Dexter: A Performance-Cost Efficient Resource Allocation Manager for Serverless Data Analytics

Anna Maria Nestorov
Barcelona Supercomputing Center,
Universitat Politècnica de Catalunya
Barcelona, Spain
anna.nestorov@bsc.es

Diego Marrón
Barcelona Supercomputing Center
Barcelona, Spain
diego.marron@bsc.es

Alberto Gutierrez-Torre
Barcelona Supercomputing Center
Barcelona, Spain
alberto.gutierrez@bsc.es

Chen Wang
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
chen.wang1@ibm.com

Claudia Misale
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
c.misale@ibm.com

Alaa Youssef
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
asyousse@us.ibm.com

David Carrera
Nearby Computing
Barcelona, Spain
dcarrera@nearbycomputing.com

Josep Lluís Berral
Universitat Politècnica de Catalunya,
Barcelona Supercomputing Center
Barcelona, Spain
josep.ll.berral@upc.edu

ABSTRACT

Leveraging serverless platforms for the efficient execution of distributed data analytics frameworks, such as Apache Spark [3], has gained substantial interest since early 2022. The elasticity, free-of-management, and on-demand scalability of serverless have motivated the effort in deploying distributed data analytics applications to serverless platforms. However, effectively auto-scaling resources for such complex workloads so that we can fully benefit from the resource elasticity of serverless remains challenging. Mis-configuration can result in severe performance and cost issues arising from resource under- and over-provisioning.

In this paper, we present Dexter, a robust resource allocation manager dynamically allocating resources at a fine-grained level to guarantee performance-cost efficiency (optimizing total runtime cost). Dexter is novel in combining predictive and reactive strategies that fully leverage the elasticity of serverless to enhance the performance-cost efficiency for workflow executions. Unlike black-box ML models, Dexter quickly reaches a sufficiently good solution, prioritizing simplicity, generality, and ease of understanding. Our experimental evaluation shows that, compared with the default serverless Spark resource allocation that dynamically requests exponentially more executors to accommodate pending tasks, our solution achieves a cost reduction of up to 4.65×, while improving performance-cost efficiency up to 3.50×. Dexter also enables a substantial resource saving, demanding up to 5.75× fewer resources.

MIDDLEWARE '24, December 2–6, 2024, Hong Kong, Hong Kong

ACM ISBN 979-8-4007-0623-3/24/12…$15.00
https://doi.org/10.1145/3652892.3700753

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

KEYWORDS

Serverless, Resource Allocation, Data Analytics, Spark, Stage

ACM Reference Format:
Anna Maria Nestorov, Diego Marrón, Alberto Gutierrez-Torre, Chen Wang, Claudia Misale, Alaa Youssef, David Carrera, and Josep Lluís Berral. 2024. Dexter: A Performance-Cost Efficient Resource Allocation Manager for Serverless Data Analytics. In *25th International Middleware Conference (MIDDLEWARE '24), December 2–6, 2024, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3652892.3700753

1 INTRODUCTION

Serverless computing [29] attracts many tenants in today's cloud environments thanks to its fundamental principles, including transparent elastic resource scaling allowing massive scalability, "pay-as-you-go" cost model with fine-grained charging granularity, and hassle-free management. While initially tailored for simple single-stage real-world applications like web services and IoT, both academic and industrial research have begun to explore its usage in more complex data analytics applications. These studies have shown that complex applications, which are typically resource-intensive and inherently parallel, benefit significantly from serverless elasticity and virtually unlimited scalability. Examples of such applications include deep learning training and inference [1, 2, 7–10, 16, 20, 22, 26, 52, 60, 65, 68], MapReduce- [28, 32, 40, 50, 71] and SQL-style analytics [25, 43, 48, 57]. The serverless market, valued at 8.93 billion in 2022, is projected to reach 55.24 billion by 2030, with a Compound Annual Growth Rate (CAGR) of 22.45% [63].

Since early 2022, a prominent effort has been made toward leveraging serverless platforms to efficiently run Spark [3] applications, which are traditionally deployed in managed cloud fixed-size clusters. GCP Dataproc Serverless [12], Databricks Serverless [11], and

IBM Analytics Engine [24] exemplify this trend. Unlike traditional Spark, which relies on a static approach requiring to specify the maximum resources prior to application execution, serverless Spark dynamically adjusts resources in real time based on workload demands. This key distinction underscores that, in contrast to conventional auto-scaling, serverless auto-scaling has virtually no bound for the number of concurrent computing instances.

In this context, a key factor for tenants and cloud providers is an automatic resource allocation guaranteeing an efficient resource utilization: even minor improvements in utilization can save millions of dollars at scale [6]. However, auto-allocating resources to complex workflows, such as data analytics applications, is challenging because the relationship between resource allocation and performance is complicated and changes over application runtime [30]. To optimize resource utilization, it is crucial to consider the diverse resource demands of application's stages by setting the right degree of parallelism (how many tasks can run in parallel) for each stage.

Recent work [27, 31, 33, 44, 48, 51, 56, 70] demonstrates the benefit of leveraging serverless for the efficient execution of distributed data analytics. However, these solutions neglect the problem of horizontal auto-scaling, statically determine the parallelism configuration, optimize resource allocation focusing solely on I/O requirements, consider only recurring applications, or fail to account for uncertainties associated with serverless execution.

Current serverless Spark frameworks, which dynamically allocate resources using First In First Out (FIFO) mappings between stages and resources, are not efficient. With fewer tasks, the default setting can: 1) Waste resources due to executor allocation overhead, as some executors might not even perform any work, and 2) See diminishing returns or even performance degradation at a higher cost. Setting the appropriate amount of resources is non-trivial, even for an expert user. Mis-configuration can lead to severe performance and cost issues due to resource under- or over-provisioning. As shown in [30], 75% of applications are over-provisioned, with 20% of them consuming more than 10× the necessary resources.

Efficient utilization of resources in serverless systems poses several challenges. First, serverless workloads often have limited or no historical data, making application profiling not always feasible. Second, finding an effective and general resource allocation policy is challenging due to the complexity of abstracting the high diversity and scale typical in production environments. Third, existing cloud approaches do not fit well with serverless Spark. Most literature focuses on optimizing resource allocation within fixed-size clusters, using cluster-wise metrics like average completion time.

In contrast, in serverless environments, resources scale dynamically, allowing each compute unit to scale independently. With a "pay by usage" pricing model, the system needs to optimize resource allocation based on both completion time and monetary cost. As a result, application performance may vary over time, influenced by changes in data sizes, or external factors such as a surge in shared storage access. Given the serverless fine-grained elasticity and the pivotal role of cluster resource allocation managers in serverless environments, the specific research question we target in this work is: *For a highly parallel data analytics workload, can we dynamically scale the amount of resources (i.e., parallelism/scale level) at per-stage granularity to balance the performance-cost tradeoff, minimizing the overall cost while providing acceptable runtime performance?*

In this work, we propose Dexter, a resource allocation manager for serverless data analytics that optimizes performance-cost efficiency by combining predictive and reactive allocation strategies. Dexter monitors each stage execution, and it is charged to allocate resources to reach the performance saturation point, accounting for the correlated monetary cost. It copes with performance-cost changes by automatically adapting allocated resources at fine-grain level, guaranteeing resource efficiency. Dexter is motivated by the lack of a serverless Spark solution, incorporating both performance and cost in the decision-making when allocating resources at the per-stage level. We fully integrated Dexter with Spark, and our evaluation shows that, compared with the current default serverless Spark dynamic resource allocation, our solution achieves a significant cost reduction of up to 4.65×, allowing up to 696 more instances (from 193 to 889 instances), while improving performance-cost efficiency up to 3.50×. Furthermore, Dexter enables a substantial resource saving, requiring up to 5.75× fewer resources, allowing to place up to 9 more workload instances (from 2 to 11) on a fixed amount of resources. Finally, Dexter provides a robust solution to new unseen workloads, achieving up to 2.87× higher performance-cost efficiency thanks to its conservative resource scaling approach.

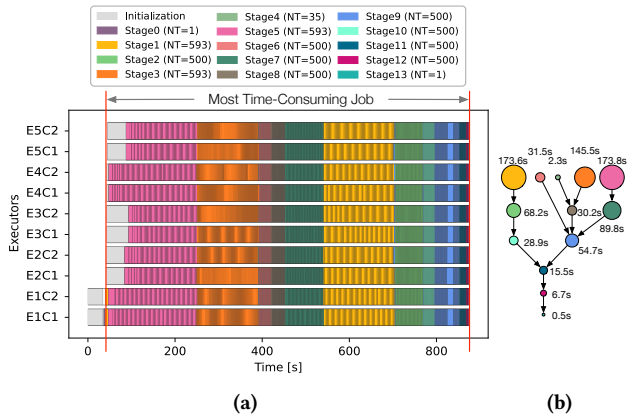
In summary, the main contributions of this paper are:

- Through a performance and cost characterization study, we highlight the effects, at different granularities, of scaling out and scaling in on execution duration and corresponding cost, showing that it is crucial to tune resource allocation at the smallest fine-grained granularity, i.e., per-stage level.
- We design Dexter, a resource allocation manager fine-tuning stage resources and enabling more efficient executions, in terms of total runtime cost, for serverless data analytics.
- To demonstrate the applicability of Dexter, we fully integrate it as a standalone and pluggable module in Spark, building a resilient and fault-tolerant system.
- We extensively evaluate Dexter to assess its effectiveness on four representative analytics benchmarking applications ranging from industry-standard benchmarks to real-world workflows. Results show that Dexter is a robust solution, reducing cost up to 4.65× while providing reasonable performance. Dexter improves performance-cost efficiency up to 3.50×, allowing up to 5.75× resource savings, enabling a higher number of deployed workload instances.

The remainder of this paper is organized as follows: Section 2 introduces serverless Spark, Dexter's design goals, and the methods used. Section 3 outlines Dexter's architecture. Section 4 describes the implementation details. Section 5 presents the experimental setup (baselines, cluster configuration, and benchmarks). Section 6 presents the system evaluation. Section 7 details the main related work. Finally, Section 8 draws the conclusions.

2 BACKGROUND AND MOTIVATION

This section first introduces Apache Spark system architecture, resource allocation, and scheduling mechanisms (Section 2.1). Next, it presents the motivations behind this work by showing how the granularity and degree of parallelism affect performance and cost (Section 2.2). Finally, it briefly reviews the search and learning methods used in this work (Section 2.3).



(a) (b)

Figure 1: TPC-H q21 query (a) sample execution schedule with dynamic allocation over a maximum of 5 executors, featuring two vCPUs each with a one-to-one core-task mapping (cold start are depicted in gray), and (b) most time-consuming job DAG reporting the average job's stages duration on 5 sequential runs and number of tasks with different circle sizes.

2.1 Apache Spark Overview

Apache Spark represents one of the most prominent multi-purpose, multi-language, in-memory big data processing frameworks, designed to process and analyze large-scale datasets implemented as immutable distributed collection of items that can be processed in parallel, the so-called Resilient Distributed Datasets (RDDs).

A Spark application is represented as a Directed Acyclic Graph (DAG), where nodes are execution stages (each comprising multiple parallel tasks) and edges are data dependencies between them. Usually, an application consists of multiple stages, defined by the dependencies between RDDs. Between different stages, it is necessary to "shuffle" the intermediate data across the network. When an application is submitted, the driver requests executors to the cluster manager, which spawns containerized Java Virtual Machines (JVMs) on worker nodes. Each executor experiences a set-up phase, known as *cold start*, involving container image downloads, container launches, and JVM initialization. Once the set-up phase has finished, stage's tasks are executed as threads in multiple "waves".

With default allocation, exponentially more executors are dynamically requested in case of backlogged tasks. While this exponential increase can facilitate the completion of an unexpected number of tasks, a notable latency exists in responding to additional executor requests. This circumstance raises the risk of either belated allocation or an exponential surplus beyond the required resources. Additionally, within a Kubernetes-based serverless setup, each node hosting an executor downloads the full application image, increasing network load and reducing per-node bandwidth. Our experiments reveal that with 50 executors spawn across 10 nodes (entailing 10 parallel image downloads, one per node), per-node bandwidth decreases by 50% compared to a single image download.

An example of TPC-H q21 query execution schedule, with the default dynamic allocation scheduling stages in FIFO fashion, is illustrated in Figure 1a. Initially, the scheduler assigns 1 executor (minimum set) and then scales out to 5 executors (maximum set) when tasks become backlogged. Among the executors, while the remaining executors experience an actual cold start, E4 executor experiences a warm start since the application image is cached because the application driver is placed on the same node. Q21 query comprises a small single-stage single-task job at the beginning, followed by a big time-consuming job (highlighted by vertical red lines in Figure 1a). As shown in Figure 1b, this most time-consuming job features a complex DAG composed of 13 stages with a variable number of tasks (i.e., NT) and duration.

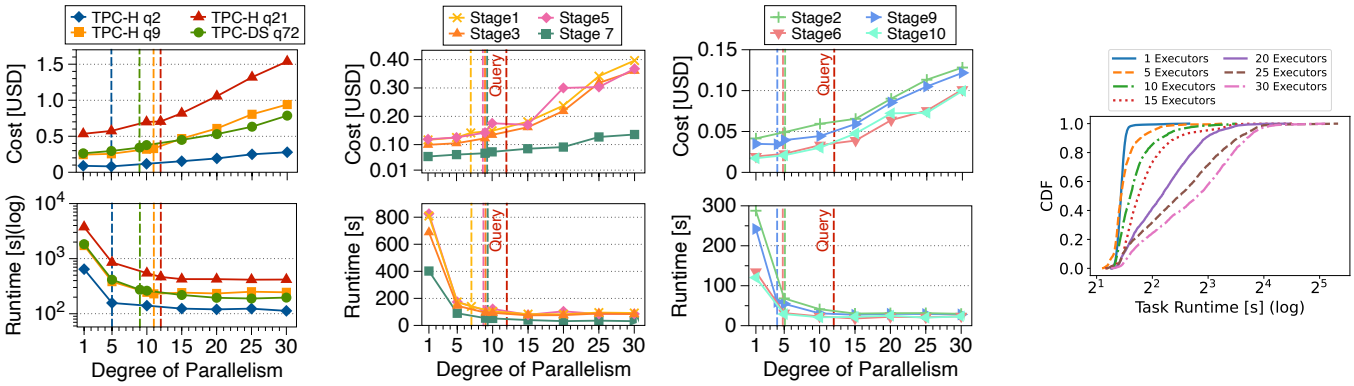
2.2 Granularity and Parallelism Analysis

Automatically adjusting resources to meet application's needs represents one of the key serverless advantages. Dynamically adapting parallelism ensures greater flexibility in resource allocation and higher resource efficiency. However, when an application is given more executors, while at the beginning performance sees a significant boost, after a saturation point, adding more resources gives either similar or even lower performance. This over-provisioning significantly impacts cost, which is dominated by the total uptime of resources. As also observed by recent works [69], once the saturation point is reached, increasing paralellism causes notable overheads due to a higher serialization and de-serialization operations, garbage collection, and intensive shuffle operations.

To highlight the effects of varying parallelism on runtime and cost and to motivate the necessity of a per-stage resource allocation, we conduct an analysis of the impact of increasing resources, i.e., number of parallel tasks, at different granularities: at application (Figure 2a), stage (Figure 2b), and task (Figure 2c) levels. During this analysis, we incrementally compare the runtime and cost at sequential degrees of parallelism and identify the saturation points, or optimal scale factors, where runtime speedup is lower than the respective cost increase beyond the assigned resources.

Figure 2a illustrates how different applications, from TPC-H and TPC-DS benchmarks, scale differently with degrees of parallelism when running on Spark with an input dataset of 100GB and each executor featuring two virtual cores and 16GB of memory. Results reveal that TPC-H q2, q9, and q21 queries exhibit strongly different scalabilities: q2 sees efficient returns of investment up to 5 executors, while q9 achieves marginal returns with no more than 11 executors, and q21 only needs 12 executors to be performance-cost efficient. Moreover, the TPC-DS q72 query shows almost the same scalability of TPC-H q9, but its saturation point occurs at a parallelism of 9 executors. It is noteworthy that 1) The saturation point tends to shift towards higher parallelism as the application runtime increases, and 2) Even if two applications show similar curves, slight variations in the respective saturation points can be observed. All the queries show a common trend in the curves, reflecting the fundamental characteristic of parallel computing: as the degree of parallelism increases, the marginal gain in performance, i.e., runtime, decreases (following an "elbow" trend) while the cost increment rises. From a performance-cost trade-off perspective, the saturation point stays between the steepy and the flattened curve region.

Similar behaviors and remarks can be observed at per-stage granularity. Figure 2b shows the runtime and cost curves for the most time-consuming stages of TPC-H q21 query. It is interesting to note that per-stage saturation points always differ from the one



(a) Queries runtimes (bottom) and costs (top). Dashed vertical lines report saturation points.

(b) TPC-H q21 query long (left) and medium (right) running stages runtimes (bottom) and costs (top). The red dashed vertical line reports the query saturation point, while the remaining lines the stages' ones.

(c) Task runtime CDF for one of the most time consuming stages in TPC-H q21 query, i.e., Stage1.

Figure 2: Influence of scaling parallelism on performance and cost at (a) per-query, (b) per-stage, and (c) per-task level.

found at application level, i.e., 12 executors. For example, allocating more than 7 executors to Stage1 yields marginal returns. This difference is even more pronounced in medium time-consuming stages: Stage9 requires at maximum 4 executors, representing a third of the application level saturation point.

Finally, we conduct a task runtime analysis for the most time-consuming stages, to investigate how the inherited overheads affect runtime when scaling resources. Figure 2c depicts the Cumulative Distribution Function (CDF) of individual tasks for one of the most time-consuming stages, i.e., Stages1 of TPC-H q21 query. We make two fundamental observations. First, scaling resources leads to a significant task runtime increase: while with 1 executor 90% of the tasks complete in less than 3.1s with a maximum of 6.9s, with 30 executors 90% of the tasks finish within 13.5s, reaching a maximum of 32.4s. When scaling-out, shuffle operations and garbage collection increase, degradating performance in terms of latency. Moreover, increasing the number of executors implies a higher number of tasks belonging to the first wave, which usually show a higher task runtime due to higher transfers of data shared across all stage's tasks (e.g., broadcast variables and task binaries), possible initializations, and loading of additional libraries. Second, the task runtime variance significantly increases with the number of executors.

Therefore, fine-tuning the amount of resources at per-stage level is crucial to avoid under- and over-provisioning, thus preventing waste of resources for negligible performance gains. This enhances the overall resource efficiency, increasing the number of workload instances deployable on a fixed amount of resources.

2.3 Search and Learning Methods

Dexter relies on the *Hill Climbing* algorithm [49], a widely used local search optimization method, to solve resource allocation optimization problems. The method starts at an initial neighbour , iteratively moving to better neighbours until an optimum is reached, beyond which no further improvement is possible with the current set of moves. The choice of the initial neighbour affects solution quality and search efficiency. Hill Climbing guarantees optimal solutions only for convex problems, while for the rest of the problems it guarantees only local optimum. However, it is a simple, effective, and intuitive algorithm that is easy to understand and represents a good choice when a sufficiently good solution is needed quickly.

In this work, to lower the number of steps in the heuristic search, we leverage historical knowledge by using Machine Learning (ML) methods. Since Dexter aims to ensure high adaptation to workload changes, the considered ML methods need to be simple, providing good balance between response time and predictive performance. In this context, we explore the following models to learn and identify a good enough initial neighbouring solution: Linear Regression (LR), Bayesian Ridge Regression (BRR), Boosted Decision Stumps (BDS), and Dropouts Additive Regression Tree (DART) [34]. While LR is a classical method that tries to fit a hyperplane to the data to describe a variable by a linear combination of the others, BRR departs from LR and includes a procedure to mitigate multicollinearity problems (high correlations). BDS, on the other hand, performs greedy function approximation [17] using decision stumps [23] as weak learner. Similarly, DART also applies a greedy function approximation, using an ensemble of boosted regression trees with dropouts as weak learner.

3 SYSTEM DESIGN

This section details Dexter, a resource allocation manager constantly monitoring each application's execution stage and automatically assigns the right amount of resources to guarantee efficient resource usage. Dexter is a significant departure from most works present in the literature, which typically rely on either reactive or predictive approaches. Instead, Dexter combines both approaches to guarantee quick adaptability to drastic workload changes. On one hand, Dexter's reactive capabilities allow the system to dynamically adjust the optimal resource allocation in case of model overfitting. On the other hand, its predictive capabilities, based on a simple ML model characterized by short retraining time, allow on-the-fly model adjustments during significant workload shifts.

Dexter's design fully leverages the virtually unlimited scalability and fine-grained elasticity of serverless platforms. Unlike cloud resource-constrained environments with a fixed pool of resources, where resource contention is common, Dexter handles each new incoming application separately. This allows applications to scale independently without waiting for resources to become available. As a result, the system can better manage resources in a multi-tenant settings where each tenant can deploy multiple applications simultaneously. In what follows, we first give an overview of the system architecture (Section 3.1), we detail Dexter's main components (Section 3.2), we describe the lifecycle of an application in our optimized serverless Spark system (Section 3.3), and finally we discuss the system's resiliency and fault tolerance (Section 3.4).

3.1 System Architecture Overview

Figure 3 shows the block diagram of the proposed system architecture, highlighting Dexter's components. The system consists of three key components: 1) The *historical module*, 2) The *search module*, and 3) The *custom executor allocation manager*. The first two together form the external *scaling agent*, which incorporates both predictive and reactive capabilities implementing the decision-making process. Meanwhile, the *custom executor allocation manager* extends Spark to dynamically allocate and remove executors based on the *scaling agent*'s decisions. At a high level, when a new stage is ready for execution (all upstream stages have completed), Spark requests and the initial resource allocation from Dexter's *historical module*. Throughout stage's execution, Spark periodically inspects Dexter's *search module* to adjust resources as needed to achieve the optimal degrees of parallelism. In particular, Dexter's *search module*'s design is performance-cost-centric, valuing completion time and cost equally. Targeting a good return on investment, if the stage performance-cost tradeoff deviates, the module adjusts the resource allocation to optimize the total runtime cost. To account for provisioning and cold start latencies, the *search module* is not invoked again until all assigned executors are up and running and an adequate amount of time has passed. This ensures that accurate stage execution information is collected under the new resource configuration. To increase resource utilization, all free executors are reallocated to active stages based on stage priority once the *scaling agent* has assigned resources to all stages.

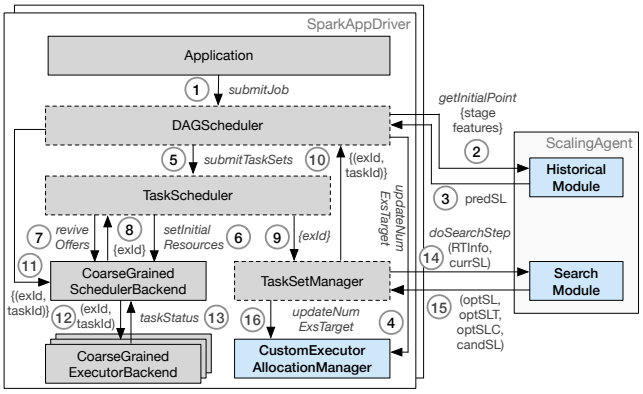


Figure 3: System architecture showing new components in light blue and modified components with dotted rectangles.

3.2 Key Components

3.2.1 Historical Module.

This section outlines our approach for the *historical module*, framing the estimation of the initial amount of resources for a given application stage as a ML regression problem. The key observation for this module is that stages of a Spark application may exhibit similar resource scalability needs (as discussed in Section 2.2), enabling the use of ML to learn from past executions to predict the initial resources for the current stage.

The *historical module* does not require user-provided information. To optimize performance and cost, it relies solely on stage attributes that are independent from the parallelism and known up front execution (compile-time). We aim to 1) Provide a good enough estimate that the *search module* can refine with minimum iterations (minimizing the total runtime), 2) Ensure bounded response times, in the order of a few microseconds per sample, and 3) Guarantee short retraining times. In this section, we compare four regression models needing low computation and re-train time while achieving good enough predictive performance, namely LR, BRR, BDS, and DART (briefly presented in Section 2.3).

The model input is a set of feature arrays, with each array containing features characterizing a certain stage, known at compile-time (e.g., total count of tasks, size of all RDD in bytes, and number of RDDs for each RDD type). We generated a dataset of 182 samples by running all 22 TPC-H queries, logging the stage's features up front stage computation (step 5 Figure 3). After removing constant features, the resulting dataset contains for every stage the optimal scaling level and the following six key features: Number of tasks, Input RDDs total size, Number of MapPartitionsRDDs, Number of ShuffledRowRDDs, Number of ZippedPartitionsRDDs, and Number of FileScanRDDs. The model's output is a real number representing the predicted optimal scaling level for a specific stage.

Model error drives model selection, as high errors in initial resource allocation estimates may disturb the *search module* convergence. We split data at query level and use Leave One Out Cross-Validation (LOOCV) to compare model errors with different data partitions. Each cross-validation iteration leaves one query out for testing and trains the model on the remaining 21 queries. The process is repeated 22 times and ensures the collection of diverse results for each model. As shown in Table 1, the analyzed models perform similarly in terms of Mean Absolute Error (MAE), with prediction errors around 1.6 executors. Given the closely comparable errors, we examine the error distributions to select the most suitable model. Although LR and BRR achieve a higher number of predictions with an error ≤ 2, we prefer models with lower maximum error since they make the *search module* deviate less from the optimal. Between BDS and DART, BDS provides slightly better percentages of errors ≤ 1. Thus, we deployed BDS to provide initial resource estimations to the *search module*, selecting the model tested on TPC-H q8 query since it features the highest number of stages.

3.2.2 Search Module.

The *search module* fine-tunes the degree of parallelism of each stage and its main target is twofolds. First, it handles imperfect resource

Table 1: Models prediction error and error distribution.

KPI	BDS	DART	LR	BRR
MAE	**1.60**	1.64	1.61	1.62
Max Error	**6**	**6**	7	7
Error ≤ 2	72.77%	70.50%	**75.00%**	74.32%
Error ≤ 1	**56.59%**	53.18%	55.36%	56.45%

prediction made by the *historical module*, potentially leading to either under- or over-estimation of resources. Second, it constantly monitors stage performance and cost metrics throughout its execution. In case the performance-cost tradeoff deviates, it adapts the number of allocated resources towards the new optimal scaling level using the Hill Climbing algorithm. Although the problem can be easily formalized as Pareto sets and fronts, the need for a sufficiently large set of samples per-application makes this approach not feasible in a serverless setup, where applications may not be used often enough to gather sufficient samples for analysis. Furthermore, the virtually unlimited resource elasticity of serverless makes challenging the normalization of performance and cost objectives space (requiring a bounded space), necessary to the Pareto optimization method to return accurate optimal solutions. While Hill Climbing algorithm does not guarantee a global optimal solution, as introduced in Section 2.3, it allows Dexter to return quickly a near-optimal solution without requiring a time-consuming profiling phase.

Algorithm 1 presents the pseudo-code of the proposed *search module*, with the top-function represented by the doSearchStep function. Each TaskSetManager periodically invokes the *search module* by calling the doSearchStep function, potentially adjusting the stage allocated resources each cycle. During each function invocation, the *search module* compares the current solution (currSL) with the optimal solution found so far (optSL). More precisely, the module initially estimates the expected runtime and cost for the current scaling level (lines 22 and 23). Given numMissingTasks missing tasks to compute for a given stage and SL available executors, each one computing $\mathsf{Ex}_{\mathsf{CPUs}}$ tasks in parallel, the number of missing task waves taskWaves is calculated as:

$$\mathtt{taskWaves} = \left\lceil \frac{\mathtt{numMissingTasks}}{\mathtt{currentlySL} \times \mathsf{Ex}_{\mathsf{CPUs}}} \right\rceil \tag{1}$$

The expected time $\mathbb{E}[T]$ is derived by multiplying the number of missing task waves, computed using Equation (1), by the average task runtime observed so far (line 2). The expected cost is then determined by adding up the two costs associated with the two components defining each Spark executor, i.e., the allocated amount of CPUs $\mathsf{Ex}_{\mathsf{CPUs}}$ and memory $\mathsf{Ex}_{\mathsf{GBs}}$ (line 8). To compute each cost component (lines 6 and 7), the module follows major cloud providers' serverless Spark billing policies. Specifically, it multiplies the expected time (expressed in hours) by the price per hour for the serverless Spark components, the number of resources, and the number of executors assigned to the stage. During the first search step, Dexter's *search module* generates and returns a new candidate solution, along with the optimal (the initial scaling level) information (lines 24-27). Otherwise, it computes the performance (line 28) and cost variation (line 29) achieved by the current solution compared to the optimal solution found so far. At this point,

Algorithm 1 Dexter's *search module* algorithm.

```
1: function computeExpTime(currSL, numMissingTasks, avgTaskTime)
2:     E[T] = ⌈ numMissingTasks / (currSL × Ex_CPUs) ⌉ × avgTaskTime
3:     return E[T]
4: end function
5: function computeExpCost(currSL, E[T])
6:     E[C_CPU] = E[T]/3600 × Price_1CPU/h × Ex_CPUs × currSL
7:     E[C_Mem] = E[T]/3600 × Price_1GB/h × Ex_GBs × currSL
8:     E[C] = E[C_CPU] + E[C_Mem]
9:     return E[C]
10: end function
11: function getCandidate(newOptSL, optSL, currSL)
12:     if |currSL − otpSL| > stepSize and otpSL < currSL then
13:         lowerBound = getMax(1, newOptSL - stepSize)
14:         searchRange = [lowerBound, newOptSL − 1]
15:     else
16:         searchRange = [newOptSL + 1, newOptSL + stepSize]
17:     end if
18:     cand = selectRandValIn(searchRange)
19:     return cand
20: end function
21: function doSearchStep(numMissingTasks, avgTaskTime, optSL, E[T]_opt,
    E[C]_opt, currSL)
22:     E[T]_curr = computeExpTime(currSL, numMissingTasks, avgTaskTime)
23:     E[C]_curr = computeExpCost(currSL, E[T]_curr)
24:     if E[T]_opt == 0.0 then
25:         candSL = getCandidate(currSL, optSL, currSL)
26:         return (optSL, E[T]_curr, E[C]_curr, candSL)
27:     end if
28:     perfVariation = E[T]_opt / E[C]_curr
29:     costVariation = E[T]_curr / E[C]_opt
30:     newOptSL = optSL
31:     if optSL <= currSL then
32:         if perfVariation > costVariation then
33:             newOptSL = currSL
34:             E[T]_opt = E[T]_curr
35:             E[C]_opt = E[C]_curr
36:         end if
37:     else
38:         perfSlowDown = |1 − perfVariation|
39:         costReduction = costVariation − 1
40:         if perfSlowDown < costReduction then
41:             newOptSL = currSL
42:             E[T]_opt = E[T]_curr
43:             E[C]_opt = E[C]_curr
44:         end if
45:     end if
46:     candSL = getCandidate(newOptSL, optSL, currSL)
47:     return (newOptSL, E[T]_opt, E[C]_opt, candSL)
48: end function
```

if resources are being increased (lines 31-37), the doSearchStep function assesses the return of investment of the current solution by comparing its performance and cost variation (line 32). Suppose the current solution is a better neighbouring solution than the known optimum solution (condition in line 32 is true). In this case, the current solution becomes the new optimum solution (line 33), and the module updates its expected runtime and cost information (lines 34 and 35). Otherwise, if the current solution does not represent an improvement, it is rejected, and the optimum solution remains unchanged. Conversely, when resources are being decreased (lines 37-45), the module computes the performance slowdown and the corresponding cost reduction (lines 38 and 39). If the performance degradation is smaller than the cost reduction (condition in line 40 is true), the current solution becomes the new optimum solution (line 41) and its expected runtime and cost information are updated

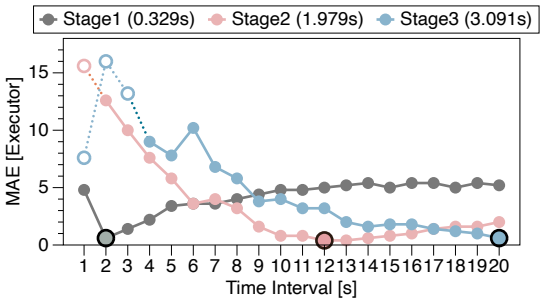


Figure 4: Time intervals performance of different stages, with varying average task runtime (within round brackets), when scaling intervals from 1s to 20s. Optimal values are reported with a black circle, while not valid time intervals (lower than the average task runtime) are reported with not filled circles.

(lines 42 and 43). Then, the module picks a new candidate neighbour based on the current optimum solution (`newOptSL`) (line 46). When selecting the new candidate, the module considers potential reassignment of free and active executors to improve the overall resource utilization. Specifically, when resources are reduced (condition in line 12 is true), the search focuses on the lower range with respect to the current solution, moving towards the optimal solution (lines 13-14), and selects a random value within this range (lines 18 and 19). This condition occurs when, to increase resource usage, free executors from parent stages are reallocated to the current stage. Conversely, when resources are increased, the module randomly selects a value from the upper range (lines 16). This design decision aims to increase resource utilization: if the candidate is set to N executors at the previous search step, testing a smaller number of executors M, i.e., $M < N$, would leave $N - M$ executors idle, reducing overall resource utilization. Finally, the module returns all information related to the current optimum solution, along with the new candidate solution (line 47).

3.2.3 *Custom Executor Allocation Manager.*

This component dynamically allocates and removes executors based on the *scaling agent*'s decisions. It maintains a target number of executors, periodically syncing to the underlying cluster manager. Every running stage TaskSetManager compares the agent-suggested candidate and the current scaling level, and it requests to increase or decrease the target if needed. More precisely, increasing the target happens when the candidate exceeds the currently allocated executors. If there are enough available active executors, there is no need to request new resources, and the *CustomExecutorAllocationManager* assigns the missing executors to the given stage from the pool of free and active executors. Otherwise, it assigns all the active executors, if any, to the stage and requests the missing ones to the cluster manager. Differently, decreasing the target number of executors happens when the candidate scaling level is more than the currently allocated resources, meaning that fewer resources are sufficient to handle the current load. In this case, idle executors are only killed after a certain amount of time (by default set to 60s), meaning that the current running stages run efficiently with the currently allocated resources.

When presented with a new application (step 1), the DAGScheduler computes the DAG of stages, keeping track of RDDs, tasks, and stage outputs, and submits stages as TaskSets. Each TaskSet contains a collection of fully independent tasks computing the same function on different data partitions. Before submitting a stage, the DAGScheduler sets its initial scale level by inspecting the *historical module* via a gRPC call, i.e., *getInitialPoint*, forwarding the stage main features (step 2), identified as detailed in Section 3.2.1. It receives back a real number, namely *predSL*, representing the predicted optimal scaling level based on historical knowledge (step 3). Once this real number is rounded down to the nearest integer (fixed to 1 if $predSL < 1$), it sets the stage scaling level to the predicted optimal scaling level and automatically requests resources to the *CustomExecutorAllocationManager* before the TaskSet is run (step 4). In turn, the *CustomExecutorAllocationManager* checks the number of free and active executors, requesting new resources to the cluster manager if necessary. Then, the DAGScheduler submits the stage TaskSet to the TaskScheduler (step 5).

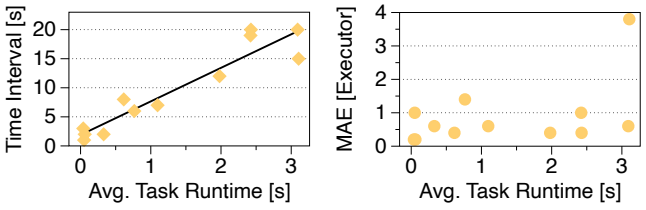
3.3 Execution Lifecycle

Upon receiving a new TaskSet with parallelism N, the TaskScheduler assesses the number of free and active executors and it then either: 1) Assigns N active executors to the TaskSet if enough are available, or 2) Assigns the available ones (if any) and submits a request for new executors to the underlying resource manager via the CoarseGrainedSchedulerBackend (step 6). For every new schedulable TaskSet, and whenever the TaskScheduler has tasks ready for scheduling, such as upon task completion, the TaskScheduler requests the list of active executors (steps 7 and 8) to the CoarseGrainedSchedulerBackend, charged to coordinate and communicate with the resource manager. The list of active executors is forwarded to the TaskSetManager. If only one TaskSet is active, the TaskScheduler makes all active executors available to that TaskSet; otherwise, the TaskScheduler allocates executors following the default stage priority order (step 9). Every TaskSetManager schedules tasks within a single TaskSet across the allocated executors, returning pairs in the form of (taskId, executorId) to the CoarseGrainedSchedulerBackend through the DAGScheduler (steps 10 and 11). Finally, the CoarseGrainedExecutorBackend launches tasks on executors, as specified in the pairs mappings (step 12), reporting task status updates to the CoarseGrainedSchedulerBackend (step 13).

Throughout the stage execution, the TaskSetManager periodically calls the *search module* through a *doSearchStep* gRPC call, exploring the search space with an additional step (step 14). The *search module* carries out this search step and returns the optimal scaling level, its related expected runtime and cost information, and a new candidate scaling level (step 15). At this point, the TaskSetManager resets to zero the task average runtime and, if the candidate scaling level, `candSL`, exceeds the current scaling level, `currSL`, it contacts the *CustomExecutorAllocationManager*, which sets the overall target number of executors accordingly and requests `currSL − candSL` new executors to the underlying resource manager (step 16).

3.4 Resiliency and Fault Tolerance

Failures in large and complex distributed systems are inevitable and can happen for many reasons, e.g., network unreliability, so



(a) Optimal time intervals and relationship with task runtime.

(b) Minimum MAEs achieved by the optimal time intervals.

Figure 5: Optimal time intervals, achieving minimum MAEs, when varying the the average task runtime.

fault tolerance plays a crucial role in preventing system-wide outages. Dexter handles transient failures, such as temporary network unavailability, temporary packet loss, or brief server unavailability, making it capable of continuing to operate in the face of outages. Dexter overcomes these failures by relying on the built-in automatic retry mechanism of Remote Procedure Call (RPC), which allow clients to retry failed calls automatically. Specifically, we have instructed the Spark *scaling agent* client to automatically retry failed gRPC calls with a maximum number of call attempts of ten, including the original attempt, and a maximum backoff delay between retry attempts of five seconds.

4 SYSTEM IMPLEMENTATION

We have implemented a complete Spark system integrating Dexter with all components detailed in Section 3.2. Kubernetes, representing the de facto standard for deploying containerized applications in cloud environments, serves as underlying resource manager for Spark. Our *scaling agent* is implemented relying on gRPC[19], a modern, high-performance, open-source universal RCP framework, accepted to Cloud Native Computing Foundation (CNCF) in 2017.

The *scaling agent* server side was implemented in approximately 150 lines of Python, and the client side required roughly 30 lines of Scala. Additionally, the protobuf file [47] defines the protocol buffer messages and services in around 20 lines. Adaptations to Spark's DAGScheduler, TaskScheduler, TaskSetManager, and Stage required roughly 50, 100, 150, and 50 lines of Scala, respectively. About 1500 lines of Scala were needed to implement the CustomExecutorAllocationManager. To store input and output data, we have deployed a MinIO server [38], a high-performance Kubernetes-native object storage tailored for large-scale systems.

5 EXPERIMENTAL SETUP

In this section we present the baseline algorithms (Section 5.1), the cluster configuration (Section 5.2), and the benchmarks that we use to assess the performance of Dexter (Section 5.3).

5.1 Baselines Algorithms

During our evaluation, we compare Dexter's performance to the following baseline algorithms. First, Spark's default dynamic allocation scheduling stages in FIFO manner, where stages are enqueued based on their order of arrival, and they get priority on all available resources. Second, Spark's dynamic allocation with FAIR scheduling gives an equal share of resources to the different runnable stages in a round-robin fashion. A fair comparison of Dexter with works in the literature is challenging due to differences in the underlying systems. Dexter relies on plain Kubernetes, whereas existing works utilize FaaS-based serverless systems, such as AWS Lambda [4], or cloud vendors' compute systems, such as Microsoft's Cosmos [46]. Unlike Kubernetes, which spawns containers, FaaS-based serverless systems enable significantly quicker function launches.

5.2 Cluster Setup

Dexter has been evaluated on an on-premise cloud deployment to avoid benchmarking cloud vendors' specific environments. To run Apache Spark v3.3.0, modified as described in Section 4, we use a virtualized Kubernetes v1.26 cluster, consisting of 1 master and 9 worker nodes. The master runs in an Ubuntu 22.04 Virtual Machine (VM) with 16 vCPUs, 120GB of memory, and 500GB of disk, while the workers run in an Ubuntu 22.04 VM with either 18 or 32 vCPUs, 120GB of memory, and 400GB of disk. While each worker node can host up to 7 executors featuring 4 vCPUs, 16GB of RAM, and 32GB of disk[1], the master node hosts the driver featuring 4 vCPUs, 16GB of memory, and 128GB of disk. Thus, we set the round number of 50 executors as the upper bound resource limit due to a lack of additional resources. The Kubernetes cluster is mapped on 10 physical nodes featuring either an Intel® Xeon Silver 4114 CPU or an Intel® Xeon E5-2630 v4 CPU. Nodes reside in the same rack and are connected through a 10Gbps Brocade VDX6740 network switch.

We deploy a MinIO server *v2023-10-14T01-57-03Z* as shared storage. It runs bare-metal on a node featuring an Intel® Xeon E5-2620 CPU, interfacing with two 1.6TB Intel® DC P3608 SSDs through NVMe. The *scaling agent* is a gRPC server v1.59.2, implemented in Python v3.11.1 using *pickle-mixin* v1.0.2 and *lightgbm* v4.1.0, and it is deployed in an Ubuntu 22.04 VM with 16 vCPUs, 32GB of memory, and 100GB of disk. The VM runs on an Intel® Xeon E5-2630 v4 CPU. The VMs and the *scaling agent* node are synchronized in the millisecond range. The experimental evaluation considers the current IBM Analytic Engine pricing plan[2].

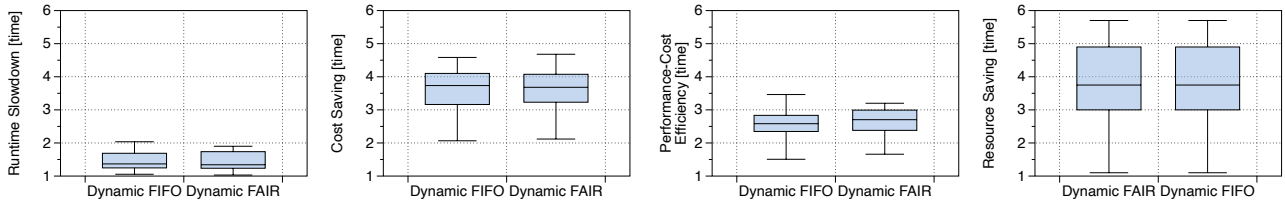
5.3 Benchmarking Applications

We have selected four benchmarking applications ranging from standard benchmarks to real-world workflows representative for data analytics frameworks, namely *TPC-H*, *TPC-DS*, *terasort*, and *page-rank*. In the following, we describe them in turn.

TPC-H. The TPC-H benchmark [62] consists of 22 business oriented ad-hoc queries, representing an industry standard benchmark for evaluating modern decision support solutions. The queries are relatively simple without pre-join and pre-aggregation of tables and with only single column indexes and operate on 8 tables with up to 16 columns. Our testbed consists of all 22 queries evaluated on a total input volume of 100GB. We do not vary the input volume size since queries show different performance characteristics and have varying input data sizes (depending on the loaded tables).

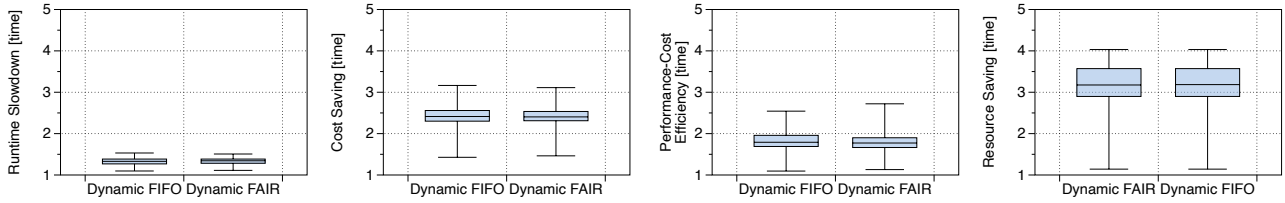
TPC-DS. The TPC-DS benchmark [61] represents another defacto industry standard benchmark, involving much more complex

[1]We mimic the executors default resource allocation configuration in serverless Spark environments in Google Cloud.
[2]0.154 USD/Virtual processor core hours and 0.0146 USD/Gigabyte hours.



(a) Runtime slowdowns. **(b) Cost savings.** **(c) Performance-cost efficiencies.** **(d) Resource savings.**

Figure 6: Dexter's results for the 22 TPC-H queries with 100GB input data (whiskers extend to min and max values).



(a) Runtime slowdowns. **(b) Cost savings.** **(c) Performance-cost efficiencies.** **(d) Resource savings.**

Figure 7: Dexter's results for the 99 TPC-DS queries with 100GB input data (whiskers extend to min and max values).

queries using wider tables compared to TPC-H. It embraces 99 query templates testing ad-hoc, reporting, interactive and iterative On-Line Analytical Processing (OLAP), and data mining queries, utilizing advanced SQL features and functions and highly applying filter predicates across 25 tables with up to 39 columns. Our testbed consists of all 99 queries evaluated on a total input volume of 100GB. Similarly to TPC-H, queries feature different performance characteristics and have varying input data sizes.

Terasort. Terasort [67] is a widely recognized benchmark used to assess the performance of sorting large volumes of data in big data processing frameworks, especially within distributed computing settings. In particular, it is used by most of the existing serverless analytics systems [5, 15, 48, 51, 53, 71] to assess their performance. Terasort operates on semi-structured data, consisting of key-value pairs. Our testbed sorts 25GB, 50GB, and 100GB of data.

Pagerank. Pagerank [41] is a graph processing algorithm widely used in web search engines to rank web pages based on the number of reference links. The benchmark input is a graph where nodes are web pages and edges are hyperlinks between pages. Pagerank is an inherently network-intensive benchmark, performing a link analysis by iteratively computing ranking scores for all pages. Our testbed ranks 10K (0.7 million edge), 20K (0.9 million edge), and 40K (1.3 million edge) nodes graph.

6 SYSTEM EVALUATION

This section presents a comprehensive evaluation of Dexter. We first analyze the impact of different search time intervals on the final solution accuracy (Section 6.1). Second, we discuss the end-to-end performance-cost efficiency and the resource usage efficiency of Dexter (Section 6.2). Third, we analyze the accuracy of our results (Section 6.3). Finally, we discuss Dexter's adaptability to unseen workloads (Section 6.4) and overheads (Section 6.5).

6.1 Dynamic Time Interval Analysis

In this analysis, we want to assess the influence of varying time intervals on the search accuracy and determine the optimal time interval for invoking Dexter search module, denoted as the time window between two consecutive calls. This analysis is based on the observation that the use of static intervals, was sub-optimal within the context of this work. The time interval needs to be: 1) Large enough to guarantee the execution of enough tasks to get accurate information, and 2) Adaptive to reflect the significant task runtime variability not only across stages but also within a given stage due to increasing parallelism overheads, as shown in Section 2.2.

The evaluation of the optimal time interval considers different task runtimes, ranging from some milliseconds up to some seconds, and different statically defined time intervals, varying from 1s up to 20s with a 1s increment. Figure 4 illustrates the MAEs for three distinct stages characterized by an average task runtime of 0.329s, 1.979s, and 3.091s, respectively. Notably, each stage exhibits a different optimal time interval - 2s, 12s, and 20s - achieving an error of 0.6, 0.4, and 0.6 executors, respectively. To define the dynamic time interval as a function of the average task runtime, we include a higher number of stages ranging from some milliseconds up to roughly 3s. Figure 5a and Figure 5b depict the optimal time intervals and their corresponding MAEs, respectively. Although optimal time intervals result in MAEs lower than 1.4 executors, there is a notable exception, where the MAE reaches 3.8 executors. Closer analysis reveals that the executors in this stage are affected by multiple long cold starts; therefore, the stage does not have enough time to scale appropriately. In contrast, another stage with an almost equivalent average task runtime shows a MAE of 0.6 executors. The relationship between the average task runtime and the optimal time interval, i.e., the one achieving minimum MAE, can be modeled by a linear regression model, as highlighted in Figure 5a. Specifically, given an average task runtime ATR, the dynamic time interval DTI

can be modeled as the linear relationship defined as:

$$DTI(ATR) = 5.77 \times ATR + 1.89 \quad (2)$$

Therefore, we configured Spark to periodically call Dexter's *search module* with a dynamic time window defined following Equation (2).

6.2 End-to-End Performance Evaluation

In this evaluation, we assess the performance-cost efficiency, and effectiveness of Dexter's improved resource utilization compared to the default Spark dynamic resource allocation strategies assigning resources in a FIFO and FAIR fashion. Differently from other works in the literature, focusing on optimizing the sharing of a fixed amount of resources among a set of workloads, we study Dexter's performance-cost tradeoff under the assumption that every incoming workload can independently scale. This assumption stems from the serverless paradigm, where each application can leverage a theoretically unlimited pool of resources. In our custom implementation of serverless Spark, the gRPC server and the MinIO object storage represent the two potential primary system bottlenecks. While the gRPC server can efficiently deal with high loads, distributing the requests optimally across a set of server instances, object storage has been proved to be a practical but powerful approach for serverless data analytics when considering optimized versions [39, 48, 51, 53, 54]. Therefore, since implementing an optimized object storage is out of the scope of this work, in this evaluation, we assume that neither the gRPC server nor the MinIO object storage imposes constraints on system performance.

We assess the effect of dynamically adjusting the amount of resources at each stage by adopting a hybrid approach that combines predictive with reactive approaches. Precisely, for each query, we measure two primary metrics: 1) The end-to-end runtime, defined as the temporal span between the timestamp preceding the first executor request and the timestamp immediately following the shutdown of the last executor (including executors' cold start time), and 2) The associated monetary cost, calculated following the serverless paradigm, accounting solely for executors uptimes while not accounting for cold start. To derive an unbiased evaluation of the overall performance-cost tradeoff, we introduce a single composite metric denoted as *efficiency*, defined as:

$$Efficiency = \frac{Runtime_{Baseline} \times Cost_{Baseline}}{Runtime_{Dexter} \times Cost_{Dexter}} \quad (3)$$

By incorporating both runtime and cost variables, the efficiency metric avoids biased solutions by reflecting any variation in terms of latency, cost, or a combination of the two. Figure 6 illustrates the results obtained from benchmarking the TPC-H dataset.

Notably, Dexter scales resources less aggressively than the two baselines, leading to comparable or slightly higher end-to-end runtimes. This increases latency by a factor ranging from 1.03× to 2.02×, and from 1.05× to 1.90× for the two baselines, respectively. However, from a pure cost perspective, Dexter always costs significantly less than the two baselines, reducing costs ranging from 2.02× to 4.59× and from 2.07× to 4.65× on each baseline. Across all analyzed queries, Dexter demonstrates improved performance-cost efficiency, ranging from 1.50× to 3.50× for the FIFO baseline and from 1.62× to 3.20× for the FAIR baseline. The resource allocation patterns for TPC-H q7 query, showing the maximum efficiency, are

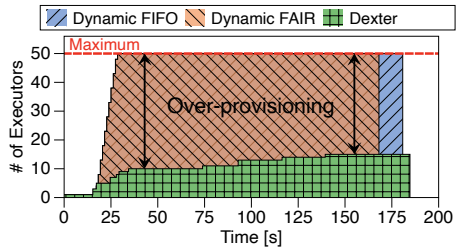


Figure 8: Resource allocation patterns for TPC-H q7 query.

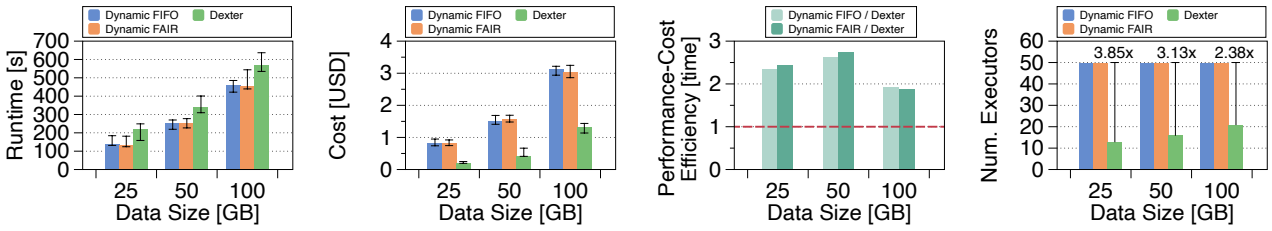
illustrated in Figure 8. Dexter has a more conservative approach on scaling resources, achieving comparable or slightly higher runtimes, while using up to 3.33× fewer resources.

Scaling at per-stage level allows to free resources for concurrent workload executions. As shown in Figure 6d, in contrast to the two baselines using 50 executors, Dexter consistently achieves resource savings ranging from 1.13×to 5.75×. Figure 8 presents the resources allocated by the three analyzed solutions in the case of TPC-H q7 query, reaching a resource saving of 3.33×, which translates in using only 15 executors, thus leaving 35 executors.

Finally, we analyze the number of deployable instances under various constraints, leveraging the three different resource allocation strategies. We constrain three aspects: 1) Amount of available resources, 2) Time availability, and 3) Monetary budget. Notably, on query q14, representing the query achieving the most significant resource saving, Dexter enables the deployment of $\lfloor \frac{100}{9} \rfloor = 11$ concurrent instances with 100 executors, compared to a maximum of $\lfloor \frac{100}{50} \rfloor = 2$ concurrent instances achievable by the baselines. In the worst case with query q13, our solution deploys $\lfloor \frac{3600}{59.81} \rfloor - \lfloor \frac{3600}{120.99} \rfloor = 60 - 29 = 31$ (2.02×) and $\lfloor \frac{3600}{63.99} \rfloor - \lfloor \frac{3600}{120.99} \rfloor = 56 - 29 = 27$ (1.90×) less instances compared to the baselines. Therefore, as expected, when focusing only on the runtime metric, Dexter deploys fewer instances than the two baselines. Lastly, when considering a monetary budget constraint of 100$, due to the significantly higher cost efficiency, Dexter yields a considerable increase in deployable instances. For the most cost-efficient query q1, Dexter enables $\lfloor \frac{100}{0.1124} \rfloor - \lfloor \frac{100}{0.5161} \rfloor = 889 - 193 = 696$ (4.60×) and $\lfloor \frac{100}{0.1124} \rfloor - \lfloor \frac{100}{0.5108} \rfloor = 889 - 195 = 694$ (4.55×) more instances compared to the baselines. In summary, our fine-grained per-stage scaling approach significantly outperforms the baselines when constraining the amount of resources and the monetary cost. In contrast, the baselines perform better when constraining the time availability since they scale more aggressively.

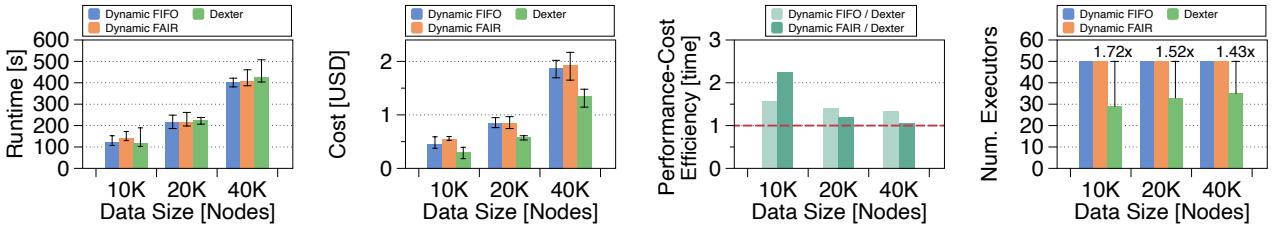
6.3 Parallelism Accuracy Analysis

While the individual accuracies of the *historical* and *search module* have been provided in Section 3.2.1 and Section 3.2.2, we now evaluate the overall system accuracy by analyzing the absolute error distribution. The analysis reveals a median absolute error of 4, with the interquartile range spanning from 1 to 8 executors. The minimum and maximum absolute errors are 0 and 18.5 executors, respectively. The large maximum error results from the free executor reassignment policy, implemented to enhance the overall resource utilization. However, this reassignment speeds up the



(a) Runtime performance. (b) Monetary cost. (c) Performance-cost efficiencies. (d) Resource usage.

Figure 9: Dexter's results for terasort with 25GB, 50GB, and 100GB input data (error bars show the min and max values).



(a) Runtime performance. (b) Monetary cost. (c) Performance-cost efficiencies. (d) Resource usage.

Figure 10: Dexter's results for pagerank with 10K, 20K, and 40K node graph (error bars show the min and max values).

stage completion, hindering Dexter's efficient resources scaling and preventing the search process from converging toward the actual stage optimal degree of parallelism. Given this distribution of absolute errors, we can conclude that Dexter gives good accuracy while achieving significantly higher efficiency, as reported in Section 6.2.

6.4 Input Workload Sensitivity Analysis

We now conduct a sensitivity analysis on the robustness of Dexter when presented with new, unseen input workloads, simulating a workload change. The primary research question we pose is: *Is our agent trained on the TPC-H benchmark capable of achieving efficient resource allocations for previously unseen workloads featuring radically different input datasets?* To address this question, we selected three representative data analytics benchmarks, namely TPC-DS, terasort, and pagerank. These benchmarks are characterized by different compute and communication requirements, as well as input data size, which renders the efficient parallelization challenging.

As shown in Figure 7, when considering the 99 TPC-DS queries, Dexter provides reasonable performance, while reducing cost up to 3.16× and 3.11×, increasing performance-cost efficiencies up to 2.54× and 2.72×, and saving up to 4.03× for the two baselines, accordingly. For terasort, Figure 9 shows that Dexter reduces cost up to 3.79× and 3.77× for small to large data sizes, without affecting runtime significantly. Moreover, performance-cost efficiency is increased by up to 2.78× and 2.87×, while freeing up to 3.85× resources. For pagerank, as presented in Figure 10, our solution allows to achieve comparable performance, while saving up to 1.52× and 1.82× monetary cost, enabling up to 1.57× and 1.51× improved performance-cost efficiencies, and using up to 1.72× fewer resources. The results of this analysis prove that Dexter is robust to workload changes with heterogeneous data and variable data sizes.

6.5 System Overhead Evaluation

In this evaluation, we discuss Dexter's overhead in terms of time necessary to fit the *historical module* model and infer the initial stage parallelism. We also discuss the time taken to perform one search step when fine-tuning the parallelism during stage computation. The reported times represent end-to-end latency, calculated as the difference between the timestamp prior to each *scaling agent* call and the timestamp after the corresponding response reception.

For the initial parallelism prediction, fitting the BDS model on a batch of 182 stages features takes, on average, 19.12ms using a single thread. This latency includes the time necessary to read the BDS model, stored as a 16K pickle file. In a multi-tenant cloud-like scenario, with a dataset scaled up to 18,200 rows (two orders of magnitude larger), the training would take an average of $19.12\text{ms} \times 10^2 = 1,912\text{ms} \approx 2\text{s}$. We emphasize that the training delay remains small enough to seamlessly execute model retraining on the fly, even at per-minute granularity. The one-time per-stage prediction of the initial degree of parallelism averages to 15.87ms.

The average delay for each search step performed by the *search module* amounts to 5.65ms. Since the total search time for a stage strictly depends on the number of performed search steps, which in turn depends on the average task runtime, to assess the impact of search delay we consider the three different stages discussed in Section 6.1, featuring an average task runtime of 0.329s, 1.979s, and 3.091s, respectively. During stage execution, the agent is invoked once, twice, and four times, translating into a median total search latency of 5.5ms, 6.5ms, and 26.5ms, accordingly.

Given the delays from Dexter's predictive and reactive components, we conclude that Dexter's effective resource allocation adds negligible overhead to the overall execution time.

7 RELATED WORK

Cloud Services and Optimizations: Cloud service providers' production systems, such as Google Borg [64], and open source systems, such as Google Omega [55], typically require manual configuration to control resource allocation. This requirement breaks the free-of-management serverless principle, making these solutions not applicable to serverless environments.

While literature extensively covers resource management systems for data analytics in serverful cloud data centers [13, 14, 18, 21, 30, 37, 42, 56, 57, 59], these approaches do not fit well with the serverless paradigm. They either optimize application resource sharing of a fixed amount of resources, focus on a specific subset of applications, or require a profiling phase.

Serverless Services and Optimizations: As detailed in Section 1, serverless Spark services leverage Spark's dynamic resource allocation. This default allocation strategy can lead to severe performance and cost issues due to mis-configurations. Additionally, these systems are fully reactive, completely neglecting potential optimization based on workload characteristics.

Various research works propose solutions for optimizing serverless data analytics. A summmary of these works appears in Table 2, where we compare them with Dexter across five dimensions: 1) *horizontal auto-scaling*, 2) *cost awareness*, 3) *considered workloads*, 4) *granularity level*, and 5) type of *parallelism*. Caerus [70] captures sub-task level pipelinability and data dependencies, scheduling tasks at just the right time. Like Dexter, Caerus tackles the performance-cost tradeoff in serverless platforms leveraging both historical and runtime information. However, Caerus sidesteps the problem of optimizing horizontal auto-scaling. Ditto [27]analyses the application's DAG, groups stages by data dependencies and I/O characteristics, and schedules functions at group granularity. Similarly to Dexter, Ditto highlights the importance of resource demand diversity across stages. While Dexter dynamically determines each stage's parallelism using solely compile-time and runtime stage information, Ditto determines this configuration statically, relying on parallelized time characteristics of the stage.

FaasFlow [35] proposes a decentralized workflow scheduling pattern, WorkerSP, designed to reduce scheduling overhead by enabling co-located functions to communicate via shared main memory. SONIC [36] is a data-passing manager leveraging a simple regression model to transparently select the optimal data-passing method and implementing communication-aware function placement. Although FaasFlow and SONIC also address performance and cost optimization on serverless platforms, they are orthogonal to Dexter since they focus on minimizing data-passing latency rather than tackling the resource auto-scaling problem.

Seer [51] is a shuffle manager dynamically selecting the optimal amount of resources and shuffle implementation based on an analytical model aiming to maximize I/O efficiency to object storage. Locus [48] is a shuffle analytical system optimizing the performance-cost trade-off by combining slow yet cheap storage with fast yet expensive storage. Pocket [33] develops a multi-tier storage approach automatically rightsizing resources to meet application I/O requirements while minimizing cost for intermediate data passing. These works specifically study the performance of intermediate data shuffling, essential for operations such as `GROUPBY` or `JOIN`, scaling resources solely on data shipping information. In contrast, Dexter offers a broader approach based on global stage-level characteristics and runtime statistics.

Table 2: Serverless related work and comparison with Dexter.

Work	Horizontal Auto-Scaling	Cost Awareness	Considered Workloads	Granularity Level	Parallelism
Caerus [70]	✗	✓	all	task	-
Ditto [27]	✓	✓	all	group of stages	static
FaasFlow [35]	✗	✗	all	task	-
SONIC [36]	✗	✓	all	stage	-
Seer [51]	✓	✓	all	stage	static
Locus [48]	✓	✓	all	application	static
Pocket [33]	✓	✓	all	application	static
TASQ [44]	✓	✓	all	application	static
AutoToken [56]	✓	✗	recurring	application	static
Kassing et al. [31]	✓	✓	queries	application	static
Dexter	✓	✓	all	stage	dynamic

TASQ [44] presents an ML-based approach, modeling the relationship between application runtime and allocated resources to select the optimal resources for each query in Microsoft's SCOPE [45]. Like Dexter, TASQ predicts application runtime using compile-time features. However, resource allocation is statically done upfront application execution. Furthermore, TASQ's model training time is significantly longer compared to our solution, taking 913s for a single epoch. Similarly, AutoToken [56] also considers a set of application features before dispatch to predict the peak resource usage of recurring applications. However, it builds specialized models for a subset of workloads (recurring applications), not predicting allocations for non-recurring applications.

Finally, Kassing et al. [31] formulates a model to estimate runtime and cost, automatically identifying resource configurations striking a good balance. The model has several parameters accounting for key serverless aspects, such as start-up and computation latencies. However, estimating these parameters is challenging due to the inherent uncertainties in serverless [58, 66]. Thus, it is unclear whether model's parameters align with serverless dynamic nature.

8 CONCLUSIONS

This work presents Dexter, a robust resource allocation manager that continuously monitors stage execution, automatically assigning the right amount of resources to maximize resource utilization. Our experimental evaluation shows that on a wide range of analytics workloads, compared with the default serverless Spark resource allocation, Dexter achieves a cost reduction of up to 4.65×, while improving performance-cost efficiency up to 3.50× and saving up to 5.75× resources. In future work, we plan to extend Dexter to accommodate user-defined performance and cost priorities.

ACKNOWLEDGMENTS

We thank Marc Sánchez-Artigas, our shepherd Jérémie Decouchant, and the reviewers for their feedback on earlier versions of this manuscript. This work is financed by the EU-HORIZON programme under grant agreements EU-HORIZON GA.101092646, EU-HORIZON MSCA GA.101086248, by Generalitat de Catalunya (AGAUR) GA.2021-SGR-00478, and the Spanish Ministry of Science (MICINN), the Research State Agency (AEI) and European Regional Development Funds (ERDF/FEDER) PID2021-126248OB-I00, MCIN/AEI/10.13039/ 501100011033/FEDER, UE.

REFERENCES

[1] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: Machine learning inference serving on serverless platforms with adaptive batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2020. doi: 10.1109/SC41405.2020.00073.

[2] Amazon. URL https://aws.amazon.com/it/blogs/machine-learning/code-free-machine-learning-automl-with-autogluon-amazon-sagemaker-and-aws-lambda/.

[3] Apache Spark: Unified engine for large-scale data analytics. URL https://spark.apache.org.

[4] AWS Lambda. URL https://aws.amazon.com/lambda/.

[5] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference*, Middleware '19, page 41–54, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450370097. doi: 10.1145/3361525.3361535. URL https://doi.org/10.1145/3361525.3361535.

[6] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[7] Anirban Bhattacharjee, Yogesh Barve, Shweta Khare, Shunxing Bao, Aniruddha Gokhale, and Thomas Damiano. Stratum: A serverless framework for the lifecycle management of machine learning-based data analytics tasks. In *2019 USENIX Conference on Operational Machine Learning (OpML 19)*, pages 59–61, Santa Clara, CA, May 2019. USENIX Association. ISBN 978-1-939133-00-7. URL https://www.usenix.org/conference/opml19/presentation/bhattacharjee.

[8] Anirban Bhattacharjee, Ajay Dev Chhokra, Zhuangwei Kang, Hongyang Sun, Aniruddha Gokhale, and Gabor Karsai. BARISTA: efficient and scalable serverless serving system for deep learning prediction services. *CoRR*, abs/1904.01576, 2019. URL http://arxiv.org/abs/1904.01576.

[9] Joao Carreira. A case for serverless machine learning. 2018.

[10] 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*, SoCC '19, page 13–24, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369732. doi: 10.1145/3357223.3362711. URL https://doi.org/10.1145/3357223.3362711.

[11] Databricks SQL Serverless. URL https://www.databricks.com/blog/announcing-general-availability-databricks-sql-serverless.

[12] Dataproc Serverless. URL https://cloud.google.com/dataproc-serverless/docs.

[13] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. *SIGPLAN Not.*, 49(4):127–144, feb 2014. ISSN 0362-1340. doi: 10.1145/2644865.2541941. URL https://doi.org/10.1145/2644865.2541941.

[14] Stratos Dimopoulos, Chandra Krintz, and Rich Wolski. Justice: A deadline-aware, fair-share resource allocator for implementing multi-analytics. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 233–244, 2017. doi: 10.1109/CLUSTER.2017.52.

[15] Jonatan Enes, Roberto R. Expósito, and Juan Touriño. Real-time resource scaling platform for big data workloads on serverless environments. *Future Generation Computer Systems*, 105:361–379, 2020. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2019.11.037. URL https://www.sciencedirect.com/science/article/pii/S0167739X19310015.

[16] 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, 2018. doi: 10.1109/CLOUD.2018.00049.

[17] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *Ann. Statist.*, 29(5):3326–3339, aug 2001. ISSN 1189-1232. doi: 10.1214/aos/1013203451. URL https://doi.org/10.1023/A:1010933404324.

[18] Han Gao, Zhengyu Yang, Janki Bhimani, Teng Wang, Jiayin Wang, Bo Sheng, and Ningfang Mi. Autopath: Harnessing parallel execution paths for efficient resource allocation in multi-stage big data frameworks. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9, 2017. doi: 10.1109/ICCCN.2017.8038381.

[19] gRPC. URL https://grpc.io.

[20] Vipul Gupta, Swanand Kadhe, Thomas A. Courtade, Michael W. Mahoney, and Kannan Ramchandran. Oversketched newton: Fast convex optimization for serverless systems. *CoRR*, abs/1903.08857, 2019. URL http://arxiv.org/abs/1903.08857.

[21] Rui Han, Chi Harold Liu, Zan Zong, Lydia Y. Chen, Wending Liu, Siyi Wang, and Jianfeng Zhan. Workload-adaptive configuration tuning for hierarchical cloud schedulers. *IEEE Transactions on Parallel and Distributed Systems*, 30(12):2879–2895, 2019. doi: 10.1109/TPDS.2019.2923197.

[22] Joseph M. Hellerstein, Jose M. Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *CoRR*, abs/1812.03651, 2018. URL http://arxiv.org/abs/1812.03651.

[23] Wayne Iba and Pat Langley. Induction of one-level decision trees. In Derek Sleeman and Peter Edwards, editors, *Machine Learning Proceedings 1992*, pages 233–240. Morgan Kaufmann, San Francisco (CA), 1992. ISBN 978-1-55860-247-2. doi: https://doi.org/10.1016/B978-1-55860-247-2.50035-8. URL https://www.sciencedirect.com/science/article/pii/B9781558602472500358.

[24] IBM Analytics Engine. URL https://cloud.ibm.com/docs/AnalyticsEngine?topic=AnalyticsEngine-getting-started.

[25] Jananie Jarachanthan, Li Chen, Fei Xu, and Bo Li. Astrea: Auto-serverless analytics towards cost-efficiency and qos-awareness. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):3833–3849, 2022. doi: 10.1109/TPDS.2022.3172069.

[26] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. Towards demystifying serverless machine learning training. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 857–871, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383431. doi: 10.1145/3448016.3459240. URL https://doi.org/10.1145/3448016.3459240.

[27] Chao Jin, Zili Zhang, Xingyu Xiang, Songyun Zou, Gang Huang, Xuanzhe Liu, and Xin Jin. Ditto: Efficient serverless analytics with elastic parallelism. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 406–419, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702365. doi: 10.1145/3603269.3604816. URL https://doi.org/10.1145/3603269.3604816.

[28] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 445–451, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350280. doi: 10.1145/3127479.3128601. URL https://doi.org/10.1145/3127479.3128601.

[29] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, João Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. *CoRR*, abs/1902.03383, 2019. URL http://arxiv.org/abs/1902.03383.

[30] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards automated SLOs for enterprise clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 117–134, Savannah, GA, November 2016. USENIX Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/jyothi.

[31] Simon Kassing, Ingo Müller, and Gustavo Alonso. Resource allocation in serverless query processing, 2022.

[32] Youngbin Kim and Jimmy Lin. Serverless data analytics with flint. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 451–455, 2018. doi: 10.1109/CLOUD.2018.00063.

[33] 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 18)*, pages 427–444, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/klimovic.

[34] Rashmi Korlakai Vinayak and Ran Gilad-Bachrach. DART: Dropouts meet Multiple Additive Regression Trees. In Guy Lebanon and S. V. N. Vishwanathan, editors, *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*, volume 38 of *Proceedings of Machine Learning Research*, pages 489–497, San Diego, California, USA, 09–12 May 2015. PMLR. URL https://proceedings.mlr.press/v38/korlakaivinayak15.html.

[35] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. Faasflow: enable efficient workflow execution for function-as-a-service. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 782–796, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392051. doi: 10.1145/3503222.3507717. URL https://doi.org/10.1145/3503222.3507717.

[36] 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. USENIX Association, July 2021. ISBN 978-1-939133-23-6. URL https://www.usenix.org/conference/atc21/presentation/mahgoub.

[37] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. *CoRR*, abs/1810.01963, 2018. URL http://arxiv.org/abs/1810.01963.

[38] MinIO Object Storage. URL https://min.io.

[39] Ingo Müller, Renato Marroquín, and Gustavo Alonso. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. *CoRR*, abs/1912.00937, 2019. URL http://arxiv.org/abs/1912.00937.

[40] 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*, SIGMOD '20, page 115–130, New York, NY, USA, 2020. Association for Computing Machinery.

ISBN 9781450367356. doi: 10.1145/3318464.3389758. URL https://doi.org/10.1145/3318464.3389758.

[41] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford Digital Library Technologies Project, 1998. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.31.1768.

[42] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A. Kozuch, and Gregory R. Ganger. 3sigma: Distribution-based cluster scheduling for runtime uncertainty. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355841. doi: 10.1145/3190508.3190515. URL https://doi.org/10.1145/3190508.3190515.

[43] Matthew Perron, Raul Castro Fernandez, David J. DeWitt, and Samuel Madden. Starling: A scalable query engine on cloud function services. *CoRR*, abs/1911.11727, 2019. URL http://arxiv.org/abs/1911.11727.

[44] Anish Pimpley, Shuo Li, Anubha Srivastava, Vishal Rohra, Yi Zhu, Soundararajan Srinivasan, Alekh Jindal, Hiren Patel, Shi Qiao, and Rathijit Sen. Optimal resource allocation for serverless queries. arXiv, July 2021. URL https://www.microsoft.com/en-us/research/publication/optimal-resource-allocation-for-serverless-queries/.

[45] Conor Power, Hiren Patel, Alekh Jindal, Jyoti Leeka, Bob Jenkins, Michael Rys, Ed Triou, Dexin Zhu, Lucky Katahanas, Chakrapani Bhat Talapady, Joshua Rowe, Fan Zhang, Rich Draves, Marc Friedman, Ivan Santa Maria Filho, and Amrish Kumar. The cosmos big data platform at microsoft: over a decade of progress and a decade to look forward. *Proc. VLDB Endow.*, 14(12):3148–3161, jul 2021. ISSN 2150-8097. doi: 10.14778/3476311.3476390. URL https://doi.org/10.14778/3476311.3476390.

[46] Conor Power, Hiren Patel, Alekh Jindal, Jyoti Leeka, Bob Jenkins, Michael Rys, Ed Triou, Dexin Zhu, Lucky Katahanas, Chakrapani Bhat Talapady, et al. The cosmos big data platform at microsoft: Over a decade of progress and a decade to look forward. *Proceedings of the VLDB Endowment*, 14(12):3148–3161, 2021.

[47] Protocol buffers. URL https://protobuf.dev;https//github.com/protocolbuffers/protobuf.

[48] 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 19)*, pages 193–206, Boston, MA, February 2019. USENIX Association. ISBN 978-1-931971-49-2. URL https://www.usenix.org/conference/nsdi19/presentation/pu.

[49] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Pearson, 2016.

[50] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. Serverless data analytics in the ibm cloud. In *Proceedings of the 19th International Middleware Conference Industry*, Middleware '18, page 1–8, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450360166. doi: 10.1145/3284028.3284029. URL https://doi.org/10.1145/3284028.3284029.

[51] Marc Sánchez-Artigas and Germán T. Eizaguirre. A seer knows best: Optimized object storage shuffling for serverless analytics. In *Proceedings of the 23rd ACM/IFIP International Middleware Conference*, Middleware '22, page 148–160, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393409. doi: 10.1145/3528535.3565241. URL https://doi.org/10.1145/3528535.3565241.

[52] Marc Sánchez-Artigas and Pablo Gimeno Sarroca. Experience paper: Towards enhancing cost efficiency in serverless machine learning training. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, page 210–222, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385343. doi: 10.1145/3464298.3494884. URL https://doi.org/10.1145/3464298.3494884.

[53] Marc Sánchez-Artigas, Germán T. Eizaguirre, Gil Vernik, Lachlan Stuart, and Pedro García-López. Primula: A practical shuffle/sort operator for serverless computing. In *Proceedings of the 21st International Middleware Conference Industrial Track*, Middleware '20, page 31–37, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450382014. doi: 10.1145/3429357.3430522. URL https://doi.org/10.1145/3429357.3430522.

[54] Pablo Gimeno Sarroca and Marc Sánchez-Artigas. On data processing through the lenses of s3 object lambda. In *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, pages 1–10, 2023. doi: 10.1109/INFOCOM53939.2023.10228890.

[55] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 351–364, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450319942. doi: 10.1145/2465351.2465386. URL https://doi.org/10.1145/2465351.2465386.

[56] Rathijit Sen, Alekh Jindal, Hiren Patel, and Shi Qiao. Autotoken: Predicting peak parallelism for big data analytics at microsoft. *Proc. VLDB Endow.*, 13(12): 3326–3339, aug 2020. ISSN 2150-8097. doi: 10.14778/3415478.3415554. URL https://doi-org.recursos.biblioteca.upc.edu/10.14778/3415478.3415554.

[57] Rathijit Sen, Abhishek Roy, and Alekh Jindal. Predictive price-performance optimization for serverless query processing. *CoRR*, abs/2112.08572, 2021. URL https://arxiv.org/abs/2112.08572.

[58] Mohammad Shahrad, Rodrigo Fonseca, Inigo 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 20)*, pages 205–218. USENIX Association, July 2020. ISBN 978-1-939133-14-4. URL https://www.usenix.org/conference/atc20/presentation/shahrad.

[59] Subhajit Sidhanta, Wojciech Golab, and Supratik Mukhopadhyay. Optex: A deadline-aware cost optimization model for spark. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 193–202, 2016. doi: 10.1109/CCGrid.2016.10.

[60] Vikram Sreekanti, Harikaran Subbaraj, Chenggang Wu, Joseph E. Gonzalez, and Joseph M. Hellerstein. Optimizing prediction serving on low-latency serverless dataflow. *CoRR*, abs/2007.05832, 2020. URL https://arxiv.org/abs/2007.05832.

[61] TPC-DS Benchmark. URL https://www.tpc.org/tpcds/.

[62] TPC-H Benchmark. URL https://www.tpc.org/tpch/.

[63] Verified Market Research. URL https://www.verifiedmarketresearch.com/product/serverless-architecture-market/.

[64] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450332385. doi: 10.1145/2741948.2741964. URL https://doi.org/10.1145/2741948.2741964.

[65] 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, 2019. doi: 10.1109/INFOCOM.2019.8737391.

[66] 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 18)*, pages 133–146, Boston, MA, July 2018. USENIX Association. ISBN ISBN 978-1-939133-01-4. URL https://www.usenix.org/conference/atc18/presentation/wang-liang.

[67] Owen O'Malley Yahoo! Terabyte sort on apache hadoop. Mai 2008 2008. URL http://www.hpl.hp.com/hosted/sortbenchmark/YahooHadoop.pdf.

[68] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. Infless: a native serverless system for low-latency, high-throughput inference. pages 768–781, 02 2022. doi: 10.1145/3503222.3507709.

[69] Hanfei Yu, Hao Wang, Jian Li, and Seung-Jong Park. Harvesting idle resources in serverless computing via reinforcement learning. *CoRR*, abs/2108.12717, 2021. URL https://arxiv.org/abs/2108.12717.

[70] 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 21)*, pages 653–669. USENIX Association, April 2021. ISBN 978-1-939133-21-2. URL https://www.usenix.org/conference/nsdi21/presentation/zhang-hong.

[71] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. Caerus: Nimble task scheduling for serverless analytics. In *Symposium on Networked Systems Design and Implementation*, 2021.