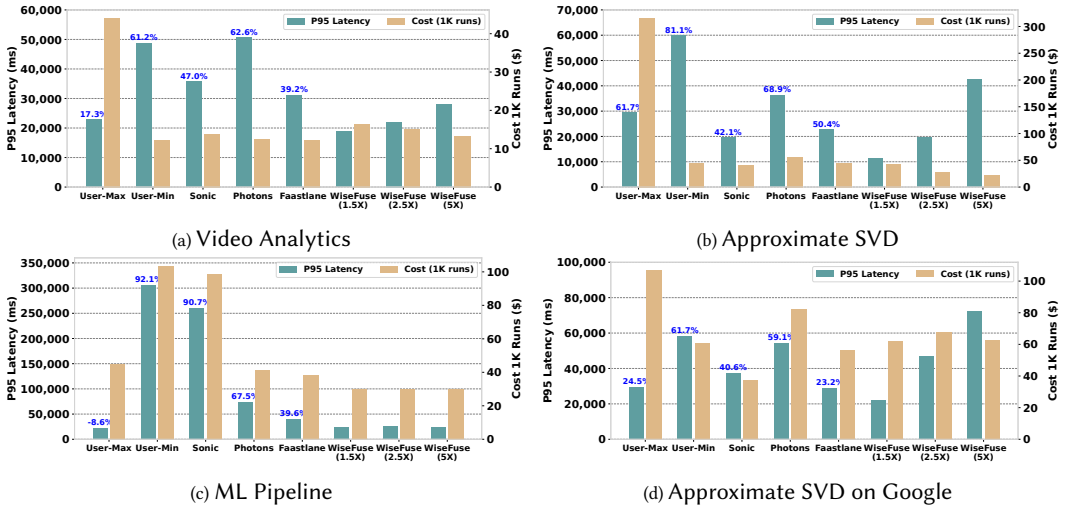
We evaluate the latency and $ cost for each application using WiseFuse's execution plan versus the following baselines and competing approaches.
1. **User-defined DAG**: This is the state-of-practice in which each function in the DAG runs in a separate VM and VM sizes are either right-sized to the functions' memory footprints (User-Min) or set to max VM size (User-Max).
2. **Faastlane [30]**: Here the entire DAG is executed within a single VM. Faastlane has a fallback strategy of using remote storage when a single VM is not large enough to execute all the functions

within one stage. In this case, the functions are divided into bundles but the bundle size is fixed to # CPU cores of the VM. The VM size is set to fit the most resource-demanding stage.
3. **Sonic** [35]: Reduces communication latency by selecting between three data passing methods: remote storage, direct passing, and local VM-storage passing. However, direct passing cannot be supported in AWS Lambda or GCP and we implement Sonic's remote storage and local VM-storage options only. Sonic performs no Bundling or any other skew mitigation technique. Further, it assumes function execution times are deterministic and hence does not use function execution time distributions.
4. **Photons [13]**: Colocates parallel invocations together to improve the memory utilization, not to meet latency or cost objectives. However, this colocation mitigates execution skew to some extent. Moreover, Photons colocates as many parallel invocations as possible as per the function's memory footprint. This can cause excessive bundling and can increase the E2E latency.
5. **Theoretical lowest latency**: This provides the theoretical (oracle), lowest latency for any serverless DAG. It runs all the functions in the serverless application within one VM, which can be arbitrarily large to accommodate the largest number of parallel workers in any stage.

5.2 Applications

The three applications[2] that we use in our evaluation are adopted from prior works and they show a diversity in structure (*i.e.*, depth and width), intermediate data volumes, and execution time skews.
(1) **Video Analytics**: Adopted from Pocket [29] and Sonic [35]. This application performs object detection and classification for frames in a video (Figure 1a). The first stage in the DAG, called `Split`, downloads the input video from remote storage (S3) and splits it into equal chunks of 2 seconds. Each chunk is then sent to `Extract` stage to extract a representative frame from the chunk. Each frame is then sent to `Classify` stage to detect and classify all objects in the frame. Finally, the objects are either sent to `Face-Detect` if the frame contains a person, or sent to `OCR` if the frame contains a car. We use 600 YouTube videos of lengths 1 minute, 5 minutes and 10 minutes. Of them, 50% are used for profiling, and the other 50%, for evaluation.
(2) **Approximate Singular Value Decomposition (SVD)**: This application estimates an approximate Singular Value Decomposition (SVD) for a large-scale input matrix, which is widely used in recommender systems [28, 51]. SVD has a time complexity of $O(n^3)$ and a space complexity of $O(n^2)$. Accordingly, several approximation techniques have been proposed to parallelize SVD computations and significantly reduce the runtime and memory footprints [25, 43, 48]. We perform SVD for the Netflix prize dataset [40] and use the Split-Merge approach [31]. The dataset has movie ratings for 17,770 movies, submitted by 480,189 users, and each rating is on a 5-star integer scale, from 1 to 5. The application has two stages. The first stage performs a matrix split operation to a user-defined number of sub-matrices. Each sub-matrix is then sent to an instance of RunSVD function to estimate its decompositions. Finally, all matrix decompositions are saved to remote storage. First, as a preprocessing step, we save the Netflix data in 128 equal blocks, divided by rows. This is the minimum degree of parallelism that we can run at without hitting the maximum memory limit in AWS Lambda (10 GB). The 128 blocks are then saved to remote storage. We set the number of split functions in the first stage to 128 (one invocation per block) and each invocation splits the block further into k sub-matrices ($k = 2$ in our case). Thus, we run 256 instances of the RunSVD function, in parallel, on the sub-matrices. With this degree of decomposition, we calculate the reconstruction error and it is small ($< 0.5\%$), which is considered acceptable for most SVD deployments.

[2]The code of the three applications is available from https://github.com/icanforce/WiseFuse-applications



(a) Video Analytics

(b) Approximate SVD

(c) ML Pipeline

(d) Approximate SVD on Google

Fig. 16: WiseFuse's comparison in latency and $ cost. % over the bars show WiseFuse's gains in E2E latency (1.5× configuration) over this particular scheme. We set the latency target to (1.5×, 2.5×, and 5×) of the best theoretical latency (total processing time using an arbitrarily large VM and zero communication latency).

(3) **ML Pipeline**: Adopted from Cirrus [8], this application trains a random forest (RF) prediction model for the MNIST handwritten characters [11], containing 810K images. The first function downloads the dataset from remote storage and performs Principal Component Analysis (PCA). In the second function, a hyper-parameter tuning stage is invoked, which includes 64 parallel lambdas, each executes with a unique set of hyper-parameters (# features, # trees, and tree max depth). In the final function, the accuracy of each model is estimated on a held-out dataset and the top 10 models are uploaded to remote storage. The LightGBM boosting framework was used for training the RF [32].

5.3 E2E Evaluation

Video Analytics: We compare the latency and cost of WiseFuse to the baselines using 300 test videos, which were not used in the profiling. We show the P95 latency and cost for 1K executions for each baseline in Figure 16a. WiseFuse and all baselines are executed in AWS Lambda, using Step Functions for DAG orchestration and Amazon S3 as the remote storage. We report the cumulative cost of 1K executions. We show the performance of three different settings of WiseFuse, corresponding to different latency objectives relative to the Theoretical Lowest Latency (1.5×, 2.5×, and 5×). This evaluates the ability of WiseFuse to be configured to meet different price-latency points. For the rest of this E2E evaluation, we refer to the performance of WiseFuse 1.5×.

Compared to the user-defined DAG baseline, WiseFuse achieves either 63% lower cost than User-Max or 61% lower P95 latency than User-Min (although it increases the cost somewhat (11%) over User-Min). This is because of WiseFuse's execution plan, which fuses the `Split` stage with the `Extract` stage to reduce the communication latency, also bundles parallel invocations of `Classify` together to mitigate computation skew. Moreover, by assigning the right resources to each set of bundled or fused functions, WiseFuse meets the latency objective with significantly lower cost.

Compared to Sonic, WiseFuse achieves 47% lower P95 latency. Recall that Sonic only considers fusing in-series functions to leverage data locality, and hence fuses the `Split` with the `Extract` stage together. However, it neither performs any Bundling nor considers the latency distribution

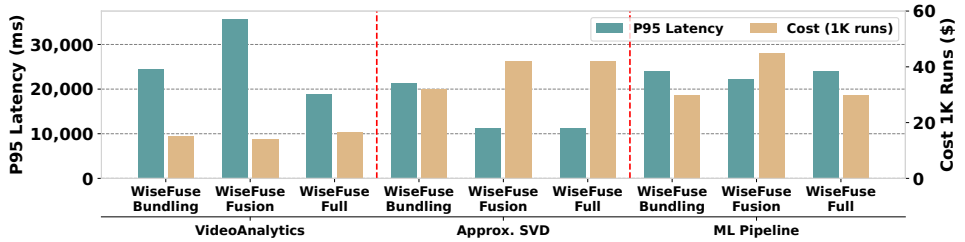


Fig. 17: Comparison between WiseFuse (-Full) and its two variants that do Bundling only, or Fusion only.

and hence assigns resources that minimize the *average* latency, in contrast to WiseFuse that assigns the right resources to meet the required latency percentile objective.

Considering Photons, we notice its latency is as high as User-Min and WiseFuse achieves a lower latency by 62%. Recall that Photons performs Bundling mainly to improve memory utilization. Therefore, it bundles as many parallel invocations as possible based on the functions' memory footprint. This causes Photons to bundle 16 parallel invocations of the `Classify` stage together in one VM. Although this maximizes the memory utilization, it assigns less than optimal resources to each worker and causes a significant increase in latency. WiseFuse uses its performance model to identify the best bundle size, which is 4 for this application.

We notice Faastlane is the closest baseline to WiseFuse, yet WiseFuse achieves 39% lower P95 latency. Faastlane (similar to WiseFuse) fuses the `Split` and the `Extract` stages together, but falls back to remote storage (S3) when executing the `Classify` stage. Moreover, it uses a fixed bundle size of 6 workers to match the 6 vCPUs that are provided by AWS Lambda's Max VM size. Additionally, Faastlane cannot mitigate the significant execution skew that exists among the parallel workers.

Approximate SVD: We show the latency and cost of WiseFuse versus baselines in Figure 16b. Compared to user-defined DAG, WiseFuse achieves lower latency than User-Min and User-Max by 61% and 81% respectively. Although User-Max assigns the maximum resources to each worker, it still suffers from the increased communication latency between the split and SVD stages. WiseFuse performs Fusion of these stages and reduces the communication latency significantly. WiseFuse achieves 50% lower latency than Faastlane, 68% lower latency than Photons, and 42% lower latency than Sonic. This again shows the importance of *jointly* performing both types of optimizations while assigning the cost-optimized VM sizes. In terms of cost, only Sonic (marginally) reduces the cost over WiseFuse (7%), while it (significantly) increases the latency (42%).

ML Pipeline: We show the evaluation results in Figure 16c. We notice two main differences in this application. First, using Min VM sizes (as done by User-Min) does not provide the lowest cost. In fact, it increases the cost by almost 2× compared to using the Max VM sizes. The reason is that the `ParamTune` function shows a superlinear improvement in the runtime when allocated additional resources. For example, the function's runtime reaches 306 sec when assigned a VM of size 1,240 MB (VM min size) and it becomes only 22.2 sec when the VM size is increased to 10,240 MB—an 8× increase in the allocated resources leads to a 14× decrease in the latency, and therefore WiseFuse provides lower cost than User-Min. Second, WiseFuse produces similar execution plans for latency targets of 1.5×, 2.5×, and 5× the best theoretical latency. This is again because exploring lower VM sizes or bundle-sizes for this application does not reduce the cost compared to the 1.5× case, and hence the optimizer converges to this particular plan as it meets all latency targets with reduced cost. Finally, WiseFuse meets the latency targets of 1.5×, 2.5×, and 5× the best theoretical latency for all three applications with a small error in the range of [0.3%,12%].

Comparison to exhaustive search: As a proxy for exhaustive search, we compare WiseFuse with grid search where we set grid search to start from the best Fusion strategy given by WiseFuse and then explore all bundle sizes between 1 and 16 (increments of 1) and all VM sizes between 1 GB and 10.24 GB (AWS Lambda's Max) (increments of 500 MB). This results in a total of 32K profiling runs for grid search (recollect we need at least 100 samples for each data point for creating a distribution) compared to 300 points for WiseFuse (which simply uses its performance model trained with these 300 points). We show the result in Figure 18. The execution plan found by grid search gives 1.1% lower latency for Video Analytics, 15% for Approx SVD, and 2.7% for ML Pipeline. However, grid search's plans use a smaller bundle sizes compared to WiseFuse and hence increase the cost by 28% for video analytics and 5% for Approx. SVD. Recall that the cost shown is only the cost of executing the DAG, and does not include the profiling cost, which as discussed above is much higher for grid search. Considering the lower profiling cost, and correspondingly lower incurred cost, of WiseFuse, this latency performance would be considered acceptable in many situations.

Compute and Communication Latency Scaling in GCP We look at the change in both compute and communication latency with varying memory sizes on Google Cloud Functions for the Approx SVD application in Figure 22. The general serverless notion of scaling all resources with respect to the memory allocated is generally true, except when going from 4GB to 8GB. Both these memory sizes have the same compute power (2.4 GHz) and thus the average compute latency doesn't change between these two. As for communication latency, we see a consistent decrease as the memory size increases. This points to the network bandwidth continuing to scale unlike in AWS Lambda where it saturates at 1792MB (1 core).

Ablation Study: To analyze the gains of Fusion and Bundling separately, we limit WiseFuse to perform Bundling only (called WiseFuse-Bundling) or Fusion only (called WiseFuse-Fusion). Both variants still select the best VM sizes for nodes in the DAG after transformation. We compare the performance of these two variants to our full solution (WiseFuse-Full) in Figure 17 for our three applications. For Video Analytics, we notice that WiseFuse-Full achieves lower latency than both variants. We also notice that WiseFuse-Bundling achieves lower latency than WiseFuse-Fusion, indicating that execution skew contributes more latency than data exchange for this application. For Approx SVD, we notice that WiseFuse-Full achieves a similar performance to WiseFuse-Fusion, whereas WiseFuse-Bundling has a higher latency but lower cost. The reason is that in this DAG, Fusion forces bundling of size 2, which WiseFuse-Full also found to be the best transformation. Finally, for ML Pipeline, WiseFuse-Full achieves similar performance to WiseFuse-Bundling as WiseFuse-Full found that all Fusion decisions were harmful to performance and hence performs Bundling only for this DAG.

In summary, WiseFuse outperforms all baselines and competing approaches and is able to provide an execution plan that reduces E2E latency and cost. Further, both Fusion and Bundling are required for achieving the gains, albeit these two contribute to different extents for different applications.

5.4 Microbenchmarks

We run microbenchmarks to evaluate the accuracy of WiseFuse's performance model, impact of Fusion, impact of Bundling, and impact of varying the input size on the E2E latency and cost.

5.4.1 Per-function Modeling Accuracy Here we show accuracy of our per-function performance model, described in Section 3.3. For each VM size $\in$ {Min, 1 GB, 1.8 GB, and 10.24 GB(Max)}, we collect profiling information for each function. For evaluation, we use 300 points for 3 test VM sizes. We use our performance model to estimate the CDFs for each function in our evaluation applications

Application	Function	P95 RMSE (Actual vs Estimated)			
		Download	*Process*	*Upload*	*Overall*
Video Analytics	Split	2.44%	2.89%	7.6%	3.61%
	Extract	0.44%	1.34%	1.8%	0.25%
	Classify	5.2%	1.6%	11.8%	2.3%
ML Pipeline	PCA	0.8%	1.7%	12%	4.4%
	ParamTune	16%	3%	18%	13%
	Combine	0.24%	0.33%	0.9%	0.4%
Approx SVD	SplitRows	3.8%	0.1%	15%	12%
	RunSVD	2.5%	7.7%	6%	2.7%
Average		**3.9%**	**2.3%**	**9.1%**	**4.8%**

Table 1: RMSE in latency estimates for each function in our evaluation applications.

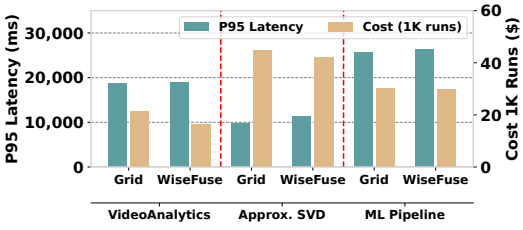


Fig. 18: Comparison between WiseFuse and Grid Search.

and compare the estimated P95 latency to the actual P95 latency. We show the results in Table 1. We notice that our performance model has very low error, in the range [0.25%, 13%] across all functions and all applications. Moreover, the upload and download operations have higher errors on average than the process operation. This is because of the high variability in data exchange latency that we empirically observe in production environments due to network bandwidth fluctuations. This stresses the benefit of our Fusion mechanism to reduce communication latency.

5.4.2 Estimating Latency After Fusion and Bundling In this section, we evaluate the accuracy of our performance model in estimating the latency after we perform either Fusion or Bundling to our video analytics application. For Fusion, we use a single linear chain of "Split→Extract→Classify" and we want to estimate the chain's E2E CDF when all three functions are fused together, thus evaluating our algorithm described in Section 3.3. As mentioned in Section 3.4, considering the correlation between the operations is essential to accurately estimate their combined CDF. Here we notice that "Classify-Process" and "Classify-Upload" have a high correlation of 0.7. This high correlation is because frames with a large number of objects take longer for identification, cropping, and uploading. We compare in Figure 19 the case where we neglect this correlation with the case where we consider it (as in WiseFuse). As in the figure, ignoring this correlation causes underestimation of the combined latency and increases the error to a maximum of 12.5%. However, WiseFuse, using the joint distributions to take this correlation into consideration, reduces the error to $\leq$ 3.4%. This shows the importance of our correlation-aware performance model to accurately estimate the impact of fusing functions together.

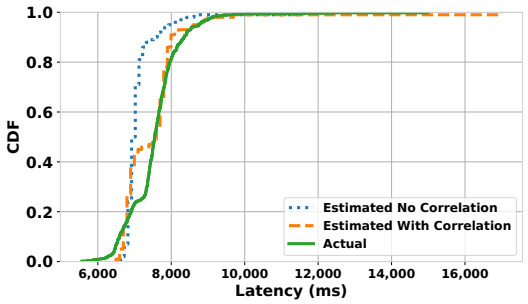


Fig. 19: Accuracy of WiseFuse's CDF estimation for a fused chain for our Video Analytics application. Without considering correlation, errors can be up to 12.5%. With correlation, errors come down to $\leq$ 3.4%.

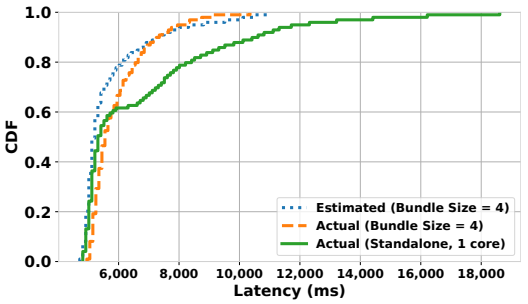


Fig. 20: Accuracy of WiseFuse's CDF estimation for a bundle of 4 `Classify` workers. WiseFuse achieves errors in the range of [-6.4%,7%] compared to the actual CDF.

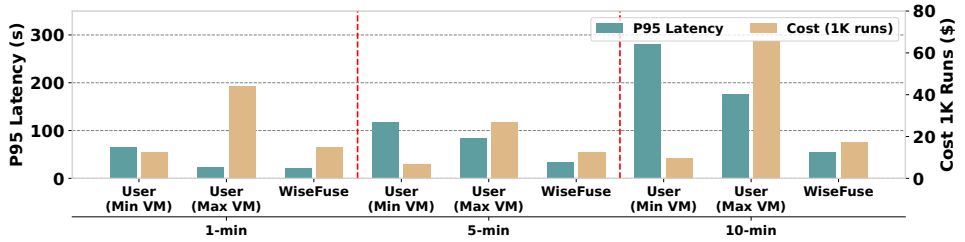


Fig. 21: Video Analytics: Comparison between WiseFuse's optimized plan and user-defined DAG using min and max VM sizes. We show how the latency and cost are impacted by varying input sizes.

To evaluate the benefit of Bundling and our ability to predict the latency CDF, we consider the Bundling strategy WiseFuse chooses for the `Classify` stage. The `Classify` stage is a ripe target as it experiences the highest computation skew in the DAG. WiseFuse selects a Bundle size of 4. We compare WiseFuse's estimated CDF using the approach in Section 3.5 to the actual CDF of the Bundle in Figure 20. We also show the standalone (*i.e.*, no bundling) CDF before Bundling using a VM size of 1,792 MB, which comes with a full vCPU core. We notice that with Bundling, the combined CDF improves the tail latency significantly compared to standalone CDF as P95 latency decreases by 27%. This highlights the gain due to skew mitigation through Bundling. Moreover, we notice that our estimated CDF for the bundle is very close to the actual, with errors in the range of ± 7% across all percentiles. In summary, WiseFuse's performance model shows high accuracy in estimating the impact of Fusion and Bundling on the DAG's latency, which is essential in selecting the best execution plan for the DAG.

5.4.3 Varying Input Data Size Here we evaluate WiseFuse's performance and cost with different input sizes. We execute the Video Analytics application with varying input video lengths of 1, 5, and 10 minutes, using 300 videos for each length. We notice that changing the input size impacts the following parameters: (1) Upload, process, and download CDFs for `Split` stage. (2) Fanout degree for `Extract` and `Classify` stages. To evaluate the accuracy of our polynomial regression, we use two sizes as inputs and estimate for the third size. Our estimated CDFs show an error of $\leq 9.4\%$ across all percentiles. Afterward, we compare WiseFuse's execution plan to the user-defined DAG with both min- and max-VM sizes in Figure 21. We notice the following: first, with 1-min video inputs, WiseFuse and User-Max achieve similar latency but with 68% lower cost for WiseFuse. The reason is that WiseFuse mitigates execution skew through Bundling. However, user-Max, by allocating the max VM size to all workers, also mitigates execution skew, albeit, with higher cost. In contrast, with 5-min and 10-min video clips, WiseFuse achieves lower latency than user-Max by 61% and 69% respectively as WiseFuse performs Fusion between `Split` and `Extract`. Hence, WiseFuse reduces the data exchange time, which increases as we increase the input video length. In summary, by estimating the impact of varying the input size on the upload, process, and download CDFs, WiseFuse finds optimized execution plans for different input sizes.

5.5 WiseFuse on Google Cloud

Here we show the results of WiseFuse and baselines on Google Cloud using the Approx SVD application (Figure 16d). This is to verify if the benefits of WiseFuse are tied to any esoteric FaaS platform features (AWS Lambda) or if they generalize. We use the same implementation of WiseFuse as on AWS Lambda, with S3 being substituted by Google Cloud Storage. Significantly, the GCP equivalent for AWS Step Functions, Google Workflows, has very limited support for invoking serverless functions directly and this feature is still in beta [23]. The recommended approach is to

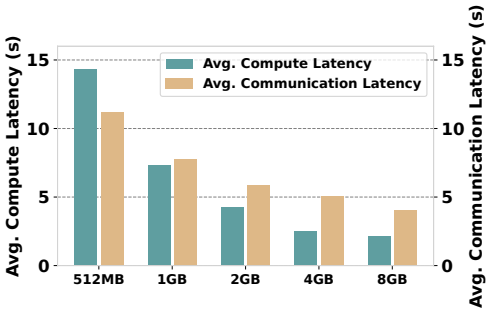


Fig. 22: Average latency for compute and communication on Google Cloud Functions with varying memory size. The network bandwidth continues to scale unlike in AWS Lambda.

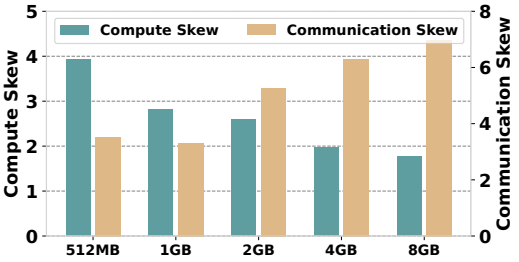


Fig. 23: Skew for compute and communication on Google Cloud Functions. While network bandwidth scales with memory size, the communication skew grows.

invoke functions using HTTP calls but this is infeasible for applications with large fanouts (*e.g.*, Approx SVD) since we hit the memory limit of 64KB [24] easily when counting the sizes of all the HTTP headers for all responses. Thus, as a workaround, we use Google Cloud Pub/Sub [21] topics where one stage publishes to a given topic upon completion, and the subsequent stage, is triggered by an event from Pub/Sub. We use the Approx SVD application in preference to the other two since this has a much higher fanout and thus stresses our workaround for GCP Workflows.

Our benchmarks show that in Google Cloud Functions, network bandwidth continues to scale with VM sizes (unlike in AWS Lambda) as shown by the decrease in communication latency with memory sizes (Figure 9). However, we still see a high data communication skew — up to 6.8× between parallel workers as seen in Figure 23 — which causes significant increase in the E2E latency and cost. This skew is mitigated by WiseFuse's Fusion since that eliminates data communication between stages. Figure 16d shows that WiseFuse achieves lower latency than all other baselines similar to the evaluation on AWS Lambda. The main difference between the two platforms is the amount of compute power available. In GCP, the maximum memory size of 8 GB still only provides the same compute power (2 cores) as a 4 GB VM [20]. In contrast, in AWS Lambda, we get up to 6 cores with the 10.24 GB VM. This means that Bundling is not as effective on Google Cloud. Consequently, baselines that perform only Fusion (Sonic and Faastlane) are much closer in performance to WiseFuse.

6 Related Work

Characterization and modeling of serverless workloads. A related study [45] characterizes FaaS workload of Azure Functions, focusing on individual functions rather than on DAGs. We draw a number of comparisons between our work and this study, and highlight their implications on performance and cost (Section 2). Atoll [47] also characterizes popular apps from AWS Serverless Application Repository (SAR) but does not shed light on the DAG structures. Microservice architecture is a more general computing model than serverless computing. A microservice is similar to a serverless function, but being stateful, can be long running. The workload characterization of microservices at Alibaba [33] shows that the distribution of microservices execution time is heavy-tailed. In contrast, serverless functions are shorter and providers impose timeouts on function execution times. Less directly related but an inspiration for our work is analysis of workloads on the cloud that was used to drive capacity planning and task scheduling decisions [37, 38]. A few prior studies have targeted predicting the execution time of serverless functions. For example,

Sizeless [14] predicts (a point estimate) and optimizes resources for a single serverless function by building regression models from a host of synthetic functions. Another study [15] observes a variance in execution time in serverless environments, and hence, applies mixture density networks to predict the distribution of the function cost. However, its Monte-Carlo simulation mechanism is sample inefficient. WiseFuse uses a more direct method by applying statistical operations to combine the distributions of individual functions, and thus, to infer the E2E latency distribution.

Optimizing performance of serverless DAGs. Several recent proposals aim at making serverless executions more efficient, in terms of both latency and cost. Sonic [35] proposed a hybrid and dynamic approach to pick between three different intermediate data passing methods in serverless DAGs. The solution focuses only on reducing data exchange latency, and hence cannot mitigate execution skew or right size the resources to meet latency targets. Moreover, Sonic makes a major simplification, assuming that function execution times are deterministic and thus ignores runtime variances. The idea of bundling multiple parallel invocations to mitigate execution skew was proposed in Orion [36]. However, the solution does not consider Fusion or any similar technique to reduce communication latency between consecutive stages in the DAG. Moreover, Orion finds the best bundle size through trial and error rather than leveraging the performance model as done by WiseFuse. Llama [44] considers optimizing the latency and cost for serverless video analytics pipelines. Llama is application specific and is designed for tuning video analytics parameters (including content-dependent parameters) such as sampling rate or batch size. Finally, Caerus [52] targets optimizing the latency and cost of two-stage Map-Reduce jobs by deciding when to start the execution of each mapper/reducer function. The solution applies to functions that can start execution when their input data is partially available, and hence, solves an orthogonal problem. Compared to these solutions, WiseFuse performs both Fusion and Bundling and allocates resources to generate a cost optimized plan that meets a user-given latency target. A complementary line of work provides efficient scheduling for serverless DAGs. Wukong [9] provides decentralized and parallel scheduling distributed across Lambda executors. It leads to serverless DAGs using the network I/O highly efficiently. Xanadu [10] and Kraken [7] tackle the problem of cascading cold starts in a dynamic DAG. Overall, no prior work in this category considers execution time distributions or combines such distributions for optimizing E2E latency or cost. Schedulers on the cloud like Apache Spark [17] and Dask [26], which deal with stateful services, have an orthogonal set of concerns, such as, state migration and cluster utilization [19, 53].

Optimizing communication latency in serverless workflows. Besides Sonic, recent work also identifies communication latency as a major performance degrading factor in serverless workflows. Pocket [29] and Locus [42] show that current options for remote storage are either slow disk-based (*e.g.*, S3) or expensive memory-based (*e.g.*, ElastiCache Redis). Therefore, Pocket combines different storage media (*e.g.*, DRAM, SSD, NVMe) that users can choose to conform to their application needs. SAND [1] (similar to Faastlane [30]) avoids network communication altogether by forcing all functions that belong to the same DAG to execute on the same container, thus leveraging data locality between them. While all these systems try to reduce communication latency between serverless functions, they either take an extreme approach of forcing *all* functions in a DAG to execute in the same container (sacrificing scheduling flexibility and parallelism), or they rely on users to select storage media for their applications.

Optimizing resources in serverless workflows. Photons [13] uses a technique similar to bundling restricted to DAGs with one fanout stage while focusing on security issues and reduction in the memory footprint. In contrast, WiseFuse optimizes more general DAGs with several fanout stages and cascade functions, rightsizing resources to functions and function-bundles. The concept of optimizing for a target latency in a serverless DAG is seen in Atoll [47], proactively initializing VMs to schedule function requests onto them. Proactive approaches run into the challenge of

predicting when to start new containers with specific dependencies, exacerbated by unpredictable request arrival rates. Moreover, this approach is complementary to WiseFuse's DAG restructuring. There is significant work in improving serverless runtimes and these can benefit us by reducing execution times of individual functions. Such approaches include reducing latencies by low-level system optimizations [27, 41, 46], keeping containers alive [18, 45], or quick checkpoint save and restore [12].

7 Discussion

Updating the profiled information: In order to keep WiseFuse's performance model current, we need to monitor the workload to detect changes, and update the profiled characteristics. WiseFuse achieves this by applying the following steps. WiseFuse models the latency of a function as the contribution of three components: download, process, and upload, and maintains a distribution for each component. The serverless provider logs the initialization time and total execution time for each function invocation, for billing, and for continuous monitoring (*e.g.*, AWS CloudWatch and Azure Cloud Monitor). Since WiseFuse already captures per-function distributions and estimates the DAG E2E latency distribution, it can detect changes to the workload characteristics or to warm-up latencies by comparing its profiled distributions to the online monitored distributions. Reprofiling is triggered when comparing statistics of the captured distribution to those of the monitored distribution indicates a change as follows: (1) when a change in a per-function statistic exceeds a threshold (10% error in P50, or 15% in P95), (2) when a change in per-DAG latency statistic or the user-specified target latency percentile exceeds similar thresholds, and (3) upon changes to the function binary or the DAG structure. When such a change is detected, WiseFuse creates a new distribution using N data samples. In many cases, the N data samples would be already available from the monitored distribution. By default $N = 300$, which we find empirically sufficient to model P95 latency with low errors. This kind of profiling for tail latencies is conceptually similar to that in OptimusCloud [34].

Larger serverless functions: WiseFuse's objective is to transform the user-defined DAG to an optimized DAG. This transformation includes colocating multiple parallel invocations together (*i.e.*, Bundling), and/or fusing multiple stages together (*i.e.*, Fusion). Accordingly, each node in the optimized DAG serves as a new function and hence becomes the new basic unit of scaling and scheduling. Thus, the new nodes in the transformed DAG will, expectedly, have higher memory footprints (*e.g.*, due to Bundling) and/or longer execution times (*e.g.*, due to Fusion) compared to nodes in the user-defined DAG. However, this is not a concern as cloud providers are increasingly supporting larger VMs and relatively long execution times (*e.g.*, AWS Lambda supports up to 10 GB of memory and 15 min of execution time [4]).

Security Concerns for Fusion and Bundling: WiseFuse only bundles or fuses functions of the same DAG that belong to the same user. However, there could be security sensitive functions that should not be bundled or fused. For example, assume function A has sensitive data and function B should not have access to that data even when both functions belong to the same DAG of the same user. In such a case, the user provides this constraint to WiseFuse so that the security-sensitive function is excluded from WiseFuse's search space.

8 Conclusion

We characterize production workloads of serverless DAGs at a major FaaS provider and see that serverless DAGs are increasing in popularity. We analyze the execution parameters (function execution time, skew among the parallel invocations of a function, and data transfer between functions). From our analysis, we identify two major performance bottlenecks in serverless DAGs: (1) communication latency between in-series functions (due to infrastructure reasons) (2) computation

skew among in-parallel function invocations (due to data skews). We present WiseFuse that addresses these challenges through two operations — Fusion (of in-series stages) and Bundling (of in-parallel function invocations), addressing communication and computation skew, respectively. Concretely, WiseFuse uses Fusion and Bundling operations to derive an optimized execution plan that meets a user-defined latency SLA with low cost. Through experimental evaluation and comparisons with baselines and competing approaches (Faastlane, Sonic, and Photons), and using three applications, we show that WiseFuse is superior. Specifically, for an ML pipeline, WiseFuse achieves a P95 latency that is 67% lower than Photons, 39% lower than Faastlane, and 90% lower than Sonic, without increasing the $ cost. Our evaluation on AWS Lambda and GCP Functions also highlights some hitherto unknown platform-specific nuances of serverless execution.

Acknowledgments

This material is based in part upon work supported by the National Science Foundation under Grant Numbers CCF-1919197, CNS-2016704, CNS-2038986, CNS-2038566, CNS-2146449 (NSF CAREER award), NIH Grant R01AI123037, and funding from Microsoft Research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors. The authors thank the reviewers for their enthusiastic comments and the shepherd, Tianyin Xu, for his insightful feedback.

References

[1] Akkus, I. E., Chen, R., Rimac, I., Stein, M., Satzke, K., Beck, A., Aditya, P., and Hilt, V. SAND: Towards High-Performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 923–935.

[2] Amazon. AWS Step Functions Documentation. https://docs.aws.amazon.com/step-functions/index.html, 2020.

[3] Amazon. Configuring Lambda function options. https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html, 2022.

[4] Amazon. Lambda quotas. https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html, 2022.

[5] Azure. Durable functions overview. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp, 2020.

[6] Azure Serverless DAG traces (public dataset). https://github.com/Azure/AzurePublicDataset, 2022.

[7] Bhasi, V. M., Gunasekaran, J. R., Thinakaran, P., Mishra, C. S., Kandemir, M. T., and Das, C. Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms. In *Proceedings of the ACM Symposium on Cloud Computing* (2021), pp. 153–167.

[8] Carreira, J., Fonseca, P., Tumanov, A., Zhang, A., and Katz, R. Cirrus: A Serverless Framework for End-to-End ML Workflows. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2019), SoCC '19, Association for Computing Machinery, p. 13–24.

[9] Carver, B., Zhang, J., Wang, A., Anwar, A., Wu, P., and Cheng, Y. Wukong: A scalable and locality-enhanced framework for serverless parallel computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (2020), pp. 1–15.

[10] Daw, N., Bellur, U., and Kulkarni, P. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st International Middleware Conference* (2020), pp. 356–370.

[11] Deng, L. The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]. *IEEE Signal Processing Magazine 29*, 6 (2012), 141–142.

[12] Du, D., Yu, T., Xia, Y., Zang, B., Yan, G., Qin, C., Wu, Q., and Chen, H. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2020), pp. 467–481.

[13] Dukic, V., Bruno, R., Singla, A., and Alonso, G. Photons: Lambdas on a Diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (New York, NY, USA, 2020), SoCC '20, Association for Computing Machinery, p. 45–59.

[14] Eismann, S., Bui, L., Grohmann, J., Abad, C., Herbst, N., and Kounev, S. *Sizeless: Predicting the Optimal Size of Serverless Functions.* Association for Computing Machinery, New York, NY, USA, 2021, p. 248–259.

[15] Eismann, S., Grohmann, J., van Eyk, E., Herbst, N., and Kounev, S. Predicting the costs of serverless workflows. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering* (2020), pp. 265–276.

[16] Elgamal, T. Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)* (2018), pp. 300–312.

[17] Foundation, A. Job scheduling - spark 3.2.1 documentation. https://spark.apache.org/docs/latest/job-scheduling.html, Last retrieved: Jan, 2022.

[18] Fuerst, A., and Sharma, P. Faascache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2021), pp. 386–400.

[19] Fuerst, C., Schmid, S., Suresh, L., and Costa, P. Kraken: Towards elastic performance guarantees in multi-tenant data centers. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2015), pp. 433–434.

[20] Google. Cloud Functions pricing. https://cloud.google.com/functions/pricing, 2022.

[21] Google. Cloud Pub/Sub. https://cloud.google.com/pubsub, 2022.

[22] Google. Google Cloud: Configuring Memory Limits. https://cloud.google.com/run/docs/configuring/memory-limits, 2022.

[23] Google. Google Cloud Function Quotas. https://cloud.google.com/functions/quotas#footnote, 2022.

[24] Google. Quota Limits for Google Cloud Functions. https://cloud.google.com/workflows/quotas, 2022.

[25] Hastie, T., Mazumder, R., Lee, J. D., and Zadeh, R. Matrix completion and low-rank svd via fast alternating least squares. *The Journal of Machine Learning Research 16*, 1 (2015), 3367–3402.

[26] Inc., A. Scheduling - dask documentation. https://docs.dask.org/en/stable/scheduling.html, Last retrieved: Jan, 2022.

[27] Jia, Z., and Witchel, E. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2021), pp. 152–166.

[28] Kalantzis, V., Kollias, G., Ubaru, S., Nikolakopoulos, A. N., Horesh, L., and Clarkson, K. Projection techniques to update the truncated SVD of evolving matrices with applications. In *Proceedings of the 38th International Conference on Machine Learning* (18–24 Jul 2021), M. Meila and T. Zhang, Eds., vol. 139 of *Proceedings of Machine Learning Research*, PMLR, pp. 5236–5246.

[29] Klimovic, A., Wang, Y., Stuedi, P., Trivedi, A., Pfefferle, J., and Kozyrakis, C. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, Oct. 2018), USENIX Association, pp. 427–444.

[30] Kotni, S., Nayak, A., Ganapathy, V., and Basu, A. Faastlane: Accelerating Function-as-a-Service workflows. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (July 2021), USENIX Association, pp. 805–820.

[31] Liang, F., Shi, R., and Mo, Q. A split-and-merge approach for singular value decomposition of large-scale matrices. *Statistics and its interface 9*, 4 (2016), 453.

[32] LightGBM. Lightgbm python-package. https://lightgbm.readthedocs.io/en/latest/Python-Intro.html, 2022.

[33] Luo, S., Xu, H., Lu, C., Ye, K., Xu, G., Zhang, L., Ding, Y., He, J., and Xu, C. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing* (2021), pp. 412–426.

[34] Mahgoub, A., Medoff, A. M., Kumar, R., Mitra, S., Klimovic, A., Chaterji, S., and Bagchi, S. {OPTIMUSCLOUD}: Heterogeneous configuration optimization for distributed databases in the cloud. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)* (2020), pp. 189–203.

[35] Mahgoub, A., Shankar, K., Mitra, S., Klimovic, A., Chaterji, S., and Bagchi, S. SONIC: Application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (July 2021), USENIX Association, pp. 285–301.

[36] Mahgoub, A., Yi, E. B., Shankar, K., Chaterji, S., Elnikety, S., and Bagchi, S. ORION and the Three Rights: Sizing, co-location, and prewarming for serverless dags. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 1–14.

[37] Mishra, A. K., Hellerstein, J. L., Cirne, W., and Das, C. R. Towards characterizing cloud backend workloads: insights from google compute clusters. *ACM SIGMETRICS Performance Evaluation Review 37*, 4 (2010), 34–41.

[38] Moreno, I. S., Garraghan, P., Townend, P., and Xu, J. An approach for characterizing workloads in google cloud to derive realistic resource utilization models. In *2013 IEEE Seventh International Symposium on Service-Oriented System Engineering* (2013), IEEE, pp. 49–60.

[39] Müller, I., Marroquín, R., and Alonso, G. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020), pp. 115–130.

[40] Netflix. Netflix prize data. https://www.kaggle.com/netflix-inc/netflix-prize-data, 2021.

[41] Oakes, E., Yang, L., Zhou, D., Houck, K., Harter, T., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. {SOCK}: Rapid task provisioning with {Serverless-Optimized} containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018), pp. 57–70.

[42] Pu, Q., Venkataraman, S., and Stoica, I. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, Feb. 2019), USENIX Association, pp. 193–206.

[43] Ren, X., and Ramanan, D. Histograms of sparse codes for object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2013), pp. 3246–3253.

[44] Romero, F., Zhao, M., Yadwadkar, N. J., and Kozyrakis, C. *Llama: A Heterogeneous & Serverless Framework for Auto-Tuning Video Analytics Pipelines.* Association for Computing Machinery, New York, NY, USA, 2021, p. 1–17.

[45] Shahrad, M., Fonseca, R., Goiri, Í., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M., and Bianchini, R. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (July 2020), USENIX Association, pp. 205–218.

[46] Shillaker, S., and Pietzuch, P. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (2020), pp. 419–433.

[47] Singhvi, A., Balasubramanian, A., Houck, K., Shaikh, M. D., Venkataraman, S., and Akella, A. Atoll: A scalable low-latency serverless platform. In *Proceedings of the ACM Symposium on Cloud Computing* (2021), pp. 138–152.

[48] Song, Y., Sebe, N., and Wang, W. Why approximate matrix square root outperforms accurate svd in global covariance pooling? In *Proceedings of the IEEE/CVF International Conference on Computer Vision* (2021), pp. 1115–1123.

[49] Wang, L., Li, M., Zhang, Y., Ristenpart, T., and Swift, M. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 133–146.

[50] Yi, X., Park, D., Chen, Y., and Caramanis, C. Fast algorithms for robust pca via gradient descent. *Advances in neural information processing systems 29* (2016).

[51] YongchangWang, and Zhu, L. Research and implementation of SVD in machine learning. In *2017 IEEE/ACIS 16th International Conference on Computer and Information Science (ICIS)* (2017), pp. 471–475.

[52] Zhang, H., Tang, Y., Khandelwal, A., Chen, J., and Stoica, I. Caerus: NIMBLE task scheduling for serverless analytics. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)* (Apr. 2021), USENIX Association, pp. 653–669.

[53] Zheng, L., Joe-Wong, C., Brinton, C. G., Tan, C. W., Ha, S., and Chiang, M. On the viability of a cloud virtual service provider. *ACM SIGMETRICS Performance Evaluation Review 44*, 1 (2016), 235–248.

Received February 2022; revised March 2022; accepted April 2022