ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs

Ashraf Mahgoub and Edgardo Barsallo Yi, *Purdue University;*
Karthick Shankar, *Carnegie Mellon University;* Sameh Elnikety, *Microsoft Research;*
Somali Chaterji and Saurabh Bagchi, *Purdue University*

https://www.usenix.org/conference/osdi22/presentation/mahgoub

This paper is included in the Proceedings of the
16th USENIX Symposium on Operating Systems
Design and Implementation.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-28-1

ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs

Ashraf Mahgoub
Purdue University

Edgardo Barsallo Yi
Purdue University

Karthick Shankar
Carnegie Mellon University

Sameh Elnikety
Microsoft Research

Somali Chaterji
Purdue University

Saurabh Bagchi
Purdue University

Abstract

Serverless applications represented as DAGs have been growing in popularity. For many of these applications, it would be useful to estimate the end-to-end (E2E) latency and to allocate resources to individual functions so as to meet probabilistic guarantees for the E2E latency. This goal has not been met till now due to three fundamental challenges. The first is the high variability and correlation in the execution time of individual functions, the second is the skew in execution times of the parallel invocations, and the third is the incidence of cold starts. In this paper, we introduce ORION to achieve this goal. We first analyze traces from a production FaaS infrastructure to identify three characteristics of serverless DAGs. We use these to motivate and design three features. The first is a performance model that accounts for runtime variabilities and dependencies among functions in a DAG. The second is a method for co-locating multiple parallel invocations within a single VM thus mitigating content-based skew among these invocations. The third is a method for pre-warming VMs for subsequent functions in a DAG with the right look-ahead time. We integrate these three innovations and evaluate ORION on AWS Lambda with three serverless DAG applications. Our evaluation shows that compared to three competing approaches, ORION achieves up to 90% lower P95 latency without increasing $ cost, or up to 53% lower $ cost without increasing P95 latency.

1 Introduction

Serverless computing (*a.k.a.*, FaaS) has emerged as an attractive model for running cloud software for both providers and tenants. Recently, serverless environments are becoming increasingly popular for video processing [12, 58], machine learning [18, 55], and linear algebra applications [32, 48]. The requirements of these applications can vary from latency-strict (*e.g.*, Video Analytics for Amber Alert responders [61]) to latency-tolerant but cost-sensitive (*e.g.*, Training ML models [28]). Accurate latency estimation is essential to meet the requirements for both, as the cost in FaaS platforms is based on resource usage and runtime. The workflow of these serverless pipelines is usually represented as a directed acyclic graph (DAG) in which nodes represent serverless functions and edges represent data flow dependencies between them.

Serverless platforms experience high performance variability [4, 27, 35, 40, 42, 56] due to three primary reasons: First, some function invocations have cold starts. Second, there is skew in the execution time of various functions because of different content characteristics that the functions operate on. Third, there exists skew in the execution time due to variability in infrastructure resources (*e.g.*, network bandwidth for an allocated VM). Because of this variance in performance, predicting the mean (or median) execution time of individual functions is not sufficient to meet percentile-specific latency requirements (*e.g.*, P95) for serverless DAGs. Rather, a distribution-aware modeling technique is essential to capture this variability and provide accurate latency SLOs.

Key Idea. We propose ORION, a novel technique for performance modeling of serverless DAGs to estimate the end-to-end (E2E) execution time (synonymously, E2E latency). We leverage this model to enable system optimizations such as allocating resources to each function to reduce E2E latency while keeping $ costs low and utilization high. The different components of ORION are shown in Figure 1. We derive insights about serverless DAGs from analysis of production traces at *Azure Durable Functions* [13]. This analysis drives our performance model and the design features.

First, we observe the inherent performance variability in serverless DAGs and therefore represent the latency of a single function, as well as that of the entire DAG, as a distribution rather than a single value. For example, Figure 5 shows the variance in execution times for the top-5 most frequently invoked DAG-based applications. The execution times of invocations of the *same* DAG vary significantly, and the P95 latency is 80X of the P25 latency, averaged over the 5 applications. Thus, our performance model profiles the latency distribution for each function in the DAG and builds a performance model to capture the impact of varying the resource allocation to that function on its latency distribution. Afterward, we estimate the DAG E2E latency distribution by applying a se-

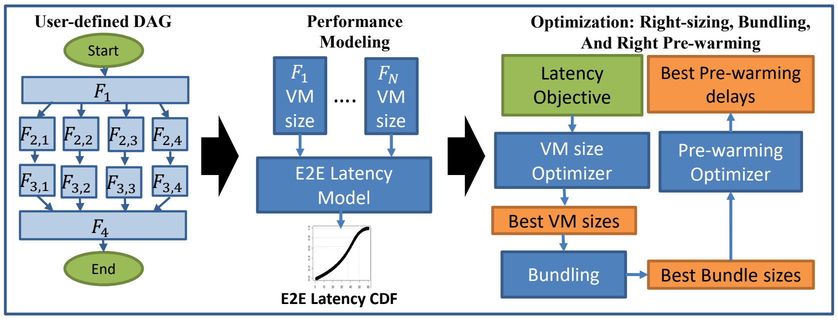


Figure 1: ORION *Overview.* ORION *profiles the DAG, estimates E2E latency CDF using CONV and MAX, and performs three system optimizations — right-sizing, bundling, and right pre-warming.*

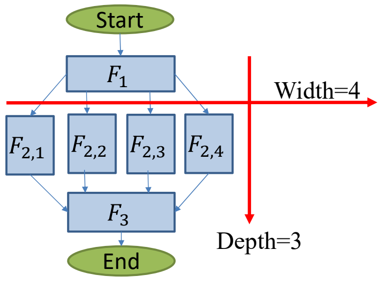


Figure 2: Illustration of DAG Depth (i.e., *number of in-series stages) and Width* (i.e., *Maximum fanout degree).*

ries of two statistical combination operations, `convolution` and `max`, for in-series and in-parallel functions, respectively. Moreover, we observe it is essential to consider the correlation between the workers across stages and within the same stage to accurately estimate the joint distributions. Our performance model does not require expensive profiling, as is needed for the leading technique, Bayesian Optimization [3, 5, 15].

We then use our performance model to design three optimizations for serverless DAGs. (1) **Right-sizing**: Finding the best resource configurations for each function to meet an E2E latency objective (*e.g.*, 95-th percentile latency < X sec) with the minimum cost. (2) **Bundling**: Identifying stages where co-locating multiple parallel instances of a function together to be executed on a single VM will be beneficial. The benefit arises when there is computation skew among the parallel workers caused by different content inputs. (3) **Right pre-warming**: The VMs to execute the functions in the DAG are pre-warmed just right, ahead of time, so that cold starts can be avoided while keeping provider-side utilization of resources high. With these three optimizations, ORION accurately meets latency SLOs while reducing execution cost (Figure 3).

ORION can be deployed by either the cloud service provider or by the end consumer. For the former, the use case is to provision its resources better to meet client SLOs. For the latter, the driving force is the appropriate resource provisioning to minimize E2E latency while minimizing execution cost.

Evaluation and Insights. We evaluate ORION on three serverless applications on AWS Lambda: two variants of Video Analytics [35], an ML Pipeline [17], and an NLP Chatbot [44] application. Our evaluation, comparing to three approaches: Best-Memory [2, 59], Speculative-Execution [30], and CherryPick [5], shows that the benefits of ORION persist across the different applications with different DAG structures, skews in execution times, and invocation frequencies. Our evaluation brings out the following insights:

(1) Latency correlation across stages *and* within a stage is important (Tables 1 and 2). Even when correlations are weak, not taking them into account can introduce significant error in the latency estimations. Further, to make the solution scalable, we have to compute the E2E latency estimation using just the right degree of correlation.

(2) Selecting the best VM sizes for serverless functions in a DAG is challenging (Table 3). This is because different resources in a VM scale up differently with their size. For example, for AWS-λ, CPU cores go from fractional to a maximum of six, network bandwidth only increases till a level, and disk capacity stays constant.

(3) Bundling multiple parallel instances of a function within a single VM helps when there is content skew and the functions are scalable, *i.e.*, they can make use of additional resources (Figure 16). Here also, the degree of bundling has to be carefully determined so as not to cause resource contention.

(4) Using the DAG structure and the function latency model, we can estimate the right time to pre-warm VMs and thus mitigate cold starts (Figure 15). With pre-warming, lower P95 latency is achieved with higher resource utilization.

In summary, the main contributions of ORION are the following: (1) Workload characterization for serverless DAGs seen by *Azure Durable Functions*. (2) A performance model for E2E latency of serverless DAGs; (3) A method for assigning the right resources for serverless DAGs to meet the E2E latency requirements within a reduced $ cost; (4) An application-independent way to bundle multiple function invocations to mitigate skews. (5) A method for deciding when to start pre-warming VMs for functions in a serverless DAG to minimize initialization latency while providing acceptable resource utilization.

ORION is open sourced and we release its code, the workload characterization data, and the evaluation applications at: https://github.com/icanforce/Orion-OSDI22

2 Motivation

2.1 Workload Characterization

Definitions. A DAG is a chain of two or more serverless function stages that execute in-series. A stage consists of one or more parallel invocations (a.k.a. instances) of the same serverless function. DAG *depth* is the number of stages in the DAG. DAG *width* is the max number of parallel worker functions (a.k.a. fanout degree) across all stages in the DAG.

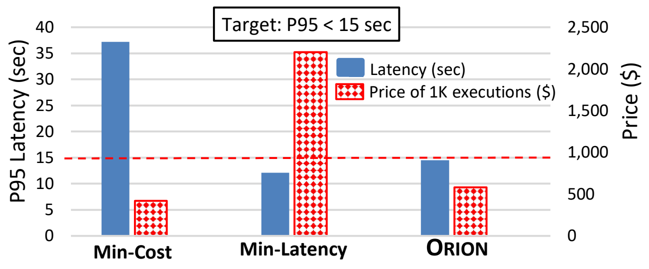


Figure 3: ORION *improves both latency and cost of Video Analytics DAG. Executions with min VM size* (i.e., *Min-Cost) and max VM size* (i.e., *Min-Latency) are for reference, and all latencies are for warm executions.*

A DAG with *width* = 1 means it is a chain of sequential function invocations, whereas a DAG with *width* > 1 means it has at least one parallel stage. We show an illustration of DAG depth and width in Figure 2. Finally, we define skew in a parallel stage as the ratio of the execution times of the slowest to the fastest worker function.

Analysis. In this section, we characterize the workload of serverless DAGs from *Azure Durable Functions*. We collect a subset of the logs of DAG executions from six datacenters — three located in the US, two in Europe, and one in Asia from 10/19/21 to 10/25/21 (1 week). The workload we analyze includes 20M-30M DAG executions/day. From our characterization, we draw the following conclusions, which in turn motivate various design decisions of ORION.

(1) DAG Structure and Execution Time. We study the depth and width of serverless DAGs and their distributions in Figure 4. We account for the DAG invocation frequencies — if a DAG is invoked *N* times, its width and depth are included *N* times in the CDFs. First, we notice that DAG depth is low, with a median of 3 stages and a P95 of 8 stages. Second, although 65% of the DAGs are linear chains (no fanout), the width can grow to as high as 37 in the 95th percentile. We also study the execution time of DAGs and find that they can range from 10 ms to 112 min, with a median of 3.7 sec and a mean (weighted by invocation frequencies) of 48 sec. This motivates the need for considering DAG structure while minimizing the E2E latency.

(2) DAG Invocation Frequency and Relation to Cold Starts. Figure 7 shows the frequency of invocations/day for each DAG and the corresponding percentages of cold starts. We notice that the frequency of invocations is heavily skewed, with the top-5 most frequent DAGs accounting for 46% of all invocations. Thus, the optimized executions of these frequently invoked DAGs result in higher cost savings. We also notice that the percentage of cold starts decreases with higher invocation frequency. For example, DAGs invoked $\geq$ 100 times/day have very low percentages of cold start with a median of 0.35%. However, most of the DAGs are rarely invoked: 80% of the DAGs have an invocation frequency of < 100 times/day, and these experience a high percentage of cold starts with a median of 50%. This shows an increase in the proportion of infrequent serverless applications (DAG-based in our case) compared to a prior study [47], which showed that 55% of the serverless applications are invoked less than 100 times/day.

Hence, using keep-alive policies (as done by most major FaaS platforms) for those DAGs will not be sufficient and pre-warming becomes essential to mitigate cold starts. Even for the DAGs that are not the most frequently invoked, keeping E2E latency low is a desirable feature.

(3) Variance in DAG Execution Time. Figure 5 shows a boxplot for the execution time of the top-5 most frequently invoked DAGs (which contribute 46% of all invocations). We notice that the variance in execution times across different invocations of the same DAG is high. For example, the P95 latency is 80X the P25 latency, averaged over the five DAGs. We also notice that the *distribution of execution times can be heavily skewed.* For example, P50 is not centered between MIN and MAX, or between P25 and P75. Hence, it is essential to represent E2E latency of serverless DAGs as a distribution when modeling their performance to capture this variability.

(4) Degree of Skew. Figure 6 shows the skew distribution for parallel stages for different ranges of DAG widths. We notice that skew among parallel worker functions is at least 2× for 98.2% of the DAGs and increases as the width increases. This motivates the need for a mechanism to mitigate latency skew of parallel worker functions to reduce the E2E latency.

2.2 Performance Modeling

Modeling Latency as a Distribution rather than a Single Statistic. To estimate the E2E latency and cost of a serverless DAG, it is essential to model the latency of each component as a distribution. For example, the latency of the image classification function (`Classify-Frame`) in the Video Analytics DAG can vary by up to 2× when processing different frames, even when keeping the VM-size fixed to 1 GB. Although similar performance variability can be observed in server-centric platforms, our model is geared to serverless platforms due to their ability to scale resources according to demand virtually unboundedly and hence showing negligible queuing times [56]. Now consider a simple stage of two `Classify-Frame` functions running in parallel. Let X and Y be random variables representing the latency of each. The E2E latency of the two workers combined is given as $P(Z \leq z) = P(X \leq z, Y \leq z)$, which depends on the slowest of the two and hence knowing their median or even their P99 latencies is not sufficient to estimate their combined CDF. In fact, we need the *entire* distribution of *both* components to estimate the E2E latency distribution. Moreover, simply using statistical tail bounds is not suitable for our purpose. For example, Chebyshev's inequality uses the mean and variance to establish a loose tail bound and it is not known how to combine tail bounds for in-series and in-parallel functions with their correlations.

Modeling Correlation. To accurately estimate the combined latency distribution for in-series or in-parallel functions, we need to capture the correlation between their execution times.

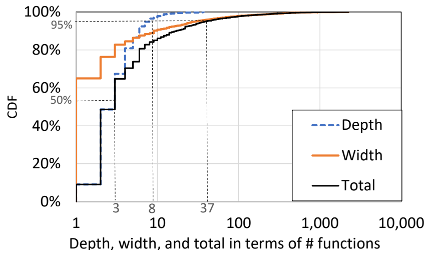


Figure 4: Characterization of depth (number of stages), width (degree of parallelism), and total number of nodes. Depth is low (P50 = 3; P95 = 8). 65% of DAGs are linear chains (no fanout) and width reaches 37 at the 95th percentile.

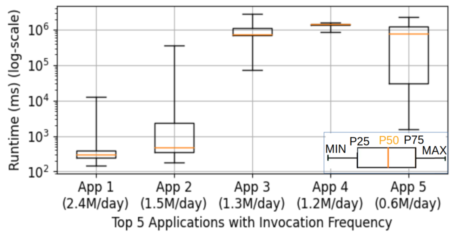


Figure 5: Latency distributions for the top-5 most frequent applications executed by **Azure Durable Functions** *over a period of 1 week. We notice that the execution time varies significantly across different invocations of the same application.*

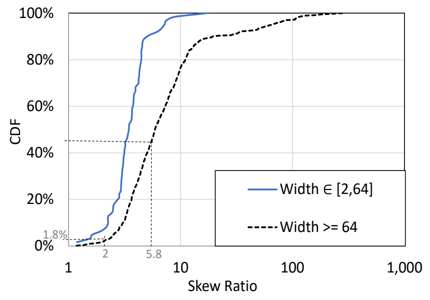


Figure 6: Skew CDF for stages with different width ranges. 98.2% of the DAGs have a skew $\geq$ 2X, and skew increases for wider DAGs.

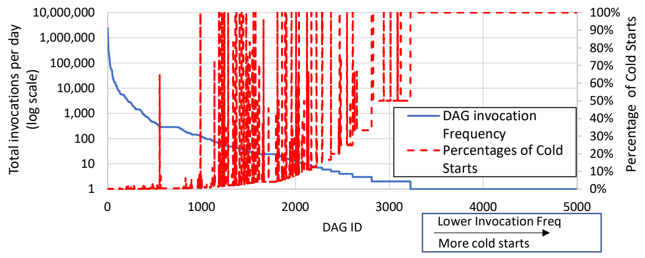


*Figure 7: DAG invocation frequency (Blue solid) and its impact on the percentage of cold starts (Red dashed). DAGs with low invocation rate (***e.g.,** *once per day) always experience a cold start, whereas very frequent DAGs (***e.g.,** *$\geq$ 500 invocations per day) have very rare cold starts.*

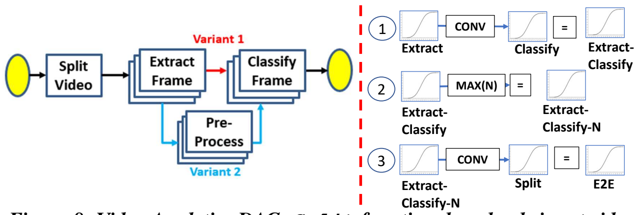


Figure 8: Video Analytics DAG. **`Split`** *function downloads input video and splits it into chunks. Each chunk is passed to an instance of* **`Extract`** *function — extracting a representative frame, sent either to* **`Classify`** *function (Variant #1), or sent to Pre-process function and then* **`Classify`** *(Variant #2). The steps of estimating E2E latency distribution for Variant #1 are on the right.*

Ignoring correlation by assuming statistical independence leads to over-estimating the combined distribution for in-parallel functions, while it leads to under-estimation for in-series functions. In our evaluation, we give quantitative evidence of these effects (§ 5.4.2) and show that a performance model that is distribution-agnostic (*e.g.*, SONIC [40]), or correlation-agnostic (*e.g.*, [26]), fails to provide accurate E2E latency estimates.

3 Design

We first describe the performance model and E2E latency estimation. Then, we describe how we use the performance model to perform our three optimizations.

3.1 Modeling E2E Latency Distribution

Modeling Functions Runtimes. We represent the runtime of a function as the sum of its initialization and execution times. Since both phases have a high variance, we represent them as separate distributions. This separation allows us to estimate the gains from each optimization. Allocating the right resources and bundling mainly impacts the execution times, whereas pre-warming reduces initialization times.

Combining Latency Distributions. Given a latency distribution for every function, ORION applies a series of statistical operations to estimate the DAG E2E latency distribution.

For two in-series functions with latency distributions represented as random variables X and Y, we use `Convolution` to estimate their combined distribution as: $P(Z=z) = \sum_{\forall k} P(X=k, Y=z-k)$.
If X and Y are independent, we simplify the computation: $P(Z=z) = \sum_{\forall k} P(X=k) \cdot P(Y=z-k)$.

The latter is simpler to estimate since it only requires the marginal distributions of the two components, which can be profiled *separately*, rather than *jointly*.

On the other hand, if the two functions execute in parallel, then their combined latency distribution will be defined by the max of the two. Therefore, we use the `Max` operation to combine their CDFs as follows: $P(Z \leq z) = P(X \leq z, Y \leq z)$. Similar to the `Convolution` operation, a simpler form can be used when X and Y are independent: $P(Z \leq z) = P(X \leq z) \cdot P(Y \leq z)$, which uses the marginal CDFs of the two functions, rather than their joint CDF.

Handling Correlation Among Functions.

We consider two types of statistical correlation in the DAG: in-series and in-parallel correlation. For example, our Video Analytics application (Figure 8) has high correlation between *Pre-process* and *Classify* stages, and also high correlation between parallel *Extract* invocations or parallel *Classify* invocations. (Table 1). By analyzing the correlation between the stages as well as the correlation between the parallel invocations in the same stage, ORION identifies both types of correlation. We consider a Pearson correlation coefficient value > experimental parameter θ, as an indication of correlation. This then determines whether to apply the independent or dependent formulation for the `CONV` or the `MAX` operation. In our experiments, we find that the performance of ORION is relatively insensitive for $\theta \in (0.2, 0.5)$ and we run all the experiments with $\theta = 0.4$.

To determine the length of correlation chains (pairwise, etc.), ORION uses conditional entropy measurements of the execution time and compares the reduction in entropy by including additional terms. Thus, if marginal entropy of stage Y is $H(Y) \gg H(Y|X_i) \approx H(Y|X_i, X_j) \approx H(Y|X_i, X_j, ..., X_s)$, where X_i denotes the random variable of invocation i's execution time, then ORION infers that correlations across stages are at most pairwise. We find that correlations in all our application DAGs *across* stages are at-most pairwise. Our formulation, however, can handle any degree of correlation, not just pairwise. Only the amount of profiling data needed will increase with higher degrees of correlation.

In case of high correlation between N parallel invocations in the same stage, the `max` operation can be expanded by the chain rule: $P(Z \leq z) = P(X_1 \leq z).P(X_2 \leq z|X_1 \leq z).....P(X_N \leq z|X_1 \leq z, X_2 \leq z, \ldots, X_{N-1} \leq z)$, which is further simplified in case of pairwise correlations by only conditioning on one invocation, hence all conditional terms reduce to: $P(X_i \leq z|X_{i-1} \leq z)$.

Since all components within a stage are identical, we estimate the above equation as follows:

$$P(Z \leq z) = P(X_i \leq z).[P(X_k \leq z|X_i \leq z)]^{n-1} \quad ,k \neq i \qquad (1)$$

Therefore, we use two distributions to model the `max` for *any* number of parallel components — the marginal distribution and the conditional distribution.

In practice, all individual components are used to estimate the marginal distribution and all pairs of components are used to estimate the conditional distribution, as all marginal distributions are identical and so are all conditional distributions.

Using this performance model, ORION designs three optimizations for serverless DAGs, which we describe next — ***Right-sizing*** in § 3.2, ***Bundling*** in § 3.3, and ***Right pre-warming*** in § 3.4.

3.2 Allocating the Right Resources

The target of this optimization is to assign the right resources for each function in the DAG so that the entire DAG meets a latency objective with minimum cost. Normally, the user picks the VM-size for each function, and the VM-size controls the amount of allocated CPU, memory, and network bandwidth capacities. What makes this problem challenging is twofold — the scaling of multiple orthogonal resources is coupled together and the scaling of different resources is not linear. As an example, the `Classify-Frame` function in the Video Analytics DAG has a small memory footprint (540 MB). However, increasing its VM-size above 1,792 MB (as that size comes with a full vCPU [7]) reduces latency. This is because larger sizes come with a fraction of a second core up to six cores, which this function utilizes. The first step in this optimization is to map a given configuration candidate (*i.e.*, a vector of VM-sizes, with one entry per stage) to the corresponding latency distribution. To achieve this, we build a per-function performance model that maps VM-sizes to latency distributions. Next, we combine these distributions to estimate the E2E latency distribution.

Per-function Performance Model. For each function in the DAG, we collect the latency distributions for the following VM sizes: *min* (the minimum VM size needed for this function to execute), 1,024 MB, 1,792 MB, and *max*. We pick 1,024 MB as it is the point of network-bandwidth saturation (increasing VM-size beyond it does not provide more bandwidth), and 1,792 MB as it comes with one full CPU core. Hence, this initial profiling divides the configuration space into 3 regions: [Min, 1024), [1024, 1792), and [1792, Max].

In order to estimate latency distribution for intermediate VM-sizes, we use percentile-wise linear interpolation. For example, the P50 for 1408 MB is interpolated as the average between the P50 for 1,024 MB, and the P50 for 1,792 MB settings. This generates a *prior* distribution for these intermediate VM-size settings. To verify the estimation accuracy in a region, ORION collects a few test points using the midpoint VM-size 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, ORION collects more data for the region midpoint and adds it to its profiling data. In summary, this approach divides the space of a potentially non-linear function into a set of approximately linear functions, and hence, more complex functions get divided into more regions, with a profiling cost overhead. In practice, we find that 5 to 6 regions accurately model the latency distributions for all functions in our applications.

Optimizing Resources for a Latency Objective. Since the performance model estimates the E2E latency distribution of the DAG, we can use it to choose a configuration (*i.e.*, the set of VM sizes) to execute a DAG in order to meet a user-specified latency objective while reducing \$ cost. We search for the configurations using a heuristic based on Best-First Search (BFS) [57]. The pseudo-code is shown in Algorithm 1.

The algorithm starts by creating a priority queue, in which all the new states are added. A state here represents a vector of VM sizes, one for each stage. Each new state expands the current state in one dimension (with a step-size of 64 MB) and the start state S_0 has the minimum VM size for every function. The priority is set to be the difference between the target latency and each state's estimated latency multiplied by the \$ cost of the new state (lower value means higher priority). Our chosen heuristic is suitable for our problem because latency is a monotonically non-increasing function of resources allocated to a function. The worst-case complexity of BFS is $O(n * log(n))$, where n is the number of states.

3.3 Bundling Parallel Invocations

Stragglers can dominate an application's E2E latency [11, 16, 36, 53]. Here we show how to bundle multiple parallel invocations in one stage within a larger VM, rather than the

Algorithm 1 Best-First algorithm to identify the best VM sizes for functions in a DAG given a user-defined latency objective.

```
Input: Latency-Percentile=P, Latency-Objective: $T_O$
Output: Best VM sizes=$S_{best}$
1:  ## Initialize priority queue pq, performance model GetLatency, cost
    model GetCost, $StepSize$ = 64 MB
2:  ##Set start state $S_0$ to minimum VM size for every function in DAG
3:  pq.insert($s_0$)
4:  while pq is not empty do
5:      $S_{next}$ = pq.pop()
6:      ## Create N new states by adding $StepSize$ to each function
7:      ## Set the priority of each state and add to pq
8:      for $i = 0- > |S_{next}|$ do
9:          $S_{new}$ = $S_{next}$
10:         $S_{new}$.VMsize[i] = $S_{next}$ .VMsize[i] + $StepSize$
11:         $S_{new}$.latency = GetLatency($S_{new}$, P)
12:         $S_{new}$.priority = -1 $\times$ $S_{new}$.latency $\times$ GetCost($S_{new}$)
13:         pq.insert($S_{new}$)
14:         ## Check if latency objective is met
15:         if $S_{new}$.latency $\leq T_O$ then
16:             return $S_{best} = S_{new}$
17:         end if
18:     end for
19: end while
20: ## If no explored state meets the latency objective
21: return State $S_{best}$ with closest latency to $T_O$
```

current state-of-practice of executing each in a separate VM. This promotes better resource sharing, thus mitigating skew.

To understand how bundling works, consider the example shown in Figure 9 for a stage of 4 parallel worker functions experiencing load imbalance. Specifically, the load for workers #2 and #4 is low and both require only one time step of execution. In contrast, the load for workers #1 and #3 is higher and requires 7 and 3 time steps of execution, respectively. Also, assume that the workers are scalable and can actually make use of additional resources made available. If we execute each worker function on a separate VM, say with 1 core each, the E2E latency is dominated by the slowest worker and the entire stage will take 7 time steps. However, if we bundle the workers together in a single VM with 4 cores, the E2E latency reduces to 3 time steps only. This is because the straggler workers get access to more resources when the lightly loaded workers finish their execution. Notice that the cost remains the same in both cases because they consume 1 core $\times$ 12 time steps or 4 cores $\times$ 3 time steps for the entire stage.

We make a few observations about the applicability of bundling. *First*, bundling is useful in reducing the latency if the execution skew is due to *load imbalance*, which arises from processing bigger partitions of data, or inputs that require more computation. We detect load imbalances due to content by subtracting latency CDF #1 from #2: (#1) When the function is executed multiple times with the same input. (#2) When the function is executed multiple times with different inputs. Moreover, the higher the correlation between workers, the lower the gap between their execution times, and hence, the lower is the benefit from bundling.

Second, for bundling to be useful, the function has to be *scalable* to benefit from the additional resources. We identify a function's scalability using our performance model to estimate the impact on the function's latency CDF when given more resources (§ 3.2). We benefit from the fact that the community has developed many highly scalable libraries, *e.g.*, [10], which are widely used in serverless applications.

Third, our example in Figure 9 assumes there will be *no contention* between bundled workers. However, in practice, we find that this contention can be high, especially for network-bound or IO-bound functions, as these resources do not scale linearly with the VM size. For example, all VM sizes in AWS Lambda get the same disk space of 512 MB and network bandwidth scales only for VM sizes until 1,024 MB.

Based on these three requirements, ORION identifies the best bundle size in two steps. First, ORION identifies functions that experience execution skew and are scalable using the performance model. Second, ORION searches the space of bundle sizes through multiplicative increase (*i.e.*, bundle sizes of 1, 2, 4, etc.). At each step, ORION collects very few profiling runs (we use 10) to capture contention. The search terminates when bundling more workers causes contention and hence increases the E2E latency. ORION strives to spread stragglers across different VMs, by performing a "redistribute" operation if needed, so that each straggler has excess resources to speed up its execution. Since skew often shows up with temporal locality, we spread the parallel functions among the available bundles in a round-robin manner. For example, for the Video Analytics application, load typically varies gradually across consecutive frames.

ORION's security considerations: ORION does not currently bundle functions in different stages for security purposes. Moreover, all the invocations to be bundled together belong to the same user and the same DAG invocation. Additionally, in cases when the stages have very different resource requirements, it becomes counter-productive to come up with one VM size that fits multiple stages. We defer the possibility of bundling invocations across different functions as future work.

3.4 Pre-warming to Mitigate Cold Starts

We describe our approach to mitigating cold starts, leveraging the DAG structure of the application. We describe how to identify when to start pre-warming the VMs for each stage in the DAG, in order to balance the E2E latency and the utilization of the computing resources. This step is performed after we perform the previous two optimizations: Right-sizing and Bundling. Figure 10 shows conceptually the impact of different pre-warming delays on E2E latency and utilization. At the extreme, a delay of zero for every stage minimizes the E2E latency but also minimizes the utilization. The other extreme is no pre-warming at all, which is the state-of-practice. First, we define `pre-warming delay` for a stage S as the time elapsed between the start of the DAG execution and the beginning of initialization of the VMs for that stage. For a given DAG of

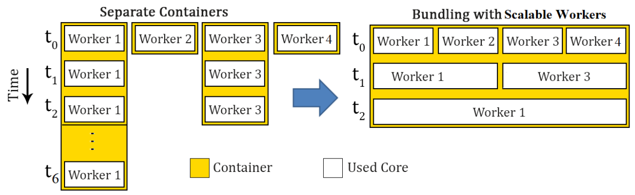


Figure 9: (Left) Separate VMs: workers #2 & #4 finish early, while workers #1 & #3 take longer. (Right) Bundling: after workers #2 & #4 finish, workers #3 & #4 get more resources reducing stage latency.

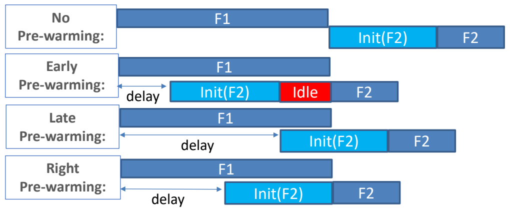


Figure 10: Impact of different pre-warming decisions on the E2E latency and utilization for a chain of two in-series functions. Without pre-warming, the E2E latency increases due to added initialization time of function F2. Both early and late pre-warming are not desirable.

N stages, we want to select a vector $\vec{d} = [d_1, d_2, ...d_N]$ representing the pre-warming delays for each stage in the DAG. For the first stage in the DAG, we have the degenerate case and set its delay (d_1) to zero. This is because pre-warming requires predicting when the DAG will be invoked, which is challenging in the general case. The optimal delay vector, given a performance model $\mathcal{P}$, is defined as follows:

$$\vec{d}^* = \underset{\vec{d}}{\arg\min}\ \textit{E2E-Latency}(\mathcal{P}, \vec{d}) \qquad \text{subject to} \quad \textit{Util}(\mathcal{P}, \vec{d}) \geq \text{Target Utilization} \tag{2}$$

The selected vector is the one that minimizes the DAG E2E latency while achieving the target resource utilization as set (and dynamically adjusted) by the provider. Both the utilization and the E2E latency are estimated by our performance model $\mathcal{P}$. The metric $\textit{Util}(\mathcal{P}, \vec{d})$ measures the utilization for a given delay vector using the performance model, and is estimated as $\frac{BusyTime(VM)}{BusyTime(VM)+IdleTime(VM)}$. $BusyTime(VM)$ includes both initialization and execution times, while $IdleTime(VM)$ is the time between when the initialization completes to when the function starts executing. We again use Best-First Search (BFS) to select vector $\vec{d}^*$ as follows. We start by setting all values of $d_i = 0$. In each iteration, we add a delta (100 ms) to the delay factor d_i that yields the best improvement in utilization over the current state without increasing the E2E latency. The algorithm terminates when adding delta to any delay factor does not improve utilization but increases E2E latency.

3.5 Further Design Considerations

Deployment. ORION is designed to serve as a DAG optimization layer. Although the primary use case is to be deployed by the provider, ORION can also be deployed by end-users. In the latter, users are able to select the VM sizes for their functions. For this, the end-user will need to profile her code and also send pre-warming requests to the provider at the right times, as identified by ORION. However, users do not need to change their function code for Bundling. Instead, ORION identifies the bundle size for each parallel stage in the DAG and executes multiple invocations together. The cloud provider still decides the mapping of specific VMs to function bundles.

Naturally, ORION's performance model is trained faster for functions with higher invocation frequency as they provide natural training data points. As discussed in 2.1, frequently invoked DAGs dominate the total set of DAG invocations. For example, the 5%-most frequent DAGs have an invocation rate of 2.3K per day. Hence, it will take us less than 3.5 hours to gather 300 training samples per function. We can accelerate this, and also handle less frequently invoked DAGs, by inserting synthetic but realistic DAG invocations to generate training data points. It is also possible that we will have to re-train our models from time to time when the workload characteristics have changed significantly, or less commonly, the application DAG or the infrastructure characteristics has changed significantly. This incremental training is *not* a computationally heavy task as it involves updating only *parts* of the distribution curves. Maintaining the latency data for performing such updating is also *not* a memory-heavy task.

4 Implementation

We implement ORION in C# and Python 3.8 with 2,100 LOC. We execute the serverless DAG applications in AWS Lambda and use Amazon S3 for data passing between the functions. We use AWS Step Functions [6] to orchestrate the DAG. Function bundling is implemented without any code change by using a wrapper around the (developer-provided) entry point to each function. We use the Python multiprocessing library [9] to execute bundled invocations together.

Runtime Overhead. In theory, the worst-case runtime of Algorithm 1 increases exponentially with the number of stages in the DAG. However, we find that ORION's BFS algorithm has a very low overhead in practice: Each iteration in Algorithm 1 takes [3,7.5] msec, and the best solution takes between 6 and 88 iterations while exploring $< 1\%$ of all possible states. The number of iterations depends on the latency target, the steepness of the latency-VM size relation, and the step size (we use 64 MB). For finding the best pre-warming delays, BFS takes between [0.4,3] seconds across all applications.

Scalability. We evaluate the scalability of ORION in Figure 18 with respect to increasing the number of stages. We synthetically replicate the last (and most time consuming) stage of the Video Analytics application to create a DAG of up to 8 stages. The overhead is defined as the inference time divided by the application lifetime, and it ranges between 0.12% for 3 stages to 0.07% for 8 stages. Also, increasing the number of stages, the prediction error increases, but slowly. Specifically, with up to 8 stages, P50 error is stable, and P90 and P95 increase slowly but never reach 15%. With wider DAGs, our inference time remains unchanged as the fanout degree is used as a parameter in our estimation of MAX (Eq. 1).

Pre-warming. Ideally, implementing pre-warming in AWS Lambda requires our control over assigning invocations to warm containers or VMs. Since we do not have such control, we rely on AWS Lambda's container reuse to implement pre-warming. Specifically, we perform pre-warming by sending a dummy call to a function, then send the actual call right after the response from the dummy call is received.

5 Experimental Evaluation

We evaluate ORION running on AWS Lambda. First, we describe the three serverless DAG applications used in our evaluation. Then, we show an E2E evaluation compared to three alternatives. Next, we show a set of microbenchmarks to evaluate each component of ORION. Finally, we provide a unit experiment on Azure Functions, a platform that allows less configurability for an external mechanism like ORION.

5.1 Serverless DAG Applications

Video Analytics. This application, adopted from Pocket [35], analyzes an input video by extracting representative frames from the video and classifying each frame. The application stages are shown in Figure 8. The first variant of this application directly calls a third function, *Classify-Frame*, which uses a YOLO [45] pre-trained DNN model to classify object(s) in the frame into 1,000 classes. The second variant calls an intermediate pre-processing function, *Pre-process*, which applies a sharpening filter to improve image quality before classification. We refer to this variant as "Video Analytics w/ Preprocess" (VA-Pre, for short). Finally, all classification results are uploaded to remote storage. For VA-Pre, there is a high correlation between the times of the *Pre-process* and the *Classify* functions. We use 600 YouTube videos (300 for profiling, 300 for testing), each of length 1 min, belonging to the "Nature" and "News" categories.

ML Pipeline. This application is a machine learning pipeline (adopted from Cirrus [18]).It consists of three stages: dimensionality reduction (PCA), model training, and testing (*Combine*). The second function, *Train-Model*, runs in parallel and each instance trains a decision tree model using the *LightGBM* Python library [37]. In this stage, a user-specified number of functions is triggered (we use 64 trees in our evaluation), and every function trains a different decision tree. The third function, *Combine*, combines the trained models into a random forest and evaluates its joint accuracy on a held-out test dataset. We use the MNIST [23] database of handwritten digits that has a total of 60K images. We execute the application with 600 runs (300 profiling, 300 test), and in each run, we use 5K images to train the ML model, and 15K for testing.

Chatbot. This application trains a domain-specific Natural Language Understanding (NLU) model, whose task is to identify the accurate "intent" of a user-spoken utterance. We use the Chatbots Intent Recognition Dataset, available on Kaggle [44], which consists of 22 intents and 455 utterances. As before, we evaluate with 300 profiling and 300 test runs. The first lambda in this application parses the dataset and constructs bag-of-words representation for all utterances. Next, a stage of parallel lambdas trains One-vs-Rest classifiers with one lambda per intent. The models are then uploaded to remote storage for real-time intent detection.

The three applications cover important characteristics of serverless DAGs. Specifically, Video Analytics and Chatbot have scatter communication pattern, whereas ML Pipeline has a broadcast pattern. They also cover different fanout degrees (22 for Chatbot, 32 for Video Analytics, 64 for ML Pipeline) and their execution times resemble the average latency of DAGs in our workload characterization (§ 2.1). Moreover, Video Analytics and ML Pipeline are both compute bound, whereas Chatbot is network bound.

5.2 ORION and Competing Approaches

We compare our E2E latency and cost to multiple resource allocation, skew mitigation, and pre-warming approaches:

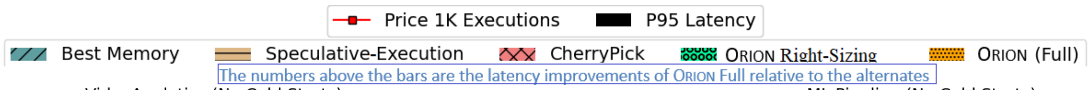
(1) **Best-Memory:** This is a resource allocation approach that uses our performance model and progressively increases the VM size for every function in the DAG till the latency objective is met. This mimics the standard VM autoscaling that is employed in many cloud scheduling solutions [2, 59]. All invocations run in separate VMs of the same size.

(2) **CherryPick [5]:** CherryPick uses Bayesian-Optimization (BO) to find latency-optimized memory configurations. BO relies on an *acquisition* function to propose new points to sample next. This makes BO a distribution-agnostic baseline as each VM size is profiled once, then a new size is selected by the acquisition function. We set the loss function in BO to be the difference between the achieved latency and the user-specified latency target. Since CherryPick is distribution agnostic, it cannot detect execution time skew and hence performs no bundling. We choose CherryPick as it (BO with Gaussian processes) was recently demonstrated as the most competitive approach in the category [15] and recent approaches have used it for configuring serverless functions [3].

(3) **Speculative-Execution:** This is a skew mitigation approach that identifies stragglers at runtime and executes a duplicate invocation on a different VM. Speculative execution is widely used in MapReduce, Hadoop, and Spark systems for skew mitigation to reduce tail latency [11, 19, 31, 51]. We adapt this baseline from Spock [30] (specifically the technique called "conservative autoscaling in predictive mode"). Since the skew is caused by the input's content, the new invocation will likely take as long as the first one unless it is assigned more resources. Accordingly, we modify the technique by assigning the "Max" resources (10 GB) for the new invocation.

(4) ORION **Right-Sizing:** This variant of ORION performs Right-Sizing only, and not the other two optimizations.

(5) ORION **Full:** This is our complete solution, which includes



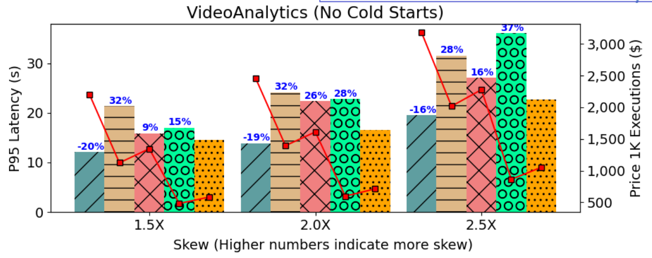


Figure 11: Skew is varied by changing the detection probability threshold as: 2%, 10%, and 15%, with lower values resulting in more detected objects and higher skews [22].

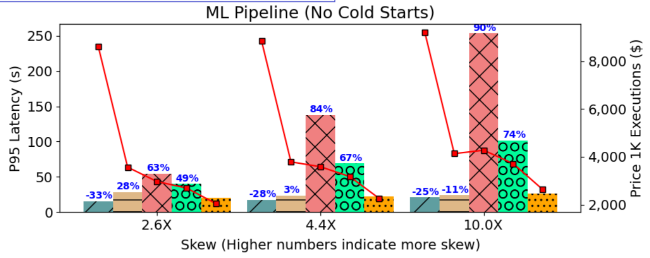


Figure 12: Skew is varied by changing the maximum value for each hyper-parameter, e.g.*, we vary the max number of trees as: 50, 100, and 200, and these map to 2.6×, 4.4×, and 10× skews.*

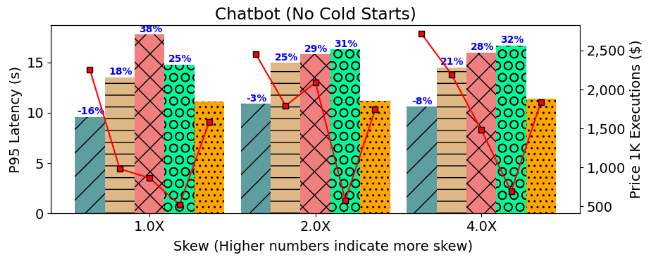


Figure 13: Skew is altered to 1×, 2×, and 4× by changing the number of training epochs as: 100, 500, and 200.

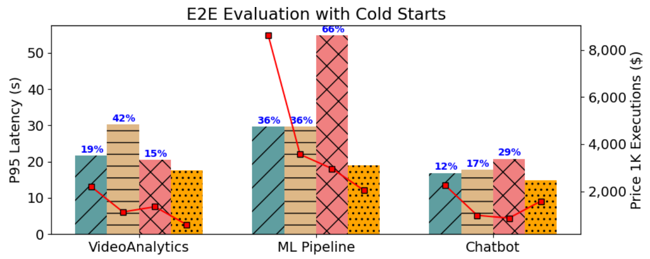


Figure 14: E2E evaluation with cold starts. ORION *achieves the lowest latency and cost compared to all baselines.*

Right-Sizing, Bundling, and Pre-Warming.

5.3 End-to-End Evaluation

We show the P95 latency (primary Y-axis, shown using bars) and cost (secondary Y-axis, shown using lines) of each approach in Figures 11, 12, and 13 for the three applications. The numbers above the bars are the latency improvements of ORION Full relative to the alternates. First, we set the latency objective to the minimum achievable latency, which is identified by executing all functions with *max* VM size, while computation skew is minimized. For each application, we vary the skew in a controlled manner through application-specific parameters. For each solution and for each experimental point, we execute each application 300 times and highlight the gains of ORION's right sizing and bundling. In this part, we take care to eliminate all cold starts for the experimental data points. Later, we show the impact of cold starts and the additional gain due to ORION's pre-warming design in Figure 14. Compared to Best-Memory, ORION has a slightly higher latency since Best-Memory assigns high resources to all workers, including stragglers. However, this baseline increases the cost significantly by assigning identical resources for each stage and parallel running workers in separate VMs, which over-provisions the resources to meet the latency objective. ORION provides [33%,71%] lower cost by assigning the right resources for each function and bundling parallel workers. Compared to Speculative-Execution, we notice that ORION has consistently lower latency and cost across all skews. For example, with the lowest skew, ORION shows [18%, 32%] lower latency and [46%, 57%] lower cost for the three applications. This is because Speculative-Execution detects straggling workers (using a user-specified threshold) and re-executes them on new VMs with the max size. This causes an additional delay due to the wasted execution time. It also increases cost as it sometimes mistakenly re-executes workers that would finish shortly after the threshold. ORION's bundling does not require any user-specified threshold to detect straggling workers, assigning them more resources once co-located workers finish and release their resources.

For CherryPick, since it is distribution-agnostic, we modify the BO algorithm so that 100 points are profiled for each point selected by BO's acquisition function to measure the latency percentiles. We run CherryPick for 100 iterations total (a generously high number compared to the original work and follow-on works), resulting in 10K profiles for each application. Notice that ORION requires only 300 profiling runs to model the E2E latency distribution, reducing the profiling burden of CherryPick by 97%. Compared to CherryPick, we notice that ORION Full consistently provides lower latency and cost, except for Chatbot where CherryPick has higher latency but lower cost. Specifically, with the highest skew, ORION Full shows [16%, 90%] lower latency and [38%, 53%] lower cost for Video Analytics and ML Pipeline. For Chatbot, this application has a lower bundle size than the others, reducing the gain from ORION's bundling mechanism. Compared to ORION Right-Sizing, adding bundling significantly reduces the latency across the three applications. However, bundling causes a slight increase in the cost for Video Analytics by

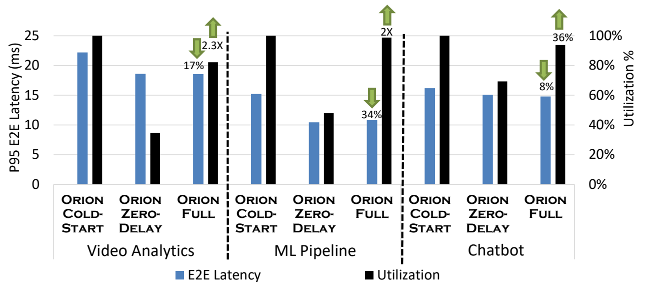


Figure 15: Impact of pre-warming on latency and utilization. We use VM and bundle sizes selected by ORION *and compare different execution strategies w.r.t. cold starts. Percentages over the bars of* ORION *show the improvements in P95 Latency (over* ORION *Cold-Start) and Utilization (over* ORION *Zero-Delay pre-warming).*

22% (relative to No-Bundling), while it causes a decrease in cost for ML Pipeline by 30%. The reason is that the ML Pipeline experiences higher skews (up to 10X), and for higher skew, Bundling is more beneficial. We also notice that the reduction in latency increases with higher skews. For Chatbot also, bundling reduces the latency compared to no bundling, but increases the cost by 163%. This is because the Chatbot application is more network bound and not compute bound than the other two, and hence the best bundle size is only 2, vs [6,8] for the other applications. However, cost with bundling is still 33% lower than Best-Memory, which is the closest to us in latency among all baselines.

Mitigating Cold Starts with Pre-warming. So far, we have compared ORION to the baselines with only warm executions. Now we show the gain of our pre-warming technique and how useful it is in reducing cold starts. Figure 14 shows ORION's latency and cost vs other baselines in the case of cold start for every function in the DAG. We notice that all baselines are impacted by cold starts and their latencies increase, whereas ORION's pre-warming technique is able to mitigate the impact of cold starts. For example, Best-Memory shows an increase of E2E latency over ORION by 19%, 36%, and 12% for Video Analytics, ML Pipeline, and Chatbot, respectively. Similarly, Speculative-Execution suffers from cold starts twice, once for the first execution with the small VM, and once more for the second execution with the max VM size. Hence, ORION's improvements in latency over Speculative-Execution increase to 42%, 36%, and 17% for the three applications. To summarize, ORION's three optimizations of Right-sizing, Bundling, and Right pre-warming provide lower E2E latency and cost over all competing approaches. In the next section, we show a set of microbenchmarks to separately evaluate the performance of each component of ORION.

5.4 Microbenchmarks

5.4.1 Impact of Pre-warming on Utilization & Latency

Figure 15 shows the latency and utilization achieved by ORION versus its two variants. The first, called ORION Cold-Start, does not perform any pre-warming and hence suffers from increased latency, yet has very high utilization as it causes no idle times. The second, called ORION Zero-Delay, initializes all the containers with zero delay for all stages, *i.e.*, at the beginning of the DAG execution. Hence it ensures the lowest latency that can be achieved, but incurs increased idle times due to early pre-warming and hence suffers from low utilization. On the other hand, ORION Full uses the right delay times identified by BFS (§ 3.4). As shown in Figure 15, ORION Full consistently achieves lower latency than ORION Cold-Start, and consistently higher utilization over ORION Zero-Delay for all three applications. We also notice that the latency gains are higher for ML Pipeline and Video Analytics than for Chatbot, which is due to the higher initialization times observed in these two applications when downloading the heavy ML packages and the large pre-trained object detection models. Therefore, estimating the right values of the delays for each stage, as done by ORION, is essential to mitigate cold starts without significantly reducing utilization.

5.4.2 Evaluation of Performance Model

Capturing Correlation between Functions.

Here we evaluate how much correlation exists in our target applications. We calculate the Pearson's correlation coefficient between in-series functions (*e.g.*, between *Split-Video* and *Extract-Frame*), and between in-parallel functions (*e.g.*, between multiple instances of *Extract-Frame*). We show the correlation scores in Table 1 for Video Analytics.

Table 1: **Correlation between execution times of functions in the Video Analytics DAG. In-series correlation is low but in-parallel correlation is high.**

VM-Sizes (in MB)	**In-series Correlation**			**In-parallel Correlation**	
Split, Extract, Classify	Split ⇳ Extract	Extract ⇳ Classify	Preprocess ⇳ Classify (VA-Pre)	Extract	Classify
192, 192, 576	0.09	0.04	0.45	0.05	0.43
1024, 1024, 1024	0.07	0.02	0.61	0.34	0.44
1792, 1792, 1792	-0.07	-0.04	0.69	0.48	0.58
3008, 3008, 3008	0.05	-0.01	0.88	0.65	0.51

The correlation scores between in-series components are close to zero (0.036 on average for Video Analytics, 0.06 for ML Pipeline, and 0.04 for Chatbot), while the correlation scores between functions in the same stage are high for Video Analytics (0.55), while low for ML Pipeline (0.052) and for Chatbot (0.03). For Video Analytics w/ Preprocess, `Pre-process` has a high correlation with in-series `Classify` functions (0.65). Therefore, we apply the dependent `conv` operation between `Pre-process` and `Classify`, while we use the independent `conv` operation for all other in-series functions for all applications. Additionally, we incorporate correlation when performing `max` operation (if correlation is detected) by using the conditional distribution.

Estimating E2E Latency Distribution. Here we evaluate the accuracy of ORION in predicting the E2E latency

Table 2: **Video Analytics: Error rates for ORION's E2E latency estimation. Abbreviations: S→Split, E→ Extract, and C→Classify**

Video Analytics						
VM Sizes (MB)	ORION		Distribution Agnostic		Correlation Agnostic [26]	
S, E, C	**P50**	**P95**	**P50**	**P95**	**P50**	**P95**
512 , 1280 , 1536	14.0%	13.0%	40.0%	15.6%	78.7%	47.5%
768 , 1280 , 2240	14.0%	12.0%	35.4%	11.6%	67.7%	38.3%
1536 , 512 , 1536	13.0%	11.0%	39.7%	16.7%	79.4%	49.9%
1792 , 1792 , 576	6.4%	11.8%	11.6%	-39.0%	49.7%	-18.2%
6000, 6000, 6000	14.5%	10.7%	24.2%	3.3%	56.9%	30.5%
MAPE	**13.0%**	**12.0%**	**32.0%**	**21.0%**	**68.0%**	**39.0%**

distribution for the entire DAG. We compare to two baselines — distribution-agnostic (as mentioned earlier, any BO-based technique like CherryPick falls in this category) and correlation-agnostic (*e.g.*, [26]). The results are shown in Table 2 for Video Analytics.

ORION estimates the E2E latency distribution for the applications with low error rate (<15%), much lower than those of both baselines. We find through drill down of our estimation error that: (i) our estimated length of correlation chains as pairwise (§ 3.1) is accurate and hence does not lead to much error (ii) the dominant source of error lies in the interpolation of the CDFs for each function for the *unseen* memory configurations. This is despite our design, where if the interpolation causes too much error, the memory region is split into two and further data points are collected (§ 3.2). These observations hold across all three applications. Error rates are higher in Video Analytics relative to ML Pipeline because the execution time is content sensitive for the former. Our technique does *not* create content-specific models since we (and any provider-side tool) cannot have visibility into user data due to privacy concerns. The ***Distribution-Agnostic*** baseline uses the median execution times and predicts the median execution times for unseen configurations by interpolation. This baseline has a high error rate in the range of [-39%, 40%] for Video Analytics, [-5%,108%] for ML Pipeline, and [-6%, 66%] for Chatbot. The ***Correlation-Agnostic*** baseline from [26] also has a higher error rate in the range of [-18%, 79%] for Video Analytics, [-5.4%,103%] for ML Pipeline, and [80%, 111%] for Chatbot. Note that the majority of the errors for the ***Correlation-Agnostic*** baseline are over-estimation, which is caused by ignoring the correlation between parallel workers. In conclusion, it is important to take into account the latency distributions and not simply a point estimate and to account for the correlation across stages and across workers within a stage, even when the correlations are quite weak (Table 1).

5.4.3 Optimizing Resources for a Target E2E Latency

We profile the applications to build the E2E performance model in ORION for all three applications as mentioned in § 5.4.2, then set 6 latency targets per application. ORION proposes the DAG configuration (*i.e.*, VM size for each function in the DAG) to meet each latency target at while reducing cost. We validate ORION's accuracy by executing the application with the proposed configuration and comparing the achieved latency to the user requirement.We notice that ORION's proposed configurations are very close to the latency requirement in all applications, with error rate of [-2.75%, 4.93%] for Video Analytics, [-1.37%, 2.6%] for ML Pipeline, and [-3.5%, 3.7%] for Chatbot. Table 3 lists detailed configurations for Video Analytics. We notice that expectedly, ORION tends to assign more resources as the latency percentile increases (*i.e.*, P50 → P95) or as the latency requirement decreases (50 sec → 30 sec). Also ORION decides to increase the allocated resources for a subset of functions and by different amounts, based on the latency requirement. For example, for ML Pipeline, ORION increases the VM-size of PCA from 768 MB to 832 MB to achieve a latency requirement of (P90 $\leq$ 50 sec). However, ORION decides to increase the VM-size of Combine from 1,408 MB to 1,472 MB to achieve a latency requirement of (P90 $\leq$ 40 sec). This shows ORION's BFS adjusts the *Best* function to increase its resources according to the estimated latency of the current state.

5.4.4 Impact of Varying Bundle Size

We evaluate the impact of varying bundle sizes on the E2E latency CDF and cost (Figure 16). First, we run our Video Analytics application with the best VM size selected by BFS but without bundling. This is an application that is both CPU bound and scalable, and thus a good candidate for demonstrating the effect of bundling. Next, we progressively increase the bundle size and the VM size proportionally. For example, if the best VM size selected by BFS is 1,792 MB (1 core), we use a VM of size 1,792 $\times$ 2 when we bundle pairs together, and so on. We notice that increasing the bundle size from 2 to 6 workers reduces the latency; however, increasing the bundle size beyond that (to 10 and 30) causes an increase in the latency. This is because the maximum number of cores available in AWS Lambda is 6 and hence, at the higher bundle sizes (10 or 30), each worker is getting less than its required resource.

Thus, the design of ORION to choose the best bundle size is essential to optimize latency by avoiding contention.

5.5 Generalizability to Microsoft Azure

To test if ORION generalizes to other FaaS providers, we evaluate our model using Azure Functions as the serverless environment. Azure Functions supports a few plans, but the most popular one is the *Consumption Plan*. In this plan, users are charged for the exact amount of resources consumed by their functions at runtime, whereas all other plans have a flat rate pricing model. Although we have no control over the resources assigned to individual functions when selecting the *Consumption Plan*, we wanted to measure the accuracy of ORION's E2E latency estimates compared to the actual la-

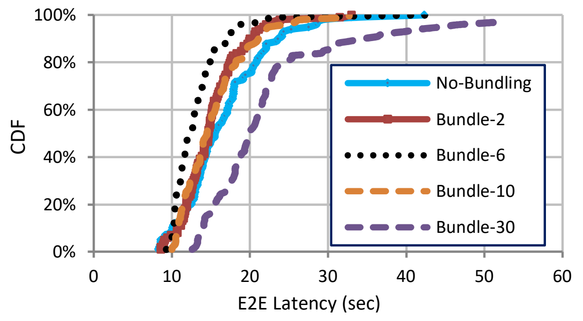


Figure 16: Video Analytics: Impact of varying bundle sizes. No-bunling has high latency due to computation skew. The optimal bundle size here is 6, and using a bundle size of ≥ 10 causes contention and the latency increases.

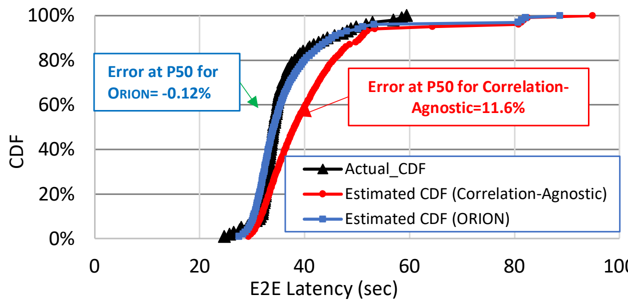


Figure 17: ORION*'s estimated latency CDF vs Actual CDF for Video Analytics application deployed in Azure Functions. Ignoring in-parallel correlation leads to higher errors for the Correlation-Agnostic baseline.*

Table 3: ORION**'s E2E latency-optimized VM sizes.** ORION **meets the latency objective with a low error rate in the range of [-2.75%, 4.93%]**

Video Analytics			
User Requirement	ORION's configs (MB) Split,Extract,Classify	Achieved Latency	Error Rate
P50 ≤18 s	192, 192, 640	18.3 s	1.5%
P95 ≤18 s	384, 192, 768	17.5 s	-2.8%
P50 ≤17.5 s	192, 192, 704	18.4 s	4.9%
P95 ≤17.5 s	640, 192, 768	17.4 s	-0.5%
P50 ≤17 s	256, 192, 768	17.8 s	4.4%
P95 ≤17 s	832, 256, 1024	17.3 s	2.0%

tency observed with this plan. We show ORION's estimated CDF and actual CDF in Figure 17. We use our Video Analytics application with the earlier-mentioned 600 YouTube videos.

We use our E2E performance model to estimate the CDF for the entire DAG. For fair comparison to AWS-Lambda, we also rely on remote-storage (*i.e.*, Azure Blob Storage) for data-passing between the functions. We also show the estimated CDF when correlations among functions are ignored — this corresponds to the "Correlation-Agnostic" baseline from our earlier experiment (§ 5.4.2). We notice that ORION predicts the E2E latency with very low error rates (-0.12% for P50, 1.9% for P90, and 2.5% for P95 latencies). The Correlation-Agnostic baseline has significantly higher errors (11.6% for P50, 14.4% for P90, and 29.2% for P95). Thus, the baseline suffers more for higher percentiles.

6 Pre-warming Policy Simulator

To better understand different pre-warming policies without being constrained by privileges granted by the cloud provider, we build a policy simulator, implemented in Python 3.8 with 1,058 LOC. The simulator takes as input the latency CDFs for stages in the DAG. Policies are implemented through a state machine with different actions being taken in each state (such as FUNC_START, FUNC_END, FUNC_PREWARM, etc.). The output of the simulator are the E2E latency CDF of the DAG and the overall resource utilization. We open source the simulator for future exploration of serverless DAGs [1].

Simulation Results. Figure 19 shows the utilization achieved by a policy with optimal pre-warming using an Oracle that knows the exact runtimes of each function invocation. The input DAG has 2 stages with width of 10 for each stage. The X-axis denotes the skew on the runtime of the first stage. The Y-axis denotes the percentage of variance on the delay chosen by the Oracle for pre-warming functions of the second stage — so if the value is $X\%$ and ORION calculated deterministic delay is Y, then the Oracle can pick a delay in the range $[Y - X\%$ of $Y, Y + X\%$ of $Y]$. Thus, the range of values the Oracle can choose from is capped even if the Oracle determines the optimal pre-warming time for a specific function invocation lies outside of the range. The lowest point on the Y-axis is the optimal deterministic delay determined by ORION for all function invocations in the second stage. We find that the E2E latency is unaffected (not shown) by increasing the size of the range on higher skews, but utilization increases. This is because the policy is able to pre-warm with the ideal delay and hence does not incur any idle time. This shows the theoretical best achievable utilization since we use an Oracle. However, implementing this Oracle has two challenges: (1) Predicting per-function *exact* latency is impractical. (2) Selecting a delay factor for each function invocation rather than each stage increases search space exponentially with DAG width.

7 Related Work

Minimizing cost and/or execution time for serverless chains is the target of a few recent studies. For example, Sequoia [52] makes the observation that current serverless platforms treat functions within a DAG separately, without making use of the DAG structure. SONIC [40] reduces the communication latency between in-series serverless functions by optimizing the data passing strategy. SONIC selects from among the data passing strategies: direct passing, remote storage, and local VM-storage, where only the latter two can be implemented directly in AWS Lambda. Caerus [60] stresses the importance of optimizing latency and cost jointly for serverless DAGs, and achieves this by identifying pipeline-amenable data de-

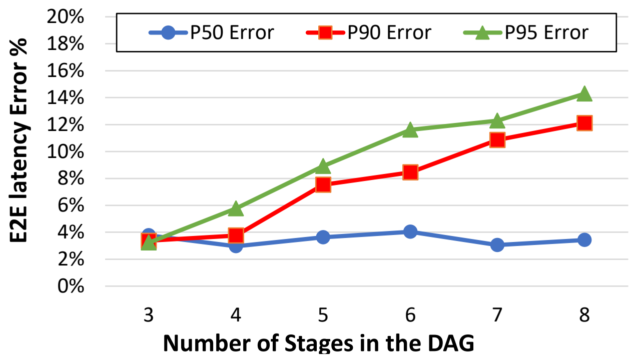


Figure 18: ORION*'s error with varying number of stages. More stages increase the error for the tail, while the median stays stable.*

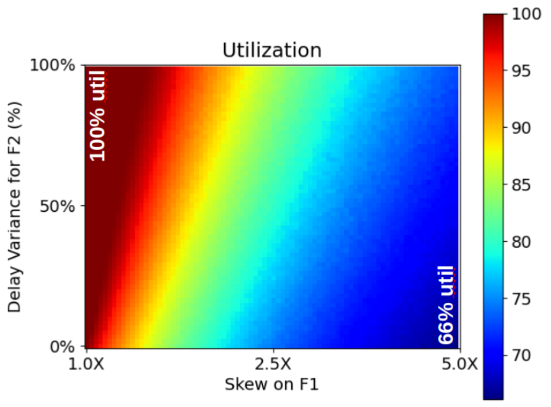


Figure 19: Simulation of an Oracle pre-warming policy where utilization improves with the width of distribution from which the pre-warming delays are chosen. ORION*'s strategy corresponds to the 0% variability,* i.e., *deterministic delay.*

pendencies between stages to find ideal task launch times. Xanadu [21] and Kraken [14] tackle the problem of cascading cold starts in a dynamic DAG. Neither can determine the optimal pre-warming time to mitigate cold starts.

Overall, no prior work in this category considers execution time variance and its impact on cost or utilization.

Latency and Cost Prediction for Serverless Functions. A few prior studies have targeted predicting the execution time and cost for serverless functions. For example, [25] predicts (a point estimate) and optimizes resources for a *single* serverless function by building regression models from a host of synthetic functions. The authors in [26] also observe a variance in execution time in serverless environments, and hence, apply mixture density networks to predict the distribution of the function cost. However, their Monte-Carlo simulation mechanism is very sample inefficient.

ORION uses a more direct method by applying statistical operations to combine the distributions of individual functions and thus, to infer the E2E latency distribution. A number of prior works target reducing the cost of serverless DAGs by optimizing the intermediate data transfer between functions, such as, Costless [27], SONIC [40], Locus [43], and Pocket [35]. They solve an orthogonal problem to ours, namely, reducing the cost of intermediate data transfer. ORION does *not* introduce a new mechanism for intermediate data transfer, nor does it limit or specify the method for state transfer between functions. We use state-of-practice remote storages, such as AWS S3 and Azure Blob Storage. However, ORION would integrate seamlessly with the mentioned systems as the read/write times are included in the latency profiles used in ORION 's model.

Scheduling in Serverless Computing. Photon [24] optimizes *single-stage* serverless functions by doing the equivalent of bundling in ORION, but not for skew mitigation. Its main motivation is to reduce the memory footprint of parallel invocations of a function, while its design sophistication is meant to address security concerns of bundling (out of scope for ORION). One work that targets meeting latency SLAs for serverless DAGs is Atoll [50].

It takes a complementary approach to ours—partitioning a cluster to lower scheduling overheads, and proactively starting up containers and then routing function requests to the appropriate containers.

Resource Optimization in the Cloud. Black-box configuration tuning systems such as CherryPick [5], Selecta [34], OptimusCloud [39], and Ernest [54] target optimizing the cloud resources for a wide range of applications by selecting the right VM type and size, which vary in the amount of allocated resources. However, these systems treat the application as a single component, and thus, do not take the DAG workflow information into account. Further, they are not directly applicable to serverless applications.

Cold Starts Mitigation. Many prior works identified cold starts as a major performance bottleneck in FaaS platforms. Accordingly, several solutions have been proposed such as keeping containers alive [29], leveraging checkpoint/restore operations [49], or using *Pause* containers [41]. Although these solutions reduce the initialization time significantly, there is still a significant user-observable initialization time. ORION hides this initialization time through pre-warming and decides the right time to start pre-warming to minimize idle time, hence keeps resource utilization high.

8 Discussion

Profiling and Modeling Overheads. ORION requires monitoring the execution of the application for a number of runs to accurately capture the latency distribution for each function in the DAG. In our evaluation, for all the applications, a total of 300 profiling runs was found sufficient for accurate 95-percentile latency estimates. Initially, and before convergence is reached, the data collection is performed as a background process while the DAG executes with user-provided configurations. An important design consideration for ORION is that this data collection does not have to happen purely offline and in batch mode. Rather, that is complemented with online data collection and incremental model refinement, which is a lightweight task. When predicted and observed latencies differ significantly (as can happen if the workload or the application changes), we restart the data collection phase to capture the changes in the latency distributions.

Bundling and Performance Model Interaction. Bundling changes the DAG structure (by reducing the fanout degree), and hence, the performance prediction model needs to be updated. Therefore, this becomes an iterative process, with each iteration being *Performance model building* ⇒ *Resource optimization* ⇒ *Bundling*. In practice, we find that a single iteration, or at most two iterations, leads to convergence.

Impact of The Three Optimization. The three optimizations of ORION can have a negative impact on performance, resource utilization, or $ cost if not performed carefully. First, over-provisioning the VM size for all workers to mitigate execution skew (as done by the Best Memory baseline in our evaluation) unnecessarily increases the $ cost (Figures 11, 12,& 13). Second, excessive Bundling (bundle size > right bundle size) can lead to resource contention and increase of the latency (Figure 16). Third, early pre-warming (delay < right delay) decreases resource utilization, whereas late pre-warming increases latency (Figure 15). This motivates the need for an accurate performance model to accurately perform these three optimizations. In terms of cost, we notice that users do not pay for initialization times, hence pre-warming does not impact cost. However, the provider should treat a pre-warming request as a hint since a true invocation is always more important.

Applicability of Performance Model.

ORION is tailored to model the performance for serverless DAGs. In general, the response time of a job includes queuing and execution times. Cloud providers operate large serverless platforms, providing virtually infinite capacity, reducing queuing delays to primarily cold-start latencies [38]. Further, serverless platforms typically limit the execution time of each invocation [8] favoring modular reusable functions. The combination of short queuing and execution times enables ORION to model E2E latency, without the need to predict variable (and long), heavy-tailed queuing times that appear in other environments [20, 33, 46].

Mitigating Infrastructure-caused Delays. In serverless platforms, two types of stragglers can be observed: (1) Stragglers that experience longer execution times due to their input content (e.g., larger data portions or more complex inputs such as video frames with many objects). (2) Stragglers that appear due to infrastructure causes (e.g., network fluctuations). Bundling mitigates the first type of stragglers. The second type is well studied in the literature, and solutions such as *Speculative Execution* [11] work well in practice. Nevertheless, Bundling has a positive side effect of using fewer VMs/containers, reducing the likelihood of occurrence for infrastructure stragglers.

Supporting Dynamic DAGs. In a dynamic DAG, the execution flow is identified at runtime, say based on input data. Such DAGs appear in microservice-based applications [14], among others. ORION, as well as other provider-side tools, cannot have visibility into user data due to privacy concerns. Hence, ORION cannot support dynamic DAGs where the path is determined based on request content.

Future Work. Our bundling approach increases VM size proportionally with the bundle size. For example, assuming a single function invocation use a VM of size VM_{single}, we bundle N invocations in a VM with a size of $N \times VM_{single}$. There is, however, room to explore choosing other VM sizes beyond linear scaling. Furthermore, combining two or more in-series functions together to execute in a single VM can improve performance compared to invoking those function in separate VMs (e.g. due to avoiding remote storage communication). We plan to explore the performance benefits of these ideas.

9 Conclusion

We proposed ORION as a novel optimization technique for serverless DAGs. It presents four design innovations: a distribution and correlation-aware performance model for E2E latency, a resource optimization strategy, a design for bundling multiple invocations of a function within a stage to mitigate execution time skews, and a pre-warming strategy to mitigate cold starts. We evaluate ORION on AWS Lambda on three serverless applications with different DAG structures, skews in execution time, and communication patterns. We compare ORION to three competing approaches and show significant improvements in E2E latency, $ cost, or both. We highlight the following insights: (1) It is challenging to decide on the right resource configurations that accurately meet latency SLOs for serverless DAGs. (2) It is important to bundle parallel workers together to mitigate skew, yet it is challenging to pick the right bundle size that avoids resource contention. (3) We can leverage the DAG structure information along with latency CDF estimates to find efficient pre-warming delays that minimize E2E latency without degrading utilization.

10 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 and artifact evaluators for their enthusiastic comments, and the shepherd, Timothy Wood, for his insightful feedback.

References

[1] Pre-warming Policy Simulator. https://github.com/icanforce/Orion-OSDI22, Last retrieved: May, 2022.

[2] Muhammad Abdullah, Waheed Iqbal, Josep Lluis Berral, Jorda Polo, and David Carrera. Burst-aware predictive autoscaling for containerized microservices. *IEEE Transactions on Services Computing*, 2020.

[3] Nabeel Akhtar, Ali Raza, Vatche Ishakian, and Ibrahim Matta. Cose: Configuring serverless functions using statistical learning. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 129–138. IEEE, 2020.

[4] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC)*, pages 923–935, 2018.

[5] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 469–482, 2017.

[6] Amazon. Aws step functions documentation. https://docs.aws.amazon.com/step-functions/index.html, Last retrieved: May, 2022.

[7] Amazon. Configuring lambda function memory. https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.htmll, Last retrieved: May, 2022.

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

[9] Amazon. Parallel processing in python with aws lambda. https://aws.amazon.com/blogs/compute/parallel-processing-in-python-with-aws-lambda/, Last retrieved: May, 2022.

[10] Amazon Web Services. Parallel Processing in Python with AWS Lambda. https://aws.amazon.com/blogs/compute/parallel-processing-in-python-with-aws-lambda/, Last retrieved: May, 2022.

[11] Ganesh Ananthanarayanan, Michael Chien-Chun Hung, Xiaoqi Ren, Ion Stoica, Adam Wierman, and Minlan Yu. GRASS: Trimming stragglers in approximation analytics. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 289–302, Seattle, WA, April 2014. USENIX Association.

[12] Lixiang Ao, Liz Izhikevich, Geoffrey M Voelker, and George Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 263–274, 2018.

[13] Azure. Durable functions overview. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp, Last retrieved: May, 2022.

[14] Vivek M Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 153–167, 2021.

[15] Muhammad Bilal, Marco Serafini, Marco Canini, and Rodrigo Rodrigues. Do the best cloud configurations grow on trees? an experimental evaluation of black box algorithms for optimizing cloud workloads. *Proceedings of the VLDB Endowment*, 13(12):2563–2575, 2020.

[16] Laurent Bindschaedler, Jasmina Malicevic, Nicolas Schiper, Ashvin Goel, and Willy Zwaenepoel. Rock you like a hurricane: Taming skew in large scale analytics. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[17] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. A case for serverless machine learning. In *Workshop on Systems for ML and Open Source Software at NeurIPS*, volume 2018, 2018.

[18] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 13–24, 2019.

[19] Qi Chen, Jinyu Yao, and Zhen Xiao. Libra: Lightweight data skew mitigation in mapreduce. *IEEE Transactions on parallel and distributed systems*, 26(9):2520–2533, 2014.

[20] Mark E Crovella, Robert Frangioso, and Mor Harchol-Balter. Connection scheduling in web servers. Technical report, Boston University Computer Science Department, 1999.

[21] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st International Middleware Conference*, pages 356–370, 2020.

[22] DeepQuest-AI. Imageai : Video object detection, tracking and analysis. https://github.com/OlafenwaMoses/ImageAI/blob/master/imageai/Detection/VIDEO.md, Last retrieved: May, 2022.

[23] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[24] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. Photons: Lambdas on a diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 45–59, 2020.

[25] Simon Eismann, Long Bui, Johannes Grohmann, Cristina Abad, Nikolas Herbst, and Samuel Kounev. Sizeless: Predicting the optimal size of serverless functions. In *Proceedings of the 22nd International Middleware Conference*, pages 248–259, 2021.

[26] Simon Eismann, Johannes Grohmann, Erwin van Eyk, Nikolas Herbst, and Samuel Kounev. Predicting the costs of serverless workflows. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pages 265–276, 2020.

[27] Tarek Elgamal. Costless: Optimizing cost of serverless computing through function fusion and placement. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 300–312. IEEE, 2018.

[28] Lang Feng, Prabhakar Kudva, Dilma Da Silva, and Jiang Hu. Exploring serverless computing for neural network training. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 334–341. IEEE, 2018.

[29] Alexander Fuerst and Prateek Sharma. 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*, pages 386–400, 2021.

[30] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Mahmut Taylan Kandemir, Bhuvan Urgaonkar, George Kesidis, and Chita Das. Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 199–208. IEEE, 2019.

[31] Yanfei Guo, Jia Rao, Changjun Jiang, and Xiaobo Zhou. Moving hadoop into the cloud with flexible slot management and speculative execution. *IEEE Transactions on Parallel and Distributed systems*, 28(3):798–812, 2016.

[32] Vipul Gupta, Swanand Kadhe, Thomas Courtade, Michael W. Mahoney, and Kannan Ramchandran. Oversketched newton: Fast convex optimization for serverless systems. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 288–297, 2020.

[33] Mor Harchol-Balter, Mark E. Crovella, and Cristina D. Murta. On choosing a task assignment policy for a distributed server system. *Journal of Parallel and Distributed Computing*, 59(2):204–228, 1999.

[34] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Selecta: Heterogeneous cloud storage configuration for data analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 759–773, Boston, MA, 2018.

[35] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 427–444, 2018.

[36] YongChul Kwon, Kai Ren, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Managing skew in hadoop. *IEEE Data Eng. Bull.*, 36(1):24–33, 2013.

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

[38] Yi Lu, Qiaomin Xie, Gabriel Kliot, Alan Geller, James R. Larus, and Albert Greenberg. Join-idle-queue: A novel load balancing algorithm for dynamically scalable web services. *Performance Evaluation*, 68(11):1056–1071, 2011. Special Issue: Performance 2011.

[39] Ashraf Mahgoub, Alexander Michaelson Medoff, Rakesh Kumar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. OPTIMUSCLOUD: Heterogeneous configuration optimization for distributed databases in the cloud. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 189–203, 2020.

[40] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. SONIC: Application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 285–301, 2021.

[41] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[42] Ingo Müller, Renato Marroquín, and Gustavo Alonso. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 115–130, 2020.

[43] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 193–206, 2019.

[44] python. Chatbots: Intent recognition dataset. https://www.kaggle.com/elvinagammed/chatbots-intent-recognition-dataset, Last retrieved: May, 2022.

[45] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[46] Bianca Schroeder and Mor Harchol-Balter. Web servers under overload: How scheduling can help. *ACM Trans. Internet Technol.*, 6(1):20–52, February 2006.

[47] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC)*, pages 205–218, 2020.

[48] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. Serverless linear algebra. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 281–295, 2020.

[49] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. Prebaking functions to warm the serverless cold start. In *Proceedings of the 21st International Middleware Conference*, pages 1–13, 2020.

[50] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. Atoll: A scalable low-latency serverless platform. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 138–152, 2021.

[51] Spark. Spark speculation. https://spark.apache.org/docs/latest/configuration.html, Last retrieved: May, 2022.

[52] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. Sequoia: Enabling quality-of-service in serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 311–327, 2020.

[53] Jason Teoh, Muhammad Ali Gulzar, Guoqing Harry Xu, and Miryung Kim. Perfdebug: Performance debugging of computation skew in dataflow systems. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 465–476, 2019.

[54] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for Large-Scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, Santa Clara, CA, March 2016. USENIX Association.

[55] Hao Wang, Di Niu, and Baochun Li. Distributed machine learning with a serverless architecture. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1288–1296. IEEE, 2019.

[56] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC)*, pages 133–146, 2018.

[57] Wikipedia. Best-first search. https://en.wikipedia.org/wiki/Best-first_search, Last retrieved: May, 2022.

[58] Shanhe Yi, Zijiang Hao, Qingyang Zhang, Quan Zhang, Weisong Shi, and Qun Li. Lavea: Latency-aware video analytics on edge computing platform. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, pages 1–13, 2017.

[59] Fan Zhang, Xuxin Tang, Xiu Li, Samee U Khan, and Zhijiang Li. Quantifying cloud elasticity with container-based autoscaling. *Future Generation Computer Systems*, 98:672–681, 2019.

[60] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. Caerus: Nimble task scheduling for serverless analytics. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 653–669, 2021.

[61] Qingyang Zhang, Hui Sun, Xiaopei Wu, and Hong Zhong. Edge video analytics for public safety: A review. *Proceedings of the IEEE*, 107(8):1675–1696, 2019.

A Artifact Appendix

Abstract

This artifact appendix includes all the necessary information to reproduce the main evaluation results of the OSDI' 22 paper: ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs.

Scope and Usage

ORION is a serverless DAG optimization layer implemented in C# and Python 3.8. ORION accepts a DAG as an input and profiles the execution time for each function in the DAG as well as the entire DAG. The execution times are represented as distributions (CDFs) to capture the variability in runtimes. Afterward, users provide ORION with requirements such as a latency target (*e.g.*, P95 $\leq$ 20 seconds) and/or an upper limit on the budget (*e.g.*, cost of 1K executions $\leq$ \$1000). Next, ORION performs three optimizations to achieve user-provided requirements. The three optimizations are: (1) **Right-sizing**: Finding the best resource configurations for each function to meet the E2E latency objective with the minimum cost. (2) **Bundling**: Identifying stages where co-locating multiple parallel instances of a function together to be executed on one VM will be beneficial. The benefit arises when there is computation skew among the parallel workers caused by different content inputs and functions are scalable. (3) **Right pre-warming**: The VMs to execute the functions in the DAG are pre-warmed just right, ahead of time, so that cold starts can be avoided while keeping provider-side utilization of resources high. With these three optimizations, ORION accurately meets latency service level objectives (SLOs) while reducing execution cost. The output of ORION is a transformed DAG that has the same semantics as the user-provided DAG, but with higher performance (*i.e.*, lower latency) and lower execution cost.

Contents

1. **Benchmarks-AWS-Lambda:** This folder contains the code for the three evaluation applications (Video-Analytics, ML-Pipeline, and NLP-ChatBot). By running `deploy_application.sh` in each application directory, a DAG serverless workflow can be deployed on AWS Lambda using AWS Step Functions.

2. **DAG_Profile:** This folder contains the code for our DAG profiler. The code is generic enough to profile any application defined as a standard state machine on AWS Step Functions.

3. **DAG_Modeler:** This folder contains the code used to build the E2E performance model of the DAG. This module also contains the `VM_Size_Optimizer` to select the best VM size for each function in the DAG.

4. **Bundling_Manager:** This folder contains the code of ORION's Bundling optimization. This component of ORION profiles the DAG with varying bundle sizes and shows the P50 Latency, P95 Latency, and \$ cost for each bundle size.

5. **Prewarming_Optimizer:** This folder contains the code to select the best pre-warming delays for each stage in the DAG.

6. **Comparison_to_Baselines:** This folder contains the code that compares ORION to two baselines: Best memory and CherryPick. The script produces the tail latency and cost (in \$) for ORION as well as the two baselines.

7. **Policy_simulator:** This folder contains the code for our pre-warming policy simulator. This component compares different pre-warming policies without being constrained by what is possible in commercial public cloud.

Hosting

ORION is open sourced and we release its code, the workload characterization data, and the evaluation applications. All these components can be obtained at: https://github.com/icanforce/Orion-OSDI22

Requirements

The artifact uses AWS Lambda to host serverless functions, and AWS Step Functions to orchestrate the functions and organize them in a DAG. Some functions have large dependencies and hence are deployed as images on AWS ECR (Amazon Elastic Container Registry). Accordingly, users need to install the following dependencies:

1. **Amazon AWS CLI:** Can be obtained at: https://aws.amazon.com/cli/

2. **Docker:** Can be obtained at: https://www.docker.com/

Environment Setup

1. First, deploy one of the evaluation applications from `Benchmarks-AWS-Lambda` directory in AWS StepFunctions.

2. Then, use the `DAG_Profiler` to profile and generate the latency distributions for each function in the DAG.

3. Use `DAG_Modeler` to build the E2E performance model of the DAG, this module also contains the `VM_Size_Optimizer` to select the best VM size for each function in the DAG.

4. Use `Bundling_Manager` to select the best bundle size.

5. Use `Prewarming_Optimizer` to select the best pre-warming delays for the stages in the DAG.

6. Use `Comparison_to_Baselines` to compare Orion with CherryPick and Best Memory baselines.