

Stage-Level Executor Allocation in Apache Spark™ with Cost-Performance Trade-offs

Miriam Rateike
miriam.rateike@ibm.com
IBM, University of Tübingen
Nairobi, Kenya

Isaac Waweru Wambugu
isaacw@ke.ibm.com
IBM
Nairobi, Kenya

Celia Cintas
celia.cintas@ibm.com
IBM
Nairobi, Kenya

Michael Kaufmann
kau@zurich.ibm.com
IBM
Zurich, Switzerland

Ioana Giurgiu
igi@zurich.ibm.com
IBM
Zurich, Switzerland

Skyler Speakman
skyler@ke.ibm.com
IBM
Nairobi, Kenya

ABSTRACT

Allocating executors (i.e. compute resources) to distributed processing systems must balance resource costs of scaling-out unnecessarily against artificial, performance-limiting bottlenecks. Naive approaches may allocate executors at the application level, which have predictable costs and performance but are almost guaranteed to be sub-optimal for each of the thousands of diverse, individual stages executed by the application. Users may also have explicit preferences, such as completing an application within a specific time budget while minimizing cost, that existing solutions usually fail to support. We propose a novel method for determining the number of executors *per stage* in a serverless Apache Spark™ environment, enabling users to specify their desired cost-performance trade-off. Our approach trains tree-ensemble models to estimate the run times and costs of a stage as a function of allocated resources. These estimates are then used to recommend resources for each stage individually. We evaluate our approach on TPC-DS and SQLStorm benchmarks and compare it against two baselines. Depending on the user-defined trade-off parameter and setup, our approach achieves ~50% cost savings across 103 TPC-DS queries with only a ~16% slowdown, and ~40.5% on 96 SQLStorm queries at a ~29% slowdown.

VLDB Workshop Reference Format:

Miriam Rateike, Isaac Waweru Wambugu, Celia Cintas, Michael Kaufmann, Ioana Giurgiu, and Skyler Speakman. Stage-Level Executor Allocation in Apache Spark™ with Cost-Performance Trade-offs. VLDB 2026 Workshop: Applied AI for Database Systems and Applications (AIDB 2026).

VLDB Workshop Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/mrateike/pvldb-submission>.

1 INTRODUCTION

Large-scale data processing increasingly relies on data-parallel engines that execute applications on shared clusters managed by automated resource managers [4, 11, 26]. As corporate cloud spending

continues to rise organizations face increasing pressure to reduce costs without compromising performance [8]. This challenge is amplified by the growing use of agentic systems where speculative querying can dramatically increase overall system workloads [12]. In serverless data processing environments (e.g., serverless Apache Spark™), billing is pay-per-use with fine-grained metering of compute and time. Compared to traditional pre-provisioned clusters where resource costs may be amortized or sunk, misallocation is more costly and visible in serverless settings.

A central difficulty in effective resource allocation arises from the heterogeneity of resource demands across stages that can exist within a single SQL query. Adding more resources to an application typically does not lead to proportional improvements in performance. Stage characteristics, such as number of tasks, can vary by orders of magnitude, leading to substantial differences in runtime and sensitivity to allocated resources. Traditional, application-level executor allocation methods assign a fixed amount of resources to each query. Every stage of the query then naively distributes tasks to the available executors. This allocation often results in substantial misallocation as some stages become overprovisioned, increasing costs unnecessarily, while others remain under-resourced and lead to performance slowdowns. We address this problem by providing *stage-level* optimization of executors.

A second, more practical challenge in resource allocation is that stakeholders often have specific optimization goals (e.g., minimizing runtime under a fixed cost cap.) Existing approaches typically optimize a single fixed system objective (e.g., execute as fast as possible) offering limited flexibility for users to express their own preferences regarding the cost-performance trade-off. In this work we propose a mechanism that facilitates the alignment of allocation decisions with clients’ organizational objectives and budget constraints.

Our approach addresses both of the identified challenges. We propose a stage-level, user-preference-aware resource allocation method. Although we designed it for serverless Apache Spark™ environments, the underlying principles, the stage-level resource allocation and explicit incorporation of user-defined objectives can be translated to other execution platforms. We first train simple machine-learning models that estimate per-stage runtime and cost as functions of the allocated executors and relevant stage features. Their predictions are then integrated into an optimization algorithm that selects the resource configuration for each stage, guided by a user-specified optimization objective. Specifically, we

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment. ISSN 2150-8097.

enable users to define their own optimization goals including explicit cost–performance trade-off parameters. Our implementation builds on the Mira resource manager [11], which uses stage level execution modeling to drive dynamic resource allocation with minimal scheduling overhead.

We evaluate our method on two SQL benchmarks: the widely used human-written TPC-DS [17] and the LLM-generated SQL-Storm [20]. Our evaluation compares against two state-of-the-art application-level baselines, CherryPick [2] and ReLoca [9], which we adapt to operate at the stage level for comparison. We refer to these adaptations as CherryPick# and ReLoca#, respectively. Depending on the user-defined trade-off parameter and setup, our approach achieves ~50% cost savings on TPC-DS with only a ~16% slowdown, and up to ~68% lower cost at a ~49% slowdown compared to the Apache Spark™ default, while also running significantly faster than CherryPick# at only a small cost increase.

The remainder of the paper is organized as follows. We first review background and related work (§ 2), then describe the design and implementation of our framework (§ 3), evaluate our approach empirically (§ 4), discuss results and limitations (§ 5), and conclude with a summary and outlook (§ 6).

2 BACKGROUND AND RELATED WORK

2.1 Apache Spark™

Apache Spark™ is a distributed data processing framework for large-scale data analytics [29]. Applications are organized into jobs (see Figure 1 top left). When an application is submitted the framework constructs a logical Directed Acyclic Graph (DAG) based on the sequence of transformations specified by the application. This DAG is then optimized and translated into a physical execution plan comprising a series of stages [29]. Each stage contains one or more tasks that operate on data partitions which can be executed in parallel. Each stage is run on a number of executors. Executors run a fixed number of cores in parallel. Each core can run one task at a time. For example, one executor with 4 cores can run 4 tasks concurrently. If the count of available cores is lower than the number of tasks in a stage then Apache Spark™ schedules those tasks sequentially per core.

2.2 Mira Resource Manager

Resource managers allocate computational resources to applications or their components.¹ For this work, we use an advanced version of the Mira resource manager [11] for Apache Spark™. Mira enables low-overhead resource sharing and fast scale-out across concurrently running applications on a shared cluster. However, similar to Apache Spark™, it still relies on the task load as the key metric to make scale-out decisions: The number of allocated resources (executors) doubles every second until a predefined maximum is reached, or until the number of cores equals the number of tasks (see Figure 3 (a)). While this is a simple and robust method, this scale-out strategy inherently assumes that doubling the allocated resources is a beneficial decision for performance. This is not generally the case and creates an opportunity for novel resource allocation strategies to reduce resource costs with minimal impact on performance. In

¹Allocation is distinct from scheduling, which concerns the concurrency of executed tasks, rather than the allocation of the underlying resources.

the scale-in configuration, executors are deallocated if they remain idle for 60 seconds after completing their assigned tasks. The novel resource allocation methods proposed in this work use a modified version of Mira that limits a stage’s tasks to a pre-determined number of executors. This is in contrast to Apache Spark™ normal behavior which evenly distributes tasks across *all* available executors.

2.3 Machine Learning for Resource Allocation

CherryPick [2] is a system that leverages Bayesian Optimization to distinguish the best or close-to-the-best resource configuration (number of VMs, number of cores, RAM, among others) at a global level with a few test runs of the application. ReLoca has employed a fully connected neural network to guide the allocation of computational resources (number of executors) by learning the impact of operations in data-parallel jobs on system overhead and execution time [9]. This approach was extended by replacing the fully connected architecture with a graph neural network, which more effectively captures the structural relationships between tasks [10]. Another approach employs a graph-based, deterministic analytical model that considers the dynamic, on-demand allocation of executors during the runtime of an Apache Spark™ application [24]. This model predicts the application’s execution time by leveraging idle and backlog time metrics to evaluate executor performance. While existing work [1, 2, 7, 9, 10, 24, 27] has shown performance improvements in finding the optimal scale-out level of an application, it does so globally, i.e., reducing the overall job completion time for the entire application, rather than optimizing resources per application stage, as our proposed strategy does.

Other closely related work employs Gradient Boosting and Random Forest models [21]. They aim to predict job execution time, stage execution time, task execution time, shuffle read/write data, and shuffle write size/records, and evaluate models using the Word-Count application. The work most closely related to ours also trains supervised ML models to predict application execution time using features available before and after execution, and evaluates these models on the test set using only pre-execution information [16]. Different from us, they focus on a single TPC-DS benchmark query across varying input sizes and two other benchmarks (ML and image processing) and use a slightly different stage feature set.

Few prior works have addressed both cost and performance optimization both within [6, 23] and beyond Apache Spark™ [15, 22]. However for Apache Spark™, cost was modeled solely as a function of estimated total runtime, and optimization was limited to a single objective such as cost or fair allocation subject to a performance constraint [6, 23]. In contrast, we model cost and performance separately using distinct trained models, and support a range of optimization objectives that can be flexibly plugged into our framework. We propose a trade-off algorithm, which enables users to determine the optimal number of executors to balance cost and performance according to their preferences.

3 DESIGN AND IMPLEMENTATION

We propose a two-step approach for stage-level executors allocation that identifies the optimal trade-off between cost and performance, guided by a predefined objective. First, we propose two predictive models that estimate performance and cost, respectively, given

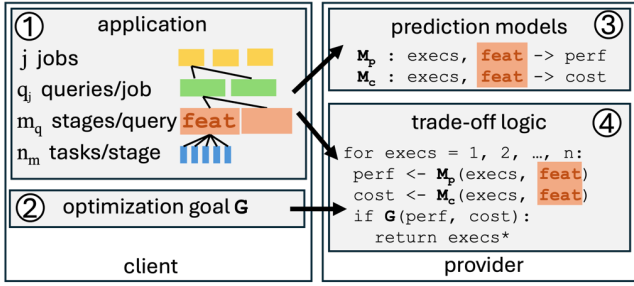


Figure 1: Overview of our method (1) Given an application and (2) a client’s optimization goal, we (3) learn models to predict performance and cost, which (4) we then use to determine the optimal number of executors for a given stage.

a stage’s features and the number of allocated executors. In the second step, we leverage these predictions to determine the optimal number of executors that balances performance and cost according to a predefined objective. We begin with an overview of the pipeline (§ 3.1), followed by data features and cost/performance definitions (§ 3.2), the prediction models (§ 3.3), and the cost-effective solver (§ 3.4).

3.1 Overall Architecture

Figure 1 illustrates the overall architecture of our proposed method which consists of four main components. (1) A client application (e.g., a set of SQL queries) is decomposed into jobs, each comprising stages and tasks. (2) The client specifies an optimization goal (e.g., minimizing costs while limiting performance degradation to 50%). (3) We train prediction models that map stage features and the number of executors to stage performance (M_p) and cost (M_c). (4) A cost-effective solver uses these models and the client’s objective to determine the optimal number of executors for each stage, balancing performance and cost according to the client’s goal.

3.2 Data and Definitions

Stage-level Features. We collect stage-level features from Apache Spark™ event logs which capture stage information pre-execution, during execution, and post-execution. The selected features capture workload size, execution dependencies, and resource demands which are the primary factors affecting stage performance and executor allocation decisions. Additionally, they are independent of application semantics. While not the focus of this work, this feature selection may support future investigations into model generalization across diverse queries and workloads. Some features, such as the number of tasks, are known and logged before execution; others, such as peak memory usage, are recorded only after the stage completes (see Table 1). This split enables two modes of operation for our framework within the client-provider interaction.

The first and most common scenario arises when a client repeatedly executes the same application. In this case the application can be initially run $n \geq 1$ times without our optimization pipeline to

Table 1: Stage features available before (0-shot) and after (n -shot) stage execution. `ListOfOperationTypes` are the different types of RDD operations, which vary across benchmarks.

Availability	Features
0-shot	PartitionsperStage, NumberofTasks, NumberOperations, NumberExecutors, NumberParents, ListOfOperationTypes
n -shot	InputBytesperStage, ResultSize, PeakExecutionMemAllTasks

collect post-execution stage-level data (n -shot).² This data is used to (i) extract the post-execution features from the n runs, record them for each stage, and use their averaged values during inference, and (ii) provide training samples that support continuous learning, improving predictions for future executions of the stage.

The second scenario applies when a client runs an application only once or a few times. Here, repeated runs are not feasible, so predictions must rely solely on features observable prior to execution (0-shot).

Why Standard Metrics Are Insufficient. A natural choice for performance is wall-clock time (StageDuration), and for cost the total wall-clock time charged across all allocated executors. Neither metric is well-suited to learning the relationship between executor count and stage behavior. StageDuration reflects (i) per-invocation overheads and cold-start latencies that cannot be attributed to specific tasks, and (ii) nondeterministic scheduling effects unrelated to the number of executors. Wall-clock cost similarly mixes scheduling artifacts with actual work performed. Below we introduce per-core time-based metrics that isolate the signal that executor allocation can directly influence.

Performance. We define the performance of a stage as the *maximum executor time* P (ExecutorRunTimeMaxbyCore), i.e., the longest duration any single core spends actively executing tasks for the stage, excluding idle time between tasks. Recall that each executor runs multiple cores in parallel. Let C be the set of all cores that run at least one task of a stage of interest, and let t_c denote the time core $c \in C$ spends running tasks of that stage. Then

$$P = \max_{c \in C} (t_c). \quad (1)$$

The intuition is that the stage’s completion time is determined by the slowest core execution which is directly influenced by executor allocation. We treat P as a time-based metric, so that maximizing performance corresponds to minimizing P . We generate training data such that queries do not run in parallel, which ensures that P cleanly captures the per-stage workload during data collection.

Three pieces of evidence support this choice over DurationStage. First, the correlation matrix in Figure 2 shows a strong (> 0.8) positive correlation between DurationStage and ExecutorRunTimeMaxbyCore, while most other features correlate more strongly with the latter, indicating it is more predictable from stage characteristics. Second, models trained to predict ExecutorRunTimeMaxbyCore

²We use “ n -shot” to denote the setting where post-execution features are averaged over n prior runs of the same application; it is not n -shot learning in the standard ML sense.

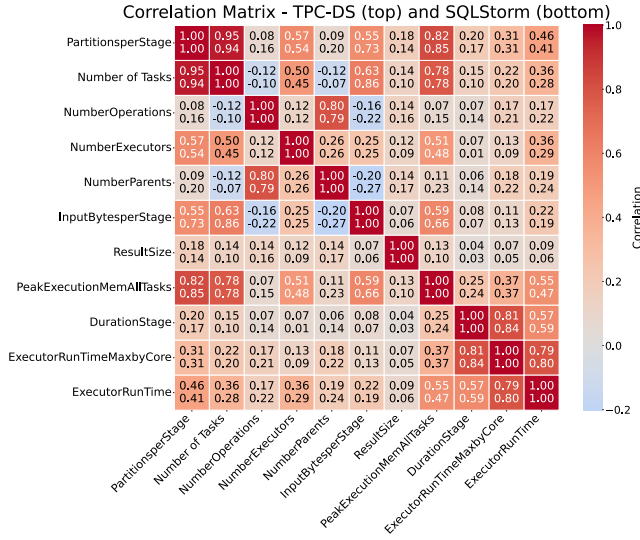


Figure 2: Feature correlation matrix for the TPC-DS and SQLStorm workload, showing pairwise Pearson correlations between model input features.

achieve lower MSE and higher R^2 than those predicting DurationStage (see Appx. B). Third, end-to-end evaluation (§ 4.3) shows that allocations driven by DurationStage-based models either degrade 0-shot runtime substantially or yield no cost reduction at $\theta_r = 0.05$ under n -shot, while ExecutorRunTimeMaxbyCore-based models consistently realize the intended cost–performance trade-off. Since allocations made under this denoised proxy are ultimately evaluated against user-visible wall-clock time in our experiments (§ 3.4), this also confirms that decisions made under P translate to real-world latency gains.

Cost. We define the cost of executing a stage as the total executor run time across all allocated executors:

$$C = \sum_{e \in \mathcal{E}} t_e, \quad (2)$$

where t_e is the run time accumulated by executor e in the executor set $\mathcal{E}$. Consistent with our performance metric, C is measured in core-time rather than wall-clock time, isolating the actual work performed from scheduling artifacts.

This corresponds to a linear per-core execution-time cost model, which aligns with common cloud pricing abstractions such as per-vCPU-second and per-worker-hour billing. We make this choice deliberately for clarity and tractability; the prediction and optimization steps depend only on the relative ordering of cost across configurations, so richer cost models (e.g., machine-based pricing with fixed provisioning costs [27], or models incorporating memory and skew effects) can be substituted without modifying the framework. We discuss this further in § 5.2.

Note that C depends on the number of allocated executors because a task’s execution time varies with its position in the scheduling order: the first task scheduled on a newly allocated core incurs spin-up overhead and therefore runs longer than subsequent tasks on the same core. For instance, Figure 3(b) allocates six tasks

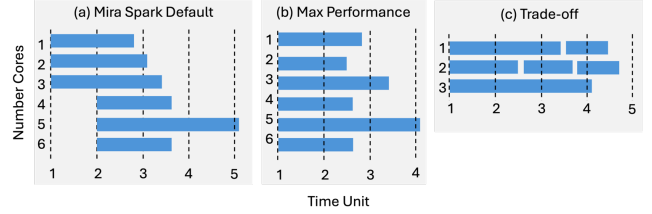


Figure 3: Illustration of different allocation strategies of a single stage with 6 tasks (blue blocks) and 2 executors each equipped with 3 cores: (a) Mira Spark Default adds the second executor after a time unit (baseline), (b) Max Performance uses both executors from the start (Scenario α), (c) Trade-off strategies limit the tasks to a single executor based off predictive models and user preference (Scenario γ).

as first-on-core, whereas Figure 3(c) allocates only three, causing the former to incur higher overall cost despite executing the same tasks. Our methods do not change the order of tasks within stages or the order of stages within the DAG; they determine *how many* executors a stage’s tasks should be distributed over.

3.3 Performance and Cost Prediction

We learn two predictive models: one that estimates performance and another that estimates cost, given a stage characterized by features x and a number of executors e .³ Although runtime and cost are highly correlated in our setting (see Figure 2), separate models preserve the flexibility of the framework to accommodate alternative, potentially less correlated, cost and performance metrics.

The predicted cost and performance values are used in the second step (§ 3.4) to determine the optimal number of executors that balances performance and cost according to the client’s objective. More formally, we define the following two models:

Definition 1 (Performance Prediction Model). Let $x \in \mathcal{X}$ denote the feature vector of stage i , and let $e \in \mathbb{N}_+$ be the number of allocated executors. The performance prediction model $M_P: \mathcal{X} \times \mathbb{N}_+ \rightarrow \mathbb{R}_+$ maps (x, e) to the predicted runtime $M_P(x, e) = \hat{p}_{i,e}$, where $\hat{p}_{i,e}$ is the estimated performance measure of stage i when using e executors.

Definition 2 (Cost Prediction Model). Let $x \in \mathcal{X}$ denote the feature vector of stage i , and let $e \in \mathbb{N}_+$ be the number of allocated executors. The cost prediction model $M_C: \mathcal{X} \times \mathbb{N}_+ \rightarrow \mathbb{R}_+$ maps (x, e) to the predicted cost $M_C(x, e) = \hat{c}_{i,e}$, where $\hat{c}_{i,e}$ is the estimated cost of executing stage i with e executors.

Interventional Training Data. Because we can directly control the number of executors during training data generation, we fix the number of executors at the beginning of each stage and keep it constant throughout the stage’s execution. We then run our benchmark queries multiple times, each with a different number of executors from a predefined range. This process yields data on the

³In our implementation, we use the number of cores as input, obtained directly from the Spark Environment Listener. The trade-off algorithm converts this to the number of executors by dividing by the number of cores per executor.

cost and performance of each query under varying executor configurations. We further exclude all samples where NumberofTasks is less than or equal to the number of cores per executor, since in such cases the optimal number of executors is always one, eliminating the need to learn a decision-making process for such stages.

Non-negative Predictions. As discussed in the previous section, performance and cost are measured in time, which is inherently non-negative. Therefore, it is essential that the predictors output only non-negative values. Off-the-shelf gradient boosting models do not enforce this constraint by default. To address this, we explore two common approaches: (i) applying a logarithmic transformation to the target variable during training and exponentiating the predictions at inference time;⁴ and (ii) post-processing predictions by clipping negative values to a small positive threshold. We evaluate both methods in our experiments to assess their impact on prediction performance. Due to space constraints, we report only the results for the logarithmic transformation, which outperformed the alternative across our experiments. For the baselines, CherryPick# produced small single digit percentages of negative predictions, we thus apply the logarithmic transformation; ReLoca# yielded only non-negative predictions while exhibiting numerical instability when using a log transform, we thus report results trained without applying the logarithmic transformation.

3.4 Cost-effective Solver

We now address how to determine the trade-off between performance (runtime) and cost when executing a stage. Using more executors speeds up execution but increases cost. Optimizing this balance improves resource efficiency and customer satisfaction, whereby the ideal trade-off depends on customer-specific goals and constraints. We begin by presenting a non-exhaustive set of common customer scenarios (see also Figure 3). Then, we translate these goals into optimization problems to determine the optimal number of executors, e^* . Let $P(x, e)$ be the performance function and $C(x, e)$ the cost function of a stage, both determined by the stage’s features x and the number of allocated executors e . $P(x, e)$ represents execution time, and minimizing it corresponds to maximizing performance.

Scenario α : Max performance. A client may be interested in maximizing the performance (minimize run time P) regardless of the costs (e.g., “complete as fast as possible”). They thus seek for $e^* \in \arg \min_e P(x, e)$.⁵

Scenario β : Min cost subject to a performance constraint. A client may aim to minimize cost while ensuring that performance does not fall below a specified threshold θ_P (e.g., “complete in at most θ_P seconds”), seeking for $e^* \in \arg \min_e C(x, e)$ s.t. $P(x, e) \leq \theta_P$.⁶

Scenario γ : Relative performance trade-off. A client may seek to minimize cost up to a relative performance decrease of θ_r (e.g., “minimize cost with at most a 50% slowdown”) compared to the fastest configuration $e' = \arg \min_e P(x, e)$. Thus, they seek for $e^* \in$

Algorithm 1 Scenario γ : Relative performance trade-off.

Require: Executor count list NE , corresponding list of predicted performance $\hat{P}$, relative performance decrease threshold θ_r .

Ensure: Returns optimal number of executors e^* .

```

1: function OPTIMALNE( $NE, \hat{P}, \theta_r$ )
2:    $i \leftarrow \arg \min \hat{P}$  ▷ Index of max performance
3:   for  $j \leftarrow 0 \dots |NE| - 1$  do ▷ Iterate over executor counts
4:     if  $\frac{\hat{P}[j] - \hat{P}[i]}{\hat{P}[i]} \leq \theta_r$  then ▷ Stopping criterion
5:       return  $e^* \leftarrow NE[j]$ 
6:     end if
7:   end for
8: end function

```

$\arg \min_e C(x, e)$ s.t. $\frac{P(x, e) - P(x, e')}{P(x, e')} \leq \theta_r$. An absolute runtime threshold could be used instead by setting $\theta_a = (1 + \theta_r) P(x, e')$.

Optimization Algorithm. The scenarios above largely depend on both performance P and cost C , which we predict using the models M_P and M_C , respectively, as introduced in § 3.3. To select the optimal number of executors e^* , we follow a two-step approach. First, we collect predictions over a predefined list (set) of candidate executor counts NE (e.g., from 16 to 128)⁷. For each stage and each executor count in NE , we generate predicted performance values resulting in the list of predicted costs $\hat{C}$ and list of predicted performance values $\hat{P}$. In the second step, we input these predictions into the optimization algorithm (see Algorithm 1 for Scenario γ ⁸). The algorithm evaluates executor counts in ascending order and returns the smallest count that satisfies the scenario’s requirements (formulated as a stopping criterion) as the optimal number of executors e^* .

4 EVALUATION

We now evaluate our proposed framework. Note that optimality depends on the client’s goal (i.e., the targeted trade-off between cost and performance), and there is no absolute “ground truth” for the optimal number of executors, so we cannot directly measure how close our predictions are to the true optimum.

To evaluate our framework, we first validate the cost and performance prediction models by comparing their outputs to the actual cost and runtime of a stage (§ 4.2). We then evaluate the cost-effective deployment of our framework (§ 4.3). Specifically, we test whether the cost-performance trade-off parameter and the resulting executor recommendations shift the trade-off as expected. We begin by describing our experimental setup.

4.1 Experimental Setup

Benchmarks. TPC-DS [17],⁹ is a human-written SQL query benchmark widely used in prior work [3, 19, 25]. It consists of 99 expert-curated SQL queries. To test robustness and adaptability, we included syntactic variations of queries 14, 23, 24, and 39, which

⁴Similar to, e.g., to [22].

⁵Alternatively, minimizing the total costs regardless of the performance decrease.

⁶Alternatively, maximize performance subject to a cost constraint.

⁷Note, these values must fall within the distribution of the training data used to train the prediction models to ensure in-distribution prediction.

⁸Note that in this algorithm (ln 2), we approximate for simplicity the number of executors with the highest predicted cost by the number of executors predicted to yield the shortest runtime, i.e., highest performance.

⁹<https://www.tpc.org/tpcds/>; last accessed 27th June 2025.

test different execution pattern with similar business logic, resulting in a total of 103 queries. SQLStorm [20]¹⁰ is an LLM-written benchmark which consist of complex queries designed to reflect real-world SQL workloads. For our experiments, we sampled 96 queries from all queries that had sufficient runtime to collect meaningful data, specifically, those exceeding 2000 milliseconds on the deployment-cluster hardware. SQLStorm contains queries that reference three databases; we restricted our selection to those using the TPC-DS schema. Consequently, all sampled queries operate on the same underlying database as the TPC-DS benchmark queries. This results in thousands of stage executions and provides a substantially larger evaluation workload than single-query studies often found in prior work [16].

Both benchmarks are executed on a dataset with 1000 scale factor and aim to capture general characteristics of real-world decision support systems and represents the types of queries expected in client applications. Note that in practice, our method leverages client-specific or application-relevant data to train and then evaluate models on the client’s application.

Training Data Generation and Testing Setup. Our data processing setup uses Apache Spark™ 3.5.4 with Apache Iceberg™ as the table format, Apache Gluten™ as the execution engine, and Mira [11] for dynamic resource provisioning. See Appx. E for details.

We generate five replicas for training and five for testing to capture variance from Apache Spark™ nondeterministic scheduling, which is independent of executor count. This procedure can be thought of running the same workload multiple times with different seeds. Each executor is fixed to have 4 cores. For training data generation, we use 10 static executor allocation with 8, 12, 16, 24, 32, 48, 64, 80, 96, and 128 executors.

Baselines. We compare our approach with the default Mira baselines as well as two state-of-the-art methods implemented from prior work. The default Apache Spark™ configuration doubles the number of executors assigned to a stage every second (see § 2.2). From prior work, we adapted the implementations of CherryPick [2] and ReLoca [9], which we denote as CherryPick# and ReLoca#, respectively. We intentionally compare against these two methods because they are well-established optimization baselines for Spark executor configuration. We do note that our work targets a fundamentally different problem, i.e., fine-grained, stage-level cost-aware prediction, that is orthogonal and complementary to recent end-to-end or online tuning methods. ReLoca and CherryPick were originally designed for application-level prediction. We adapted their input features and modeling assumptions to our stage-level resource-recommendation framework. For both baselines, we use 5-fold cross-validation for hyperparameter tuning, consistent with our own model training. Neither method provided publicly available code, we thus reimplemented them using the information available in the respective papers. Below we summarize our implementations; further details are provided in Appx. B and C.

ReLoca employs a neural network model (Figure 5 [9]). We adopt the original input features (Table 1 [9]), corresponding to our n -shot features plus the number of cores per executor, as well as DAG width and depth metrics (Table 4 [9]). CherryPick is a Gaussian

Process method for cost estimation used in a Bayesian Optimization framework to recommend virtual machine (VM) configuration parameter setups. We modified it to take as input our n -shot features and train their Gaussian Process model to predict cost and performance. For end-to-end testing, we allocate executors according to Algorithm 1 with $\theta_r = 0.0$, matching our setup. Although the baselines are adapted as carefully and fairly as possible, differences in original prediction granularity and input feature requirements may affect direct comparability.

Our method and all baselines are trained on the same interventional dataset, ensuring a fair comparison. The training data consists of 10 executor configurations with 5 repetitions each (50 runs per benchmark). We note that this dataset size is substantially smaller than the approximately 3,000 runs used by ReLoca and the 66 cloud configurations evaluated by CherryPick, which may contribute to higher prediction error for the neural network based baselines. This, however, highlights a key advantage of our approach, which can achieve effective resource allocation with significantly less training data.

Technical Specifications. The data processing infrastructure leverages Apache Spark 3.5.4 integrated with Iceberg as the underlying table format. Iceberg is configured with a Hive-backed catalog and an external Hive Metastore for metadata management while table data is stored in S3-compatible object storage accessed via the S3A connector. The execution engine is accelerated using Gluten with the Velox native backend, enabling columnar execution and native vectorized processing.

The maximum available number of executors is 128. For Training, we shut down an executor after 60 seconds of idleness, following the default Apache Spark™ configuration. For testing, Apache Spark queries Mira for updates on the recommended number of cores computed by our optimization framework at 300-millisecond intervals. Multiple stages may run in parallel as long as their combined recommended executors remain within the executor limit. For our methods and CherryPick#, any executor that becomes idle is shut down immediately when no further tasks remain for its active stage. The Apache Spark baseline uses the default 60-second executor idle timeout.

4.2 Validation of Prediction Models

We first evaluate the test performance of our models and baselines using 5-fold cross-validation on the interventional training data. We then assess, in the following section (§ 4.3), their end-to-end performance when used to predict and adjust resource allocations on benchmark runs. Importantly, the evaluation of the test performance should be interpreted in context: the primary objective of our proposed framework is not highly precise runtime prediction, but rather the effective use of these predictions to support resource allocation decisions. Theoretical work has indeed shown that near-optimal allocation decisions can be achieved despite imperfect or inaccurate predictions [5].

Metrics. We evaluate the prediction performance of the models using Mean Squared Error (MSE) and R^2 score. MSE quantifies the average squared difference between the actual and predicted values. A lower MSE indicates better model performance. However, its

¹⁰<https://github.com/SQL-Storm/SQLStorm>; last accessed 12 January 2026.

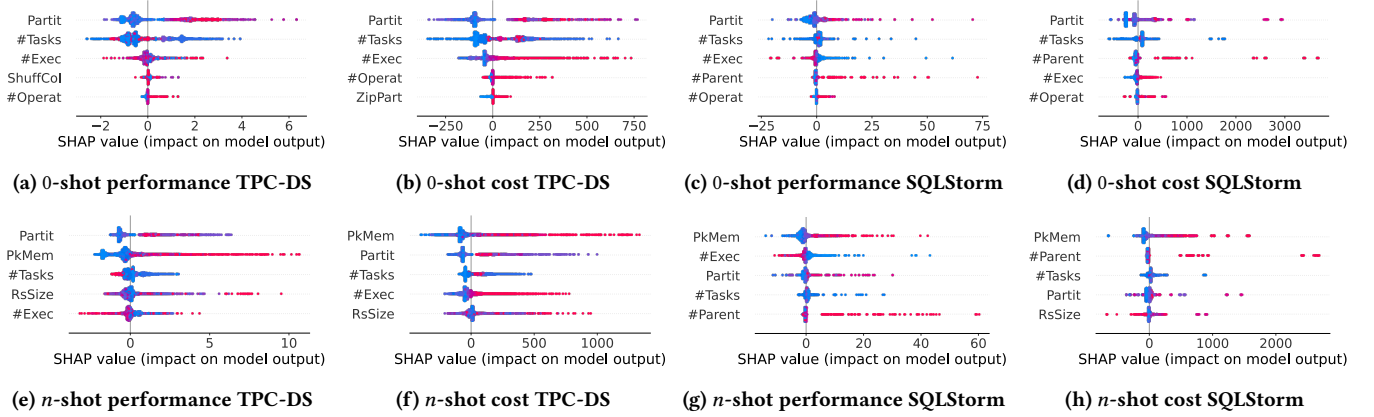


Figure 4: Top-5 SHAP values across our models. Each point represents a stage execution; points to the right indicate that a feature increases the prediction, points to the left indicate a decrease. Red denoting higher feature values and blue lower. Features: PartitionsperStage (Partit), NumberOfTasks (#Tasks), NumberExecutors (#Exec), NumberOperations (#Operat), NumberParents (#Parent), ResultSize (RsSize), PeakExecutionMemAllTasks (PkMem), ShuffledColumnarBatchRDD (Shuffcol), WholeStageZippedPartitionsRDD (ZipPart).

Table 2: Model performance for different benchmarks (B) and targets (T). Cost and Performance (Perf.), Dataset (D), Mean Squared Error (MSE, in seconds) and $R^2 \times 10$. Results averaged over 5-fold cross-validation, our methods marked *. Best (significant) bold.

B	T	Model	MSE ($\downarrow$)	R^2 ($\uparrow$)
TPC-DS	Perf.	ReLoca#	27.11 $\pm$ 8.41	33.52 $\pm$ 12.84
		CherryPick#	3.34 $\pm$ 0.43	89.28 $\pm$ 4.06
		0-shot*	2.90 $\pm$ 0.18	54.34 $\pm$ 1.56
		n-shot*	0.51 $\pm$ 0.05	91.86 $\pm$ 0.76
	Cost	ReLoca#	170,842.84 $\pm$ 30,056.95	19.04 $\pm$ 12.02
		CherryPick#	36,859.59 $\pm$ 4,823.53	81.08 $\pm$ 2.51
		0-shot*	38,652.09 $\pm$ 3,158.79	67.66 $\pm$ 1.49
		n-shot*	4,592.48 $\pm$ 334.10	96.14 $\pm$ 0.35
SQLStorm	Perf.	ReLoca#	40.77 $\pm$ 11.79	15.74 $\pm$ 2.43
		CherryPick#	3.33 $\pm$ 0.56	92.70 $\pm$ 1.78
		0-shot*	2.67 $\pm$ 0.20	94.00 $\pm$ 1.76
		n-shot*	0.65 $\pm$ 0.50	98.78 $\pm$ 0.61
	Cost	ReLoca#	188,705.44 $\pm$ 48,227.33	20.07 $\pm$ 13.63
		CherryPick#	25,239.84 $\pm$ 6,276.70	89.35 $\pm$ 1.99
		0-shot*	11,545.14 $\pm$ 358.13	94.98 $\pm$ 0.75
		n-shot*	856.26 $\pm$ 522.65	99.62 $\pm$ 0.25

absolute value is difficult to interpret, as it is highly dependent on the scale of the target variable. R^2 gives an indication of how well the model explains the variance in the target variable. A higher R^2 suggests that the model captures a significant proportion of the variability of the data. An R^2 value between 0 and 1 indicates the proportion of variance in the target variable explained by the model.

Our Models. For our approach, we evaluate several standard Scikit-learn [18] models, including penalized linear regression

(Lasso, Ridge), Random Forests, and Gradient Boosting (GB). Linear regression and Random Forests consistently underperformed on the test set compared to GB. This finding is consistent with prior work showing that tree-based models perform best for predicting Apache Spark™ performance across different configuration settings [28]. We omit these results in the main paper and report only the GB configuration. Details on the hyperparameter search space and selected models are provided in Appx. B.

Results. Table 2 reports the validation performance of our models and baselines in predicting stage-level performance and cost. We first observe that ReLoca# attains substantially higher MSE and lower R^2 scores (0.1574–0.3352) across benchmarks, i.e., fails to predict cost and performance behavior reliably. Due to its poor predictive performance, we omit ReLoca# from subsequent runtime evaluations (§ 4.3). In contrast, CherryPick# performs competitively with our methods. Although its inference inputs, like those of n -shot, depend on post-execution features, its MSE remains closer to our 0-shot (and higher than n -shot), while its R^2 is more aligned with (and often a slightly lower than) n -shot. Our n -shot model consistently achieves the lowest MSE and highest R^2 (0.9186–0.9962), outperforming all other models. Overall, these results demonstrate that our models, particularly the n -shot model, provide highly reliable predictions of both cost and performance.

Figure 4 reports the top-5 SHAP values [13, 14], which quantify each feature’s contribution to the model’s predictions by indicating both its relative importance and whether it increases or decreases the predicted value. More results in Appx. D. We make the following observations: For 0-shot on both datasets, the most influential features for cost and performance prediction are PartitionsperStage, NumberExecutors, NumberofTasks, and NumberOperations, indicating that parallelism, task granularity, and overall plan size are dominant predictors. For TPC-DS, we additionally observe the importance of specific Spark execution operators. In performance prediction, the presence of ShuffledColumnarBatchRDD is

particularly influential, while in cost prediction it is `WholeStage-ZippedPartitionsRDD`. This suggests that low-level execution mechanisms related to columnar shuffling, and zipping multiple input RDD partitions together are critical drivers of model predictions on this benchmark. In contrast, for `SQLStorm`, `NumberParents` strongly influences cost and performance prediction, suggesting a larger importance of execution-plan structure and dependencies. For n -shot on both datasets, the most influential features for cost and performance prediction are `PartitionsperStage`, `PeakExecutionMemAllTasks`, `NumberExecutors`, and `NumberofTasks`. For TPC-DS, `ResultSize` additionally appears among the most influential features. In `SQLStorm`, `NumberParents` again emerges as an important predictor for performance, while for cost prediction `NumberExecutors` does not rank among the top-5 features; instead, `ResultSize` and `NumberParents` appear in the top five most influential predictors. These results can be explained by differences in benchmark design. `SQLStorm` is intentionally constructed to generate a wide range of queries, from simple to complex, resulting in substantially greater diversity in query plans and execution traces than TPC-DS [20]. Consequently, TPC-DS is more operator-driven, whereas `SQLStorm` is more structure-driven, explaining why execution-plan depth and dependency complexity (`NumberParents`) emerge as key predictors of both cost and performance.

For `NumberExecutors` in TPC-DS, higher executor counts lead to lower predicted runtimes (better performance), while fewer executors result in higher predicted costs. This is matching general expectations. In contrast, for `SQLStorm`, larger numbers of executors are generally associated with higher predicted runtimes and costs. An exception is the n -shot setting, where the effect of `NumberExecutors` is more mixed, contributing to both lower runtimes and higher costs. Importantly, the role of the SHAP analysis in our framework should be viewed in context: the models are primarily intended to inform allocation decisions rather than to deliver highly precise runtime predictions. Accordingly, SHAP values are informative only insofar as they provide actionable insights that support these allocation decisions.

4.3 Evaluation of Cost-effective Solver Deployment

We now evaluate our cost-effective solver in a real-world setting. We focus on Scenario γ (see § 3.4), which we consider the most practical, and run Algorithm 1 with varying values of θ_r . This user-supplied parameter can be thought of as a throttle on the performance of the stage, with $\theta_r = 0.0$ being no throttle (full performance) and $\theta_r > 0$ reducing the recommended resources. We compare the cost and performance of running both benchmarks using our recommendation algorithm with n -shot and 0-shot models against Mira Apache Spark™’s default resource allocator [11], which doubles resources every second until demand is met, as well as the `CherryPick#` baseline. The evaluation is end-to-end: we report cost and performance metrics aggregated over the execution of the entire benchmark across all queries, reflecting net benefits in overall latency and cost.

End-to-End Results. Figure 5 illustrates how different resource-allocation strategies influence both cost (as defined in § 3.2) and performance (importantly, measured here as the wall-clock time

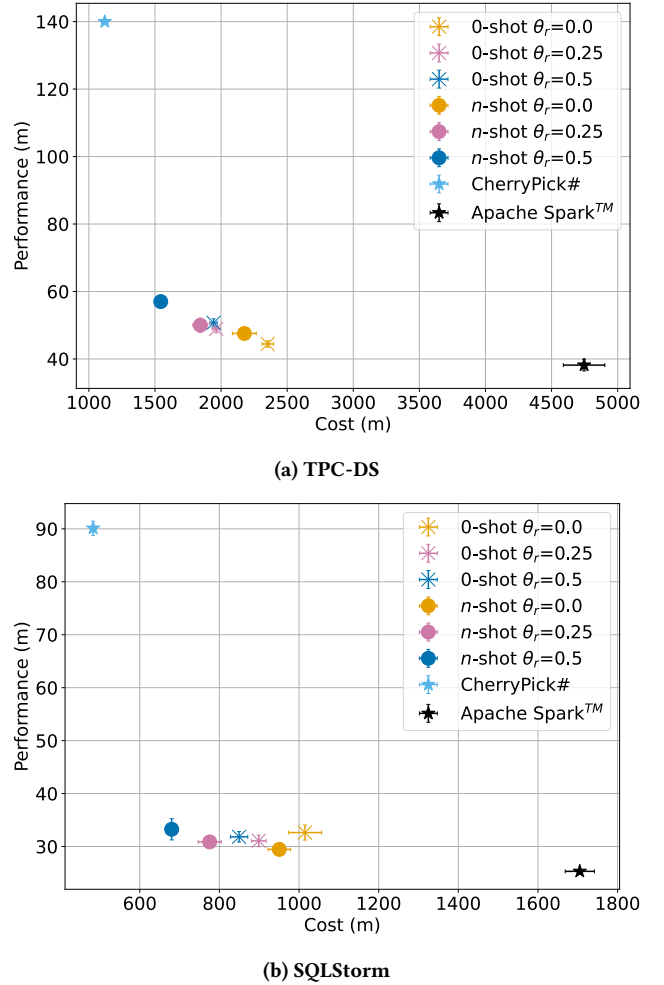


Figure 5: Cost-performance trade-off for Scenario γ across benchmarks, our 0-shot and n -shot methods and baselines (`Apache Spark™`, `CherryPick#`). Cost ($\downarrow$) and performance ($\downarrow$) in minutes over 5 runs.

from query initiation to completion) for both benchmarks (TPC-DS, `SQLStorm`). Metrics are aggregated over all benchmark queries.

First, across both benchmarks, we observe that `Apache Spark™`’s default allocation method achieves the fastest execution times while it also incurs the highest costs, whereas `CherryPick#` yields the lowest costs while also exhibiting the slowest execution times (lowest performance). Our 0-shot and n -shot methods show a balance between these two extremes, while achieving different performance-cost trade-offs depending on the value of the θ_r parameter.

Compared to `Apache Spark™`, our methods deliver substantial cost reductions while maintaining competitive performance. We achieve a cost reduction of at least $\sim 50.4\%$ on TPC-DS and $\sim 40.5\%$ on `SQLStorm` at an increase of runtime by $\sim 16.4\%$ and $\sim 29\%$, respectively (0-shot $\theta_r = 0.0$), and cost reductions up to $\sim 67.5\%$ on TPC-DS and $\sim 60.1\%$ on `SQLStorm`, with an increase of run time of $\sim 49.3\%$ and $\sim 31.4\%$, respectively (n -shot $\theta_r = 0.5$). Compared

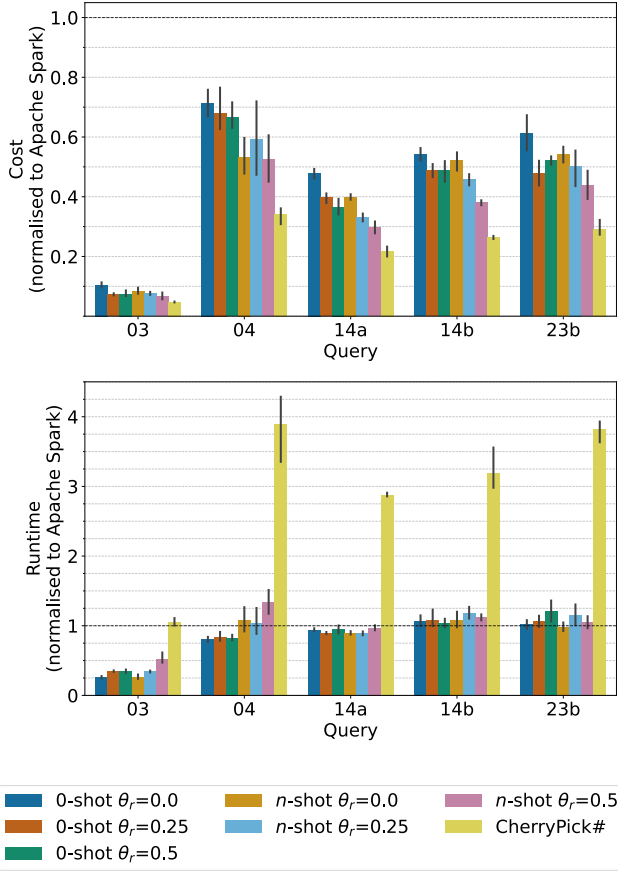


Figure 6: Normalized cost (top) and execution time (bottom) for the five TPC-DS queries with the largest absolute cost difference between Apache Spark™ and our 0-shot $\theta_r = 0.0$. Showing mean values (± 1 standard deviation error bars) across 5 runs, normalized by Apache Spark™ mean for the same query (dashed line at 1.0); values below 1.0 indicate improvement over the baseline.

to CherryPick#, our methods show substantial performance improvements while incurring modest cost increases. Especially for n -shot $\theta_r = 0.5$, our method runs TPC-DS 59.3% faster at an 37.8% cost increase, and 63.1% faster on SQLStorm at an 40.9% cost increase. These results demonstrate that our methods, especially n -shot, achieve a better performance-cost trade-off compared to both baselines across both benchmarks.

The trade-off parameter θ_r serves as a control mechanism for balancing performance and cost objectives in resource allocation. Increasing θ_r reduces costs across both benchmarks and methods. On TPC-DS, increasing θ_r from 0.0 to 0.5 reduces costs from 2353 to 1943 mins for 0-shot and from 2176 to 1544 mins for n -shot. A similar trend can be observed for SQLStorm. For TPC-DS, this cost reduction comes with the expected increase in runtime across models, whereas for SQLStorm 0-shot, $\theta_r > 0$ reduces both cost and runtime compared to $\theta_r = 0.0$.

0-shot and n -shot differ in the data available at inference time. The 0-shot method operates based on stage characteristics known before execution, while n -shot additionally leverages stage information available after execution that was collected over the past n runs. For the same value of θ_r , n -shot consistently achieves across benchmarks lower cost than 0-shot, and in some achieves Pareto-optimality (both lower cost and lower runtime). We attribute this n -shot performance to its more accurate cost predictions enabled by the richer input data.

Query-level Results. Figure 6 presents normalized cost (top) and performance (bottom) for five TPC-DS queries that showed the greatest divergence in cost between our 0-shot ($\theta_r = 0.0$) approach and the Apache Spark™ baseline. Cost and performance metrics are normalized per query to the Apache Spark mean (Apache Spark = 1.0); values below 1 indicate improvement over the baseline. For results across all queries, see Appx. D. Query 3 sums discount amounts for items from a specific manufacturer sold at a specific time, grouped by year and brand, returning the top results. Query 4 computes yearly customer sales across channels, and returns customers whose catalog sales growth exceeds both store and web growth. Queries 14 (a/b) select items sold across channels over a given period, and compute sales and counts by channel, brand, class, and category for groups exceeding average sales. Query 23 (a/b) sums sales from different channels for a given period for frequently sold items and high-spending customers.

We observe that 0-shot and n -shot variants of our method behave similarly across individual queries, while there is a large difference to CherryPick# baselines. For query 3, our approaches reduce costs to below 10% of Apache Spark™’s, whereas CherryPick# lowers costs further to about 5%. However, this additional cost reduction comes with execution times slightly above Apache Spark™, while our methods achieve execution times of only 0.25-0.5% of Apache Spark™’s. The displayed queries appear to demand substantially greater resources from Apache Spark™, which, as mentioned above, doubles its allocated resources at each time step until a limit is reached. This suggests that the advantages of our methods arise primarily from their improved handling of more resource-intensive queries.

Investigation of Executor Allocations. Figure 7 shows the distribution of recommended executors for our 0-shot and n -shot methods across both benchmarks for $\theta_r = 0.0$ and $\theta_r = 0.5$. This provides insight into how the different methods realize the desired cost-performance tradeoff and whether they achieve it through substantially different resource provisioning strategies. We remind the reader that selecting between n -shot and 0-shot modes, and tuning θ_r , depends on user- and application-specific priorities.

Overall, the recommended number of executors for SQLStorm ranges from 1-54, and for TPC-DS ranges from 1-123. At $\theta_r = 0.0$, we observe a wider spread of recommendations compared to $\theta_r = 0.5$ across benchmarks and models, reflecting that the stronger cost-reduction objective for $\theta_r = 0.5$ favors smaller number of executors. Across both benchmarks, the n -shot method recommends on average less executors than 0-shot at a lower variance.

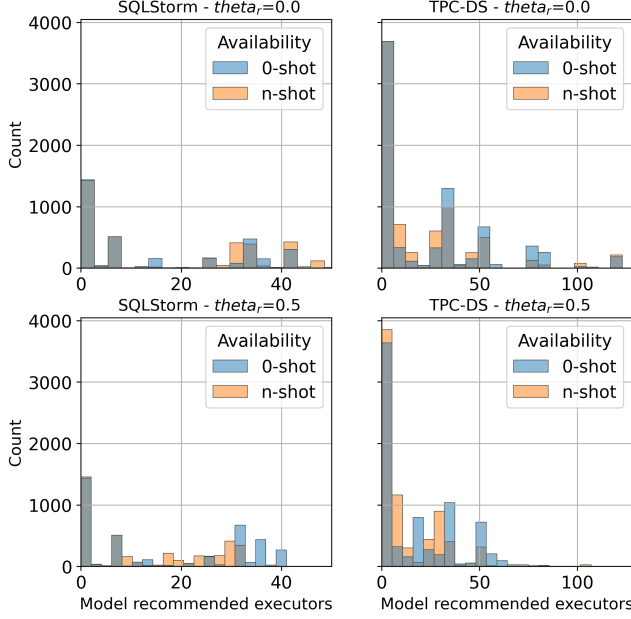


Figure 7: Distribution of recommended executors over all stages of all queries for 0-shot (left) and n -shot (right) with $\theta_r = 0.0$ (top) and $\theta_r = 0.5$ (bottom); Scenario γ . Across 5 runs.

5 DISCUSSION AND LIMITATIONS

5.1 Practical Insights

Our framework and experimental results yield three practical insights for resource allocation in serverless data processing.

Allocation Quality Does Not Require Prediction Accuracy. Although we frame the method in terms of predicting per-stage runtime and cost, our results show that allocation quality depends on preserving the *relative* cost-performance ordering of stages across resource configurations, not on the absolute accuracy of either prediction. This is consistent with recent theoretical results showing that near-optimal allocation can be achieved with substantially fewer samples than accurate prediction requires [5]. In practice, this means a small interventional training set (50 runs per benchmark in our setup) suffices to drive effective allocation, which is an order of magnitude less than the $\sim 3,000$ runs used by ReLoca [9] or the 66 cloud configurations explored by CherryPick [2].

Stage-Level Allocation Closes a Structural Gap in Apache SparkTM’s Scale-Out Heuristic. Apache SparkTM’s default strategy doubles allocated executors every second until task demand is met. This eager scale-out systematically over-provisions stages whose tasks finish quickly, where the marginal executor contributes little but is still billed. Our stage-level formulation reasons explicitly about the number of executors each stage should receive, and the resulting allocations (Figure 7) concentrate around far smaller executor counts than Apache SparkTM would have allocated, while recovering most of Apache SparkTM’s runtime. On TPC-DS, this translates to $\sim 50\%$ lower cost at a $\sim 16\%$ runtime increase relative to the Spark default (0-shot, $\theta_r = 0.0$).

User-Defined Trade-Offs Are the Right Interface for Serverless Allocation. Users of pay-per-use environments typically express preferences as objectives (“minimize cost subject to a runtime budget”) rather than as resource plans. Our θ_r parameter exposes the cost-performance frontier directly: increasing θ_r from 0.0 to 0.5 moves the operating point smoothly along this frontier on both benchmarks, allowing users to select allocations that match organizational priorities. This is, to our knowledge, the first stage-level allocation method that exposes such an interface.

5.2 Limitations

Cost Model. We instantiate the framework with a linear per-core execution-time cost model, which aligns with common cloud pricing abstractions such as per-vCPU-second and per-worker-hour billing. The framework itself depends only on relative cost ordering across configurations and is compatible with richer cost models (e.g., machine-based pricing with fixed provisioning costs [27], or models incorporating memory and skew effects). Integrating and evaluating such models is an important direction for future work.

Prediction Model Family. We adopt tree-based ensemble models, which capture non-linear feature interactions well and provide stable predictions within the training distribution. As is common for learned system models [28], predictive accuracy degrades under substantial distribution shift, for instance when executor counts or hardware configurations lie far outside the training range. We view this as an acceptable trade-off given that production cluster configurations and workloads tend to be stable over meaningful time horizons, and that incremental retraining can absorb gradual drift.

Evaluation Scope. Due to confidentiality constraints, our evaluation relies on two public benchmarks rather than production workloads. TPC-DS and SQLStorm together span thousands of stage executions and a broad spectrum of analytical query patterns, from human-curated decision-support queries to LLM-generated complex queries, but they cannot fully represent the heterogeneity of production environments. Evaluation is further bounded by the executor configurations and scale factor used during data collection.

5.3 Deployment Considerations

When to Use 0-shot vs. n -shot. The two modes target different deployment regimes. The n -shot mode achieves consistently better cost-performance trade-offs (Figure 5) and is the natural choice when an application is executed repeatedly, since the post-execution features it relies on can be extracted from prior runs. The 0-shot mode applies when only pre-execution features are available, such as for one-off queries or newly deployed applications, and still delivers substantial cost reductions, though with somewhat higher variance in recommended executor counts (Figure 7).

Cost of Training Data Collection. Generating the interventional training set is a one-time, offline cost amortized over subsequent executions. In our setup, this required 59.7 hours for TPC-DS and 26.5 hours for SQLStorm (10 executor configurations $\times$ 5 repetitions per benchmark). In production settings where workload and cluster characteristics are stable, this cost is small relative to the cumulative cost savings during deployment. Where workloads drift

gradually, models can be updated incrementally from newly observed executions, avoiding full retraining.

Integration Overhead. Our implementation extends the Mira resource manager [11] and requires only that the scheduler honor per-stage executor caps. The prediction and optimization steps run in milliseconds per stage and are queried at 300 ms intervals during execution, introducing negligible scheduling overhead.

6 CONCLUSION AND OUTLOOK

Efficient resource allocation remains a key challenge in distributed data processing systems, especially in serverless environments where cost and performance must be carefully balanced. This work presents a practical, data-driven approach to per-stage resource allocation in Apache Spark™, offering substantial improvements over static, application-level strategies. By training tree-based models to predict stage-level runtime and cost, and using these predictions to guide resource allocation, our framework enables user-defined trade-offs. E.g., in 0-shot mode with $\theta_r = 0.25$, we observe an average cost reduction of 58.6% on TPC-DS at a 28.1% runtime increase, compared to $\theta_r = 0.0$, which yields a 50.4% cost reduction at a 16.4% slowdown relative to the Apache Spark™ default. This demonstrates that predictive modeling and user-defined trade-offs enable fine-grained, cost-efficient optimization of large-scale data pipelines.

Our results are based on an empirical evaluation of Scenario γ using a fixed set of TPC-DS and SQLStorm queries for both 0-shot and n -shot approaches. Future work should broaden this evaluation to additional scenarios introduced in § 3.4 and investigate generalization to unseen queries.

Outlook on Generalization. In this work, the training data is limited to configurations with up to 128 executors (512 cores), which also defines the scope of our evaluation. We make no claims about model behavior outside the observed configuration range. Changes in available resources or cluster configurations would require retraining the models to ensure reliable predictions and allocation decisions, as discussed in the previous section. A promising direction for improving generalization beyond is to shift from stage-level to *task*-level modeling, where predictions are based on more fine-grained execution characteristics that may extrapolate better across workloads and applications. Incorporating task-level features and/or with hierarchical or multi-level learning frameworks could lead to more robust cost and performance estimates that generalize more effectively to unseen applications and workloads.

Outlook on Intelligent Scale-in. The performance metrics of this paper highlight the importance of intelligently allocating resources to a query’s individual stages. Further cost reductions can be obtained from reducing resources during a query’s execution. Perhaps simple “timeout” methods will be effective, however, the larger goal would be to preemptively scale-in resources based on estimated demand from subsequent stages.

REFERENCES

- [1] Nasim Ahmed, Andre LC Barczak, Mohammad A Rashid, and Teo Susnjak. 2021. An enhanced parallelisation model for performance prediction of apache spark on a multinode hadoop cluster. *Big Data and Cognitive Computing* 5, 4 (2021), 65.
- [2] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. {CherryPick}: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 469–482.
- [3] Melyssa Barata, Jorge Bernardino, and Pedro Furtado. 2015. An overview of decision support benchmarks: TPC-DS, TPC-H and SSB. *New Contributions in Information Systems and Technologies: Volume 1* (2015), 619–628.
- [4] Eric A. Brewer. 2015. Kubernetes and the path to cloud native. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (Kohala Coast, Hawaii) (SoCC '15)*. Association for Computing Machinery, New York, NY, USA, 167. <https://doi.org/10.1145/2806777.2809955>
- [5] Silvia Casacuberta and Moritz Hardt. 2026. Good Allocations from Bad Estimates. In *The Fourteenth International Conference on Learning Representations*. <https://openreview.net/forum?id=rxZdaKhu2l>
- [6] Stratos Dimopoulos, Chandra Krintz, and Rich Wolski. 2017. Justice: A deadline-aware, fair-share resource allocator for implementing multi-analytics. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 233–244.
- [7] Han Gao, Zhengyu Yang, Janki Bhimani, Teng Wang, Jiayin Wang, Bo Sheng, and Ningfang Mi. 2017. 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)*. IEEE, 1–9.
- [8] Gartner, Inc. 2023. Why Cloud Budgets Don’t Stay in Check – and How to Make Sure Yours Do. *Gartner Research* (2023). <https://www.gartner.com/en/articles/why-cloud-budgets-don-t-stay-in-check-and-how-to-make-sure-yours-do> Accessed: 2025-06-27.
- [9] Zhiyao Hu, Dongsheng Li, Dongxiang Zhang, and Yixin Chen. 2020. Reloca: Optimize resource allocation for data-parallel jobs using deep learning. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 1163–1171.
- [10] Zhiyao Hu, Dongsheng Li, Dongxiang Zhang, Yiming Zhang, and Baoyun Peng. 2021. Optimizing resource allocation for data-parallel jobs via GCN-based prediction. *IEEE Transactions on Parallel and Distributed Systems* 32, 9 (2021), 2188–2201.
- [11] Michael Kaufmann, Kornilios Kourtis, Adrian Schuepbach, and Martina Zitterbart. 2018. Mira: sharing resources for distributed analytics at small timescales. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 231–241.
- [12] Shu Liu, Soujanya Ponnappalli, Shreya Shankar, Sepanta Zeighami, Alan Zhu, Shubham Agarwal, Ruiqi Chen, Samion Suwito, Shuo Yuan, Ion Stoica, et al. 2026. Supporting our ai overlords: Redesigning data systems to be agent-first. In *6th Annual Conference on Innovative Data Systems Research (CIDR '26)*.
- [13] Scott M. Lundberg, Gabriel Erion, Hugh Chen, Alex DeGrave, Jordan M. Prutkin, Bala Nair, Ronit Katz, Jonathan Himmelfarb, Nisha Bansal, and Su-In Lee. 2020. From local explanations to global understanding with explainable AI for trees. *Nature Machine Intelligence* 2, 1 (2020), 2522–5839.
- [14] Scott M Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. *Advances in Neural Information Processing Systems* 30 (2017).
- [15] Chenghao Lyu, Qi Fan, Fei Song, Arnab Sinha, Yanlei Diao, Wei Chen, Li Ma, Yihui Feng, Yaliang Li, Kai Zeng, et al. 2022. Fine-grained modeling and optimization for intelligent resource management in big data processing. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3098–3111.
- [16] Alexandre Maros, Fabricio Murai, Ana Paula Couto da Silva, Jussara M Almeida, Marco Lattuada, Eugenio Gianniti, Marjan Hosseini, and Danilo Ardagna. 2019. Machine learning for performance prediction of spark cloud applications. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 99–106.
- [17] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The making of tpc-ds. In *Proceedings of the International Conference on Very Large Data Bases*, Vol. 32. 1049–1058.
- [18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
- [19] Meikel Poess, Raghunath Othayoth Nambiar, and David Walrath. 2007. Why you should run TPC-DS: a workload analysis. In *Proceedings of the 33rd international conference on Very large data bases*. 1138–1149.
- [20] Tobias Schmidt, Viktor Leis, Peter Boncz, and Thomas Neumann. 2025. SQLStorm: Taking Database Benchmarking into the LLM Era. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4144–4157.
- [21] Piyush Sewal and Hari Singh. 2022. A machine learning approach for predicting execution statistics of spark application. In *2022 Seventh International Conference on Parallel, Distributed and Grid Computing (PDGC)*. IEEE, 331–336.
- [22] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. Cost models for big data query processing: Learning, retrofitting, and our findings. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 99–113.
- [23] Subhajit Sidhanta, Wojciech Golab, and Supratik Mukhopadhyay. 2016. Optext: A deadline-aware cost optimization model for spark. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 193–202.
- [24] Hina Tariq and Olivia Das. 2023. Execution time prediction model that considers dynamic allocation of spark executors. In *European Workshop on Performance Engineering*. Springer, 340–352.

- [25] Anna Thanopoulou, Paulo Carreira, and Helena Galhardas. 2012. Benchmarking with TPC-H on off-the-shelf hardware an experiments report. In *14th International Conference on Enterprise Information Systems, ICEIS 2012*. 205–208.
- [26] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*. 1–16.
- [27] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient performance prediction for {Large-Scale} advanced analytics. In *13th USENIX symposium on networked systems design and implementation (NSDI 16)*. 363–378.
- [28] Guolu Wang, Jungang Xu, and Ben He. 2016. A novel method for tuning configuration parameters of spark based on machine learning. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 586–593.
- [29] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.

APPENDIX

This appendix provides supplementary material that accompanies the paper, including extended experimental results, model details, and additional analyses. Because of the workshop’s PDF submission size limit, all figures illustrating the additional results are provided in the Online Appendix: <https://github.com/mrateike/pvldb-submission>.

- *Appendix A: Code and Repository Details* — Additional information on code release and additional artifacts.
- *Appendix B: Our Model Details* — Additional information on the prediction models used in our evaluation.
- *Appendix C: Baselines* — Additional information on the baseline methods used in our evaluation.
- *Appendix D: Extended Results* — Extended analyses for TPC-DS and StormSQL benchmarks.
- *Appendix E: Hardware and Setup* — Additional information on hardware and train/test setup.

A CODE AND REPOSITORY

We are unable to release code due to corporate restrictions. We do, however, provide the following artifacts in our Online Appendix: <https://github.com/mrateike/vldb-submission>.

- *Results Data* — Data used to generate the figures in the main paper and appendix (available in the data directory).
- *Models* — Trained prediction models used to generate results in the paper and appendix (available in the models directory).

B OUR PREDICTION MODELS

B.1 Hyperparameters Best Models

In Table 3, we report the hyperparameters for the GradientBoostingRegressor¹¹ models used in the main paper for end-to-end testing for Cost and Performance (ExecutorRunTimeMax). The best hyperparameters have been selected using 5-fold-cross-validation.

B.2 Cross-Validation of Our Models

We report complete results for training 0-shot and n -shot models, comparing LinearRegression, GradientBoostingRegressor, and RandomForestRegressor for different targets, as explained in the main paper:

- Table 4 shows TPC-DS 0-shot model evaluation results
- Table 5 shows SQLStorm 0-shot model evaluation
- Table 6 shows TPC-DS n -shot model evaluation
- Table 7 shows SQLStorm n -shot model evaluation

C BASELINE MODELS

For all cross-validation experiments for our models and baselines we used the same training/testing splits.

C.1 Adaptations

ReLoca Adaptation (ReLoca#).

- *Category:* n -shot

¹¹<https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html>

Table 3: Model hyperparameters and performance metrics for our best models used during evaluation.

Data	Model	Hyperparameters	MSE	R ²
TPC-DS	0-shot	n_estimators: 410, max_depth: 6, learning_rate: 0.1, subsample: 0.5, min_weight_fraction_leaf: 0.0	2.8957 $\pm$ 0.1779	0.5434 $\pm$ 0.0156
TPC-DS	<i>n</i> -shot	n_estimators: 460, max_depth: 7, learning_rate: 0.1, subsample: 0.5, min_weight_fraction_leaf: 0.0	0.5148 $\pm$ 0.0385	0.9186 $\pm$ 0.0076
SQLStorm	0-shot	n_estimators: 460, max_depth: 7, learning_rate: 0.1, subsample: 0.5, min_weight_fraction_leaf: 0.0	2.6658 $\pm$ 0.2016	0.9400 $\pm$ 0.0176
SQLStorm	<i>n</i> -shot	n_estimators: 460, max_depth: 6, learning_rate: 0.1, subsample: 0.5, min_weight_fraction_leaf: 0.0	0.6524 $\pm$ 0.5041	0.9878 $\pm$ 0.0061

Table 4: TPC-DS 0-shot model evaluation results. Linear Regression (LREG), Random Forest (RFOR), Gradient Boosting (GBST), SD (DS), ExecutorRunTimeMax(Max).

Target	Model	Parameters	MSE	R ²
Cost	LREG	fit_intercept: False	12,765,895.7817 $\pm$ 5,101,713.6907	-106.5049 $\pm$ 43.5657
Cost	RFOR	bootstrap: True, n_estimators: 160, max_depth: 7	45,759.9685 $\pm$ 4,042.5299	0.6174 $\pm$ 0.0187
Cost	GBST	n_estimators: 210, max_depth: 6, learning_rate: 0.1, subsample: 0.5, min_weight_fraction_leaf: 0.0	38,652.0878 $\pm$ 3,158.7891	0.6766 $\pm$ 0.0149
Perf (SD)	LREG	fit_intercept: False	87.2132 $\pm$ 22.3929	-3.2026 $\pm$ 1.1611
Perf (SD)	RFOR	bootstrap: True, n_estimators: 410, max_depth: 7	11.7462 $\pm$ 0.5864	0.4377 $\pm$ 0.0116
Perf (SD)	GBST	n_estimators: 410, max_depth: 6, learning_rate: 0.1, subsample: 0.5, min_weight_fraction_leaf: 0.0	10.2669 $\pm$ 0.5193	0.5085 $\pm$ 0.0103
Perf (Max)	LREG	fit_intercept: True	27.6213 $\pm$ 5.3687	-3.3617 $\pm$ 0.8546
Perf (Max)	RFOR	bootstrap: True, n_estimators: 160, max_depth: 7	3.0474 $\pm$ 0.1948	0.5195 $\pm$ 0.0175
Perf (Max)	GBST	n_estimators: 410, max_depth: 6, learning_rate: 0.1, subsample: 0.5, min_weight_fraction_leaf: 0.0	2.8957 $\pm$ 0.1779	0.5434 $\pm$ 0.0156

- *Key idea:* ReLoca is a deep learning-based framework that optimizes resource allocation for data-parallel jobs to minimize job completion time. It employs a deep neural network trained with an adaptive sampling method to learn the impact of job operations on system overhead and compute time, achieving a 29.85% reduction in job completion time compared to existing methods.
- *Original setting:* ReLoca focuses in optimizing performance rather than cost at an application level. The computational domains differ substantially between the two approaches—ReLoca targets machine learning workloads, whereas our evaluation utilizes SQL query execution traces.
- *Adaptations considerations:* We adopted the input features specified in Table 1 from the original ReLoca publication

and incorporated the proposed neural network architecture (as depicted in Figure 5) into our hyperparameter optimization framework. Following the computational methodology described in Figure 4, we derived the DAG width and depth metrics for each application. A fundamental adaptation was required due to our framework’s stage-level granularity for resource recommendation: we transitioned from application-level execution duration prediction (as originally formulated in ReLoca) to stage-level predictions. Regarding data collection and experimental design, ReLoca employs an interventional sampling protocol that systematically balances under-allocation scenarios (which degrade performance) and over-allocation scenarios (which introduce substantial system overhead). This approach yields

Table 5: SQLStorm 0-shot model evaluation. Linear Regression (LREG), Random Forest (RFor), Gradient Boosting (GBst). SD (DS), ExecutorRunTimeMax(Max).

Target	Model	Parameters	MSE	R ²
Cost	LREG	fit_intercept: True	924,778.1087 $\pm$ 244,326.3015	-2.9266 $\pm$ 0.7885
Cost	RFor	bootstrap: True, n_estimators: 10, max_depth: 7	13,358.5137 $\pm$ 846.0770	0.9416 $\pm$ 0.0115
Cost	GBst	n_estimators: 260, max_depth: 7, learning_rate: 0.1, subsample: 0.5, min_weight_fraction_leaf: 0.0	11,545.1384 $\pm$ 358.1276	0.9498 $\pm$ 0.0075
Perf (DS)	LREG	fit_intercept: True	58.4296 $\pm$ 14.6535	0.0816 $\pm$ 0.0072
Perf (DS)	RFor	bootstrap: True, n_estimators: 10, max_depth: 7	8.5018 $\pm$ 1.2000	0.8609 $\pm$ 0.0255
Perf (DS)	GBst	n_estimators: 360, max_depth: 7, learning_rate: 0.1, subsample: 0.5, min_weight_fraction_leaf: 0.0	6.0936 $\pm$ 0.2921	0.8971 $\pm$ 0.0284
Perf (Max)	LREG	fit_intercept: False	60.9472 $\pm$ 11.1733	-0.3060 $\pm$ 0.1907
Perf (Max)	RFor	bootstrap: False, n_estimators: 10, max_depth: 7	20.0779 $\pm$ 7.2291	0.5891 $\pm$ 0.0637
Perf (Max)	GBst	n_estimators: 460, max_depth: 7, learning_rate: 0.1, subsample: 0.5, min_weight_fraction_leaf: 0.0	2.6658 $\pm$ 0.2016	0.9400 $\pm$ 0.0176

Table 6: TPC-DS n -shot model results. Linear Regression (LREG), Random Forest (RFor), Gradient Boosting (GBst). SD (DS), ExecutorRunTimeMax(Max).

Target	Model	Parameters	MSE	R ²
Cost	LREG	fit_intercept: False	56,421,446.1955 $\pm$ 31,701,008.5355	-475.6542 $\pm$ 270.6458
Cost	RFor	bootstrap: True, n_estimators: 110, max_depth: 7	21,867.7576 $\pm$ 1,541.9805	0.8164 $\pm$ 0.0151
Cost	GBst	n_estimators: 460, max_depth: 7, learning_rate: 0.1, subsample: 0.5, min_weight_fraction_leaf: 0.0	4,592.4841 $\pm$ 334.0984	0.9614 $\pm$ 0.0035
Perf. (SD)	LREG	fit_intercept: False	136.5464 $\pm$ 33.6868	-5.5600 $\pm$ 1.6930
Perf. (SD)	RFor	bootstrap: True, n_estimators: 110, max_depth: 7	6.1407 $\pm$ 0.3343	0.7061 $\pm$ 0.0079
Perf. (SD)	GBst	n_estimators: 460, max_depth: 7, learning_rate: 0.1, subsample: 0.5, min_weight_fraction_leaf: 0.0	1.9204 $\pm$ 0.1576	0.9079 $\pm$ 0.0080
Perf. (Max)	LREG	fit_intercept: True	174.3614 $\pm$ 46.5823	-26.7242 $\pm$ 8.0328
Perf. (Max)	RFor	bootstrap: True, n_estimators: 110, max_depth: 7	1.2151 $\pm$ 0.0902	0.8077 $\pm$ 0.0190
Perf. (Max)	GBst	n_estimators: 460, max_depth: 7, learning_rate: 0.1, subsample: 0.5, min_weight_fraction_leaf: 0.0	0.5148 $\pm$ 0.0385	0.9186 $\pm$ 0.0076

Table 7: SQLStorm n -shot model results. Linear Regression (LREG), Random Forest (RFOR), Gradient Boosting (GBST). StageDuration(SD), ExecutorRunTimeMax(Max).

Target	Model	Parameters	MSE	R ²
Cost	LREG	fit_intercept: True	1,105,307.8681 $\pm$ 255,780.1731	-3.7082 $\pm$ 0.8108
Cost	RFOR	bootstrap: True, n_estimators: 460, max_depth: 7	4,958.8561 $\pm$ 363.9871	0.9783 $\pm$ 0.0042
Cost	GBST	n_estimators: 410, max_depth: 7, learning_rate: 0.1, subsample: 0.5, min_weight_fraction_leaf: 0.0	856.2648 $\pm$ 522.6476	0.9962 $\pm$ 0.0025
Perf. (SD)	LREG	fit_intercept: True	56.7856 $\pm$ 13.1241	0.1033 $\pm$ 0.0300
Perf. (SD)	RFOR	bootstrap: True, n_estimators: 110, max_depth: 7	6.0071 $\pm$ 0.7221	0.9017 $\pm$ 0.0169
Perf. (SD)	GBST	n_estimators: 460, max_depth: 7, learning_rate: 0.1, subsample: 0.5, min_weight_fraction_leaf: 0.0	2.2269 $\pm$ 0.3552	0.9636 $\pm$ 0.0064
Perf. (Max)	LREG	fit_intercept: True	65.6093 $\pm$ 10.3240	-0.4159 $\pm$ 0.2239
Perf. (Max)	RFOR	bootstrap: True, n_estimators: 10, max_depth: 7	8.5395 $\pm$ 3.5709	0.8305 $\pm$ 0.0439
Perf. (Max)	GBST	n_estimators: 460, max_depth: 6, learning_rate: 0.1, subsample: 0.5, min_weight_fraction_leaf: 0.0	0.6524 $\pm$ 0.5041	0.9878 $\pm$ 0.0061

approximately 600 samples per application (approximately 3,000 total experimental runs), with 200 samples per application reserved for training.

In contrast, our experimental dataset comprises approximately 50 runs per benchmark (totaling 100 experimental runs across both benchmarks), substantially constrained by data collection limitations inherent to production system monitoring. These methodological disparities in data scale, workload characteristics, prediction granularity, and experimental scope, represent significant constraints in our experimental setup and may impact the direct comparability of baseline method performance in our evaluation.

- *Grid search parameters:* The final model configuration was selected through a grid search procedure that systematically evaluated the hyperparameter space across five dimensions: hidden layer sizes and depth, activation functions, learning rate parameters, maximum iteration thresholds, and solver algorithms.

CherryPick Adaptation (CherryPick#).

- *Category:* n -shot
- *Key idea:* CherryPick addresses the challenge of selecting optimal cloud configurations for recurring big data analytics jobs. The system employs Bayesian Optimization to construct performance models that achieve sufficient accuracy to identify the best or near-best configurations from numerous VM instance types and cluster sizes with minimal test runs. By building models that distinguish optimal configurations from suboptimal alternatives through only

a few experimental evaluations, CherryPick significantly reduces the search cost overhead. Experimental validation demonstrates that CherryPick achieves a 45-90% success rate in identifying optimal configurations, while reducing search cost by up to 75% compared to existing configuration selection approaches.

- *Original setting:* CherryPick is a solution to predict multiple resources at VM and application level, this include recommending CPU, Disk, RAM, Network resources, their main objective is to reduce cost. This means their recommendations are static to create the VM, not a dynamic recommendation of resources like is proposed in our paper. They include SQL applications, such as TPC-DS and TPC-H, as well as machine learning workloads.
- *Adaptations considerations:* We adopted the Gaussian Process formulation employed by CherryPick in their Bayesian Optimization engine (Section 3.5 in the original publication) as the foundational approach for cost estimation. However, we adapted the input feature space to reflect our system’s architectural constraints: rather than utilizing global VM configuration parameters (instance type, CPU allocation, network bandwidth, memory allocation), we employed stage-level performance indicators including task count, executor allocation, partition sizes, and related metrics to estimate cost (in our case the target variable ExecutorRunTime). This transition represents a substantial distribution shift in the feature space and data characteristics compared to

CherryPick’s formulation, while maintaining the same modeling approach. CherryPick’s training dataset comprises interventional data derived from 66 distinct cloud configurations, systematically varying VM instance types, CPU cores, network specifications, and memory allocations (as detailed in Section 5.1 from their paper). In contrast, our evaluation utilizes 10 configuration variants, reflecting constraints in experimental data collection feasibility. Furthermore, the resource recommendation mechanisms differ fundamentally: CherryPick employs a Bayesian Optimization allocation process that leverages the Gaussian Process prior for configuration recommendation, whereas our approach implements the cost-effective solver described in Section 3.4 of our paper with $\theta_r = 0.0$. These methodological differences in data variations, workload characteristics, prediction granularity, configuration space, and resource allocation methodology represent significant constraints in our experimental setup and may impact the performance of our adaptation relative to CherryPick approach.

- *Grid search parameters:* The final model configuration was selected through a grid search procedure that systematically evaluated the hyperparameter space across four critical dimensions: regularization parameter (alpha), target variable normalization strategy, optimizer restart behavior, and kernel specifications.

C.2 Input Features

TPC-DS ReLoca# features: Number of Tasks, NumberExecutors, InputBytesperStage, PartitionsperStage, NumberOperations, DAGwidth, DAGdepth, DataSourceRDD, FileScanRDD, MapPartitionsRDD, ParallelCollectionRDD, ShuffledRowRDD, UnionRDD, ZippedPartitionsRDD2, CartesianColumnarBatchRDD, GlutenWholeStageColumnarRDD, ShuffledColumnarBatchRDD, WholeStageZippedPartitionsRDD.

SQLStorm ReLoca# features: Number of Tasks, ExecutorCores, NumberExecutors, InputBytesperStage, PartitionsperStage, NumberOperations, DAGwidth, DAGdepth, DataSourceRDD, FileScanRDD, MapPartitionsRDD, ParallelCollectionRDD, ShuffledRowRDD, UnionRDD, ZippedPartitionsRDD2, CartesianColumnarBatchRDD, GlutenWholeStageColumnarRDD, ShuffledColumnarBatchRDD, WholeStageZippedPartitionsRDD.

CherryPick# features, both TPC-DS and SQLStorm: Number of Tasks, NumberExecutors, InputBytesperStage, PartitionsperStage, NumberOperations, DataSourceRDD, FileScanRDD, MapPartitionsRDD, ParallelCollectionRDD, ShuffledRowRDD, UnionRDD, ZippedPartitionsRDD2, CartesianColumnarBatchRDD, GlutenWholeStageColumnarRDD, ShuffledColumnarBatchRDD, WholeStageZippedPartitionsRDD.

C.3 Model Availability

The trained regression models and their configurations are available in our repository¹² models/models_baselines directory, organized by dataset (TPC-DS and SQLStorm). Each model is stored as a ONNX and JSON files. The JSON files contain comprehensive model metadata including: the model name and type, hyperparameters

specific to each model architecture, cross-validation performance metrics, information about negative predictions, the best model instance metrics, target column name, run type classification, log transformation flag, feature column names used for training, training and test dataset sizes, and folder organization information. This structured format enables easy retrieval and deployment of models while maintaining full reproducibility of the modeling configurations. The feature sets include computational characteristics such as task counts, executor configuration, data volumes, partition information, operation counts, DAG topology metrics, and RDD type distributions, providing comprehensive coverage of the factors influencing query performance and resource utilization. The ONNX file contains the trained model ready to use at inference time.

C.4 Training Time Considerations

While this work compares performance and cost across methods, we need to note that training time can differ substantially between the implemented methods. On a M1 Max, 64GB RAM laptop, ReLoca# [9] trained in ≈ 12 seconds on both benchmarks, similar to our approaches, which require ≈ 11 seconds for both benchmarks. In contrast, CherryPick# [2] required ≈ 3 hours 39 minutes on SQLStorm and ≈ 8 hours 54 minutes on TPC-DS.

D EXTENDED RESULTS

Due to the aforementioned workshop’s PDF submission size limit, we provide (i) additional SHAP [14] plots to analyze feature importance and influence for both our *cost* and *performance* prediction models across benchmarks and training regimes, and (ii) end-to-end testing Scenario γ using models that predict DurationStage in the Online Appendix: <https://github.com/mrateike/pvldb-submission>.

Extended Per Query Results. We report additional results on the per query analyses in both benchmarks, namely the top-5 queries with the largest divergence in cost from 0-shot and Apache Spark™ approach for both benchmarks:

- Table 9: top-5 queries for TPC-DS
- Table 10: top-5 queries for SQLStorm

Additional radar plots for cost and performance across all queries and benchmarks can be found in the Online Appendix.

Data Resources. In the Online Appendix¹³, data/perf-cost_tcpds_baselines_duration.csv and data/perf-cost_tcpds_baselines_maxbycore.csv contain aggregated statistical summaries of performance and cost metrics for the TPC-DS benchmark comparing different strategies. Columns provide mean and standard deviation values for both Performance (Perf. mean, Perf. std) and Cost (Cost mean, Cost std) metrics across multiple query executions. The dataset includes six optimization strategies: 0-shot approaches with varying theta parameters ($\theta_r = 0.0, 0.25$, and 0.5), three n -shot approaches with corresponding theta settings ($\theta_r = 0.0, 0.25$, and 0.5), and the Apache Spark™ baseline for comparison.

data/perf-cost_sqlstorm_maxbycore_per_query.csv contains performance and cost metrics for the SQLStorm benchmark across multiple proposed methods and baseline strategies. Each record includes four key metrics: Performance (Perf.), Cost, both

¹²<https://github.com/mrateike/pvldb-submission>

¹³<https://github.com/mrateike/pvldb-submission>

Table 8: Baseline hyperparameter configurations and performance metrics. DurationStage (DS), ExecutorRunTimeMaxbyCore (Max). CherryPick# (CP) and ReLoca# (RLo).

Dataset	Model	Target	Log	Parameters	Train	Test	Mean MSE	Mean R^2	Best MSE	Best R^2
TPC-DS	CP	DS	True	alpha: 1e-06, normalize_y: true, n_restarts_optimizer: 1	29,721	7,431	8.04	0.829	7.25	0.834
TPC-DS	RLo	Max	False	hidden_layer_sizes: [20,10,10,10,10], activation: logistic, learning_rate_init: 0.0001, max_iter: 1000, solver: adam	49,580	12,395	27.11	0.335	13.42	0.591
TPC-DS	CP	Max	True	alpha: 1e-06, normalize_y: true, n_restarts_optimizer: 1	29,721	7,431	3.34	0.893	2.81	0.938
TPC-DS	RLo	DS	False	hidden_layer_sizes: [20,10,10,10,10], activation: tanh, learning_rate_init: 0.0001, max_iter: 1000, solver: adam	49,580	12,395	44.22	0.207	36.07	0.242
SQLStorm	CP	Max	True	alpha: 1e-06, normalize_y: true, n_restarts_optimizer: 1	49,580	12,395	6.55	0.798	3.35	0.932
SQLStorm	RLo	DS	False	hidden_layer_sizes: [10,5,5,5,5], activation: relu, learning_rate_init: 0.001, max_iter: 200, solver: adam	19,858	4,965	124.52	-1.233	56.14	-0.001
SQLStorm	CP	DS	True	alpha: 1e-06, normalize_y: true, n_restarts_optimizer: 1	49,580	12,395	18.16	0.662	10.55	0.776
SQLStorm	RLo	Max	False	hidden_layer_sizes: [10,5,5,5,5], activation: relu, learning_rate_init: 0.001, max_iter: 200, solver: adam	19,858	4,965	103.21	-1.540	41.15	0.003

Table 9: Top-5 queries with largest cost divergence, TPC-DS

Query	Cost 0-shot	Cost Apache Spark™	Cost Difference
14b	7,209,175	16,333,383	9,124,208
03	528,223	9,253,531	8,725,308
14a	6,705,714	14,700,672	7,994,958
23b	5,357,945	9,911,296	4,553,351
04	6,476,707	10,448,988	3,972,281

Table 10: Top-5 queries w. largest cost divergence, SQLStorm

Query	Cost 0-shot	Cost Apache Spark™	Cost Difference
3053	1,416,380	5,810,624	4,394,244
1296	3,088,394	5,723,702	2,635,308
3020	910,406	3,111,072	2,200,666
23239	694,750	2,267,374	1,572,624
749	561,207	1,674,908	1,113,701

in milliseconds, Query identifier, and method (optimization strategy). The dataset encompasses multiple run methods including 0-shot/THETA-0.0, 0-shot/THETA-0.25, 0-shot/THETA-0.5, n-shot/

THETA-0.0, n-shot/THETA-0.25, n-shot/THETA-0.5, VANILLA baseline, and CHERRY-PICK strategies. Where 0-shot = 0-shot approach, n-shot = n -shot approach, VANILLA = Apache Spark™ run. The data represents multiple executions per query (indicated by repeated Query IDs with varying Perf. and Cost values), allowing for analysis of performance variability and consistency across different approaches. Similarly data/perf-cost_tpcds_maxbycore_per_query.csv contains same metrics for TPC-DS.

E HARDWARE AND TRAIN/TEST SETUP

Cluster Hardware Configuration. The data collection and model evaluation hardware configuration includes 20 OpenShift worker nodes, each equipped with 4 x Intel Xeon E5-2683 v4 CPUs running at 2.10 GHz (16 cores per CPU, 64 physical cores per node), 1.5 TB of RAM, and 128 GB NVMe local storage. Hyper-Threading is enabled, providing 128 logical CPUs per node. The nodes are interconnected using dual 25 Gbit Ethernet links. Persistent data is stored on an S3-compatible object store accessed via the S3A connector.

Data Processing Infrastructure. Our data processing infrastructure leverages Apache Spark™ 3.5.4 integrated with Iceberg as the underlying table format. Iceberg is configured with a Hive-backed catalog and an external Hive Metastore for metadata management

while table data is stored in S3-compatible object storage accessed via the S3A connector. The execution engine is accelerated using Gluten with the Velox native backend, enabling columnar execution and native vectorized processing.

Train/Test Configurations. For Training, we shut down an executor after 60 seconds of idleness, following the default Apache Spark™ configuration. The maximum available number of executors is 128. For testing, Apache Spark™ queries Mira for updates on

the recommended number of cores computed by our optimization framework at 300-millisecond intervals. Multiple stages may run in parallel as long as their combined recommended executors remain within a limit of 128 executors. For our methods and CherryPick#, any executor that becomes idle is shut down immediately when no further tasks remain for its active stage. The Apache Spark™ baseline uses the default 60-second executor idle timeout.