Rockhopper: A Robust Optimizer for Spark Configuration Tuning in Production Environment

Yiwen Zhu
Microsoft
Mountain View, USA
yiwzh@microsoft.com

Rathijit Sen
Microsoft
Redmond, USA
rathijit.sen@microsoft.com

Brian Kroth
Microsoft
Madison, USA
bpkroth@microsoft.com

Sergiy Matusevych
Microsoft
Redmond, USA
sergiym@microsoft.com

Andreas Mueller
Microsoft
Mountain View, USA
amueller@microsoft.com

Tengfei Huang
Microsoft
Beijing, China
tengfeihuang@microsoft.com

Rahul Challapalli
Microsoft
Mountain View, USA
rachalla@microsoft.com

Weihan Tang
Microsoft
Beijing, China
weihantang@microsoft.com

Xin He
Microsoft
Beijing, China
xinhe1@microsoft.com

Mo Liu
Microsoft
Mountain View, USA
mol@microsoft.com

Estera Kot
Clouds on Mars
Warsaw, Poland
estera.kot@cloudsonmars.com

Sule Kahraman
Microsoft
New York, USA
sulekahraman@microsoft.com

Arshdeep Sekhon
Microsoft
Fort Mill, USA
asekhon@microsoft.com

Dario Bernal
Microsoft
Cambridge, USA
dariobernal@microsoft.com

Aditya Lakra
Microsoft
Redmond, USA
adityalakra@microsoft.com

Shaily Fozdar
Dynamo AI
New York, USA
shailyfozdar@gmail.com

Dhruv Relwani
Microsoft
Redmond, USA
dhrelwan@microsoft.com

Rui Fang
Microsoft
Beijing, China
rufan@microsoft.com

Long Tian
Microsoft
Beijing, China
ltian@microsoft.com

Karuna Sagar Krishna
Microsoft
Redmond, USA
karkrish@microsoft.com

Ashit Gosalia
Microsoft
Redmond, USA
ashitg@microsoft.com

Carlo Curino
Microsoft
Redmond, USA
ccurino@microsoft.com

Subru Krishnan
Microsoft
Barcelona, Spain
subru@microsoft.com

Abstract

Apache Spark, renowned for its scalability and ease of use, has become the standard for big data processing. However, optimizing Spark performance in production environments poses significant challenges. Traditional machine learning-based configuration tuning methods often necessitate extensive resources, lengthy experimentation, and risk performance regressions. Observational noise in production environments further complicates the tuning process, leading to suboptimal results. This paper presents an adaptive, robust learning approach leveraging insights from benchmark workloads to improve production tuning strategies. We propose a Centroid Learning algorithm resilient to noise, minimizing regressions and prioritizing promising configurations, combined with a workload embedding technique for context-aware adaptation and transfer learning. Evaluations using benchmark and customer workloads show consistent performance gains. Released in June 2024 as part of the Microsoft Fabric Spark offering, even with dynamic and evolving workloads, the system delivers approximately a 20% performance improvement in production for customer workloads by only tuning three query-level configurations.

CCS Concepts

• **Computing methodologies → Machine learning algorithms**; *Modeling and simulation*; • **Information systems → Autonomous database administration**.

Keywords

Bayesian Optimization, Autotuning, Performance Tuning, Gradient Descend

ACM Reference Format:
Yiwen Zhu, Rathijit Sen, Brian Kroth, Sergiy Matusevych, Andreas Mueller, Tengfei Huang, Rahul Challapalli, Weihan Tang, Xin He, Mo Liu, Estera Kot, Sule Kahraman, Arshdeep Sekhon, Dario Bernal, Aditya Lakra, Shaily Fozdar, Dhruv Relwani, Rui Fang, Long Tian, Karuna Sagar Krishna, Ashit Gosalia, Carlo Curino, Subru Krishnan, et al.. 2025. Rockhopper: A Robust

Optimizer for Spark Configuration Tuning in Production Environment. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25), June 22–27, 2025, Berlin, Germany*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3722212.3724451

1 Introduction

Apache Spark has emerged as a formidable engine for big data processing due to its speed, scalability, and ease of use. With the increasing adoption of cloud computing, Spark has also found a place in the cloud, with various Spark offerings on cloud platforms, such as Google Cloud [9], Microsoft Azure [20, 21], Amazon AWS [2] and others [10]. These offerings provide easy access to Spark clusters and take away the burden of managing the infrastructure.

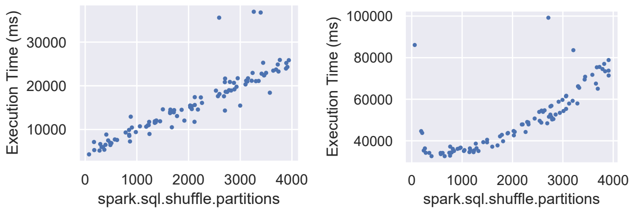
As Spark operates over distributed computing frameworks, the configuration of runtime parameters significantly influences its performance. Optimizing these settings remains a critical challenge [6, 16, 24, 46]. At the *application level*, for instance, parameters like spark.executor.instances and spark.executor.memory determine the number and size of Spark executors. Although increasing resources might expedite processing, insufficient allocations can lead to overheads or failures, necessitating a balanced approach to optimize resource use while adhering to customer expectations. At the *query level*, parameters such as spark.sql.shuffle.partitions crucially impact performance. As depicted in Figure 1, varying this parameter can significantly alter execution times, with each query reaching peak efficiency under different settings.

Several machine learning-based approaches have demonstrated success in modeling and optimizing Spark's performance through configuration tuning, including Rover [34], DAC [47], LOCAT [46], and others [11, 24, 28, 36]. Building on predictive models, our work integrates advanced workload characterization techniques to automate the tuning of Spark configurations on cloud platforms such as Microsoft Fabric, minimizing additional effort or cost for customers. However, deploying these machine learning methods in production environments presents several practical challenges:

You only run once—no access to customer workload (queries or data). Prior research often requires multiple rounds of "exploration" [24, 46], where customer workloads are executed under various configurations to develop an accurate performance model or to pinpoint critical tuning parameters. This method, although effective in controlled settings, poses significant challenges in a production environment, as it requires additional customer consent and incurs further costs, thereby increasing the complexity of system design. Moreover, repeated executions may lead to data overwriting, potentially disrupting ongoing customer operations.

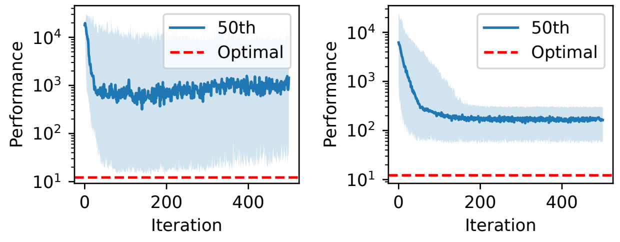
Risk of performance regression. Performance regressions can lead to significant repercussions. Relying on machine learning as a black-box approach introduces considerable risks, particularly when the model is trained on inadequate or noisy data. Such scenarios can result in the recommendation of unsuitable configurations, leading to substantial performance declines or even complete failures of cloud jobs.

Extremely noisy data. Machine learning algorithms, such as Bayesian Optimization [5, 13] and gradient descent approaches like FLOW2 [44], often struggle in production environments due to



(a) Query 33. **(b) Query 78.**

Figure 1: Query execution time for TPC-DS.



(a) Bayesian Optimization **(b) FLOW2.**

Figure 2: Performance convergence with noisy data.

the presence of extremely noisy observational data [8]. In our analysis within the Microsoft Fabric environment, two primary types of noise have been identified that complicate the optimization process:

- **Fluctuation Noise**: Characterized by random, small variations in execution time. These frequent deviations contribute to the variability in observational data, making it challenging to achieve consistent performance.
- **Performance Spikes**: Occasional, severe slowdowns where execution time increases by more than 2x, albeit with a lower probability.

Figure 2 illustrates the performance convergence of Bayesian Optimization and FLOW2, based on 200 runs of simulation. Query execution time was modeled as a convex function of configuration parameters, with noises incorporated to simulate real-world production environments (see Section 6 for more details). The solid line represents the median (50$^{\text{th}}$ percentile) performance, while the shaded regions depict the confidence interval between the 5$^{\text{th}}$ and 95$^{\text{th}}$ percentiles. We can see that both methods exhibit poor convergence, highlighting how noisy data with high variance and frequent outliers significantly hinders the performance of the algorithms.

Constantly changing workloads. In production environments, even with recurrent workloads, the outcomes often vary due to changes in input parameters, underlying data, variations in data skew and correlations, etc.. Such dynamics necessitate an autotuning mechanism that can swiftly and effectively adapt to these evolving conditions. Adding further complexity, the size of the data is often unknown at the start of a job, and cardinality estimation may either be unavailable or highly error-prone [46, 53].

Introduction to Rockhopper. To address the outlined challenges, we developed Rockhopper, an autotuning system integrated into Microsoft's Spark offering [23]. It performs efficient and robust online configuration tuning for recurrent workloads, requiring no customer intervention or additional experimentation. To ensure data privacy, Rockhopper trains models individually for each query, tailoring tuning to specific recurrent queries. While prior research has focused on reducing the dimensionality of tuning parameters [15, 39, 48, 51], our work emphasizes a tuning algorithm that targets a small subset of configurations, enhancing convergence efficiency in production environments.

Rockhopper implements a robust tuning algorithm that minimizes the risk of severe performance degradation. In production settings, avoiding performance regression is critical. Our novel tuning algorithm introduces a Centroid Learning (CL) algorithm and guardrails to prevent significant performance regressions during the tuning process by restricting the exploration space for configuration candidates and disabling the tuning process when regression is detected. This approach reduces performance variability and ensures more stable recommendations.

Centroid Learning incorporates an effective "de-noising" mechanism. To mitigate the impact of noisy data, the Centroid Learning algorithm evaluates observations with reduced sensitivity to noise, enhancing convergence during tuning. Our learning strategy demonstrates exceptional robustness, even under conditions of extreme data noise.

An offline learning phase boosts Rockhopper's performance by experimenting with open-source benchmark queries. To understand the impact of different configuration parameters on query performance, we developed an offline exploration pipeline capable of executing arbitrary benchmark workloads with varying Spark cluster sizes and configurations. This pipeline trains a warm-start baseline model using experimental data from well-known benchmarks, as customer data is unavailable due to privacy constraints.

Rockhopper's innovative workload embedding enables transfer learning from benchmarks to production workloads. To integrate knowledge collected offline, our Centroid Learning algorithm incorporates workload embeddings as additional "context" in the surrogate model for candidate selection. While previous works have used direct features such as data size in performance models [34, 46, 47], our implementation expands the feature set with embeddings that capture detailed query characteristics, leveraging insights from cutting-edge research [53].

Contribution. Our contributions are summarized as follows.

- We design a robust and efficient learning algorithm with built-in safeguards to prevent performance regressions during the tuning process.
- We utilize workload embeddings to enable transfer learning from benchmarks to dynamic production workloads, accelerating convergence through warm-starts.
- We perform a user study to understand user preferences for the autotuning feature and the tuning patterns employed by domain experts.
- The proposed feature is integrated into Microsoft Fabric Spark and publicly released in June 2024 [23], delivering approximately a 20% performance improvement in production for customer workloads.

The remainder of this paper is organized as follows: Section 2 provides an overview of Azure Spark offerings and presents customer survey results on autotuning. Section 3 outlines the design and architecture of Rockhopper. Section 4 describes the configuration tuning algorithm, while Section 5 details its implementation. Section 6 presents experimental results, and Section 7 reviews related work. Finally, Section 8 discusses remaining challenges and highlights directions for future research.

2 Background

In November 2023, Microsoft introduced Fabric, an all-in-one platform designed to streamline various data-related tasks, including database management, analytics, messaging, data integration, and business intelligence. With its intuitive interface, Fabric aims to provide seamless onboarding, provisioning, and autonomous management capabilities [54].

2.1 User Study

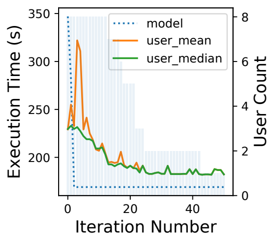
To evaluate Microsoft Fabric's Spark performance and gain insights into user preferences, we conducted a study using workloads from an existing Microsoft Spark platform. The findings revealed that over 95% of queries relied on default configuration settings, despite the availability of customizable parameters. We conducted extensive interviews with six key Microsoft customers whose workloads ranged from ad-hoc, large-scale jobs with complex DAGs and vast data, to regular batch jobs running on monthly or hourly cadences. Their workloads exhibited significant variance in job size, from "micro-batch" jobs lasting a few minutes to long-running jobs exceeding 20 hours, as well as exploratory notebook jobs and streaming workloads. Our analysis yielded several key insights:

- Nearly all customers reported tuning configurations related to memory and core size.
- Some customers mentioned tuning configurations related to partitions, garbage cleaning, and heartbeat monitoring.
- All customers valued execution time, but some teams with particularly large resource utilization or fixed budgets also noted the importance of cost.
- All customers expressed enthusiasm for auto-tuning solutions but emphasized the need for robust monitoring capabilities and transparency in understanding the configuration decisions and the underlying tuning algorithms.

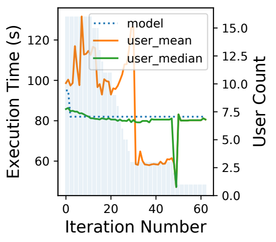
These findings informed our design of an automated tuning process aimed at improving performance and reducing costs while enhancing the overall user experience.

2.2 Manual Tuning Experiments

We conducted an experiment to understand the manual tuning process employed by our Spark experts. We developed a simulation platform for Spark configuration tuning, where users select configurations and iteratively "execute" queries. Instead of running actual queries, the platform predicts execution times using a baseline model trained on data from over 275 configuration combinations for all TPC-DS/TPC-H queries. After each configuration choice, the platform provides the predicted execution time alongside a visualization of the results from all prior choices, enabling participants to assess the impact of their decisions in real-time. We recruited over 50 volunteers to tune 5 queries across 7 Spark configurations, resulting in more than 4,000 query execution records. The configurations included spark.sql.files.maxPartitionBytes, spark.sql.autoBroadcastJoinThreshold, spark.sql.shuffle.partitions, spark.executor.instances,



(a) Query 1.



(b) Query 2.

Figure 3: Manual tuning versus Bayesian Optimization

spark.executor.memory, spark.memory.offHeap.enabled, and spark.-memory.offHeap.size.

For each query, users typically performed between 0 and 40 tuning iterations, generally achieving performance levels comparable to those suggested by the model by the end of the process. The tuning results for selected queries are illustrated in Figure 3, where solid lines represent the average execution times across all users per iteration. Dashed lines indicate the results obtained through model-based tuning using vanilla Bayesian Optimization as the baseline, in which each query is supported by a fine-tuned surrogate model. While the model generally converged faster, domain experts occasionally achieved better results. However, the model sometimes became stuck in local minima.

3 Overview

In this section, we present an overview of the Rockhopper design principles and architecture.

3.1 Design Principles and Choices

The following core design principles guided the architectural development of Rockhopper:

Reducing the number of experiments needed. We adopt two strategies to reduce the number of experiments. First, an offline phase leverages open-source benchmark queries to transfer knowledge to customer workloads. Second, a model-guided search improves convergence speed.

Reducing inference latency. The inference latency is on the critical path of the job submission/execution. We reduce the latency by (i) constraining the candidate search areas using a Centroid Learning algorithm, and (ii) pre-computing application-level configurations for recurrent workloads based on query statistics.

Scalability. This work focuses on per-query tuning, which proposes tailored configuration adjustments for each query. Given the high volume of queries processed daily by Fabric, this approach introduces significant scalability challenges. We consider two hosting options for the training pipeline—specifically, the feature ETL streaming job and the ML training pipeline: (i) the customer's Azure workspace or (ii) the admin workspace managed by the engineering team. Each option has trade-offs in terms of cost, maintainability, and operational complexity. The admin workspace is typically easier to maintain since the engineering team has direct access to resources. However, it poses challenges such as (1) increased compliance requirements for accessing customer data, and (2) scalability concerns, as it must handle models for all customer workloads. Conversely, hosting in customer workspaces avoids privacy and compliance issues but limits access.

Based on these considerations, we chose to host the pipeline in the admin workspace, leveraging scale-up and scale-out options for compute instances. To ensure compliance, we established a comprehensive data privacy and security review process.

Choice between Scala and Python. Python fulfills most of our requirements due to its API stability and extensive library support, including: (i) feature construction (e.g., adding interactions and permutations to the feature set), (ii) ONNX [25] model training and saving (not available in Scala), and (iii) access to libraries like Scikit-learn [27], NimbusML [22], and Bayesian Optimization [4]. For inference, to minimize and simplify communication between Python and Spark, we retrieve model in Scala. Models are trained in Python, converted to ONNX, and loaded in Scala using the ONNX Spark runtime. This setup ensures efficient deployment of surrogate models used in the Centroid Learning algorithm.

3.2 End-to-end Pipeline

The Rockhopper process consists of two phases: (1) an *offline phase* and (2) an *online phase*. In the offline phase, we developed an experiment platform to run open-source benchmarks (e.g., TPC-DS) under various Spark configurations and pools. This process collects extensive training data to analyze query performance across configurations. Using this data, we trained a *baseline model*, a regression model that predicts performance based on query characteristics and configuration settings. This model serves as the default *surrogate model* in the Centroid Learning algorithm, similar to Bayesian Optimization [4], providing a warm-start at iteration 0. By leveraging benchmark insights, the baseline model improves configuration recommendations for customer workloads in early iterations. In the online phase, when customers enable the "autotune" feature, Rockhopper recommends optimal Spark configurations at query or application start, applies them, and updates the surrogate models with observations after execution.

4 Algorithm

In this section, we discuss the tuning algorithm. Inspired by Bayesian Optimization and gradient descent methods, we develop an innovative tuning algorithm that combines the two to avoid potentially "risky" configurations and leverage ML models to improve efficiency. Section 4.1 discusses the surrogate model that supports the model-guided search. Section 4.2 discusses the offline phase. Section 4.3 discusses the online phase, i.e., Centroid Learning algorithm. Section 4.4 discusses the modification of the algorithm for pre-computing of the app-level configurations at the app-startup time where all the query-level characteristics are unknown.

4.1 Surrogate Model

In this section, we discuss the surrogate model used in the online tuning process to select the optimal configuration from a set of candidates. This approach is analogous to the surrogate model employed in Bayesian Optimization [4]. In Bayesian Optimization, a surrogate model is trained to predict the performance of candidate configurations, thereby guiding the search process. This regression model takes configuration values as input and predicts their corresponding performance:

$$f([\text{Configs}]) = \text{Perf.} \quad (1)$$

At each iteration of Bayesian Optimization, the configuration that is expected to yield the best performance, according to the surrogate model, is returned as the next suggested candidate.

To enable effective generalization across diverse workloads with varying characteristics and input data sizes, we propose augmenting the input features with workload characteristics. This enhancement supports transfer learning using extensive data collected from benchmark workloads. The surrogate model in this work is formulated as shown in Equation (2), where the workload embedding encapsulates key characteristics of the workload, providing contextual information for optimization:

$$f([\text{Workload embedding, Configs}]) = \text{Perf.} \quad (2)$$

At inference time, the configuration with the highest predicted performance or the best performance guarantee is recommended.

Workload embedding. We utilize *workload embedding* to characterize workloads as the "context" in the surrogate model defined in Equation (2). Effective workload characterization significantly enhances tuning performance for unseen workloads by capturing their unique features. The key intuition is that workloads with similar contexts are expected to exhibit comparable performance behavior for a given set of configurations.

We leverage simple and efficient workload embedding schemes as in [14, 53] to extract information related to the query optimizer that is available at compile time, without requiring additional training for feature extraction. Each workload embedding, represented as a vector, comprises three components: (1) the estimated cardinality of the root node operator, (2) the total input cardinality of all leaf node operators, and (3) the frequency of operator occurrences within the execution plan. For operator occurrences, each vector entry corresponds to a distinct operator type. To enable a finer-grained vectorization of execution plans, we further introduce the concept of a *virtual operator*, which distinguishes physical operators based on variations in input and output sizes. Specifically, each physical operator is subdivided into multiple virtual operators according to the optimizer's estimates of input and output row counts.

As illustrated in Figure 4, virtual operators are introduced for the Filter operator to capture distinctions between input and output sizes. For example, Filter1 and Filter2 may share a common virtual operator type when their outputs are small relative to their inputs. Conversely, Filter2 may also represent a second type of virtual operator based on different size thresholds. In this case, the execution plan contains two virtual operators of Filter-Type-I and one virtual operator of Filter-Type-II. In our experiments, we fine-tune the clustering thresholds for input and output sizes based on end-to-end performance optimization. The virtual operator counts are subsequently incorporated into the workload embeddings.

A potential direction for future work is to introduce more comprehensive workload characterization methods that incorporate complex execution plan structures, such as those proposed in [43]. While the current approach fine-tunes the prediction model on a per-query basis, more general predictors would require richer embedding techniques to improve the model's generalizability.

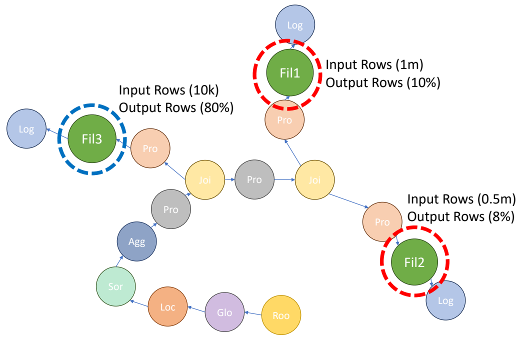


Figure 4: Virtual operator.

4.2 Offline Phase: Training Baseline Model

To enhance the accuracy of the surrogate model, we propose an offline process to gather observations from benchmark workloads, thereby enriching the model's knowledge base. This data is used to train a *baseline model*, which serves as the initial iteration when query-specific observations are unavailable. Subsequently, the model is fine-tuned with new observations during the tuning process.

The offline phase of Rockhopper consists of two main components: (1) an experimental platform, referred to as the "*flighting pipeline*," which executes open-source benchmarks and collects data points to train the surrogate model (see Equation (2)); and (2) a machine learning (ML) training pipeline used to construct the baseline model.

The flighting pipeline operates based on a configuration file that specifies essential parameters, including the benchmark database (e.g., TPC-DS, TPC-H), query name, scaling factor, number of runs, pool ID (linked to node configurations), and the Spark configuration generation algorithm (currently set to "`Random`"). Future work may explore enhancing the efficiency of configuration generation. After executing the queries, the pipeline triggers an ETL job (see Figure 7) to process logs and train the model.

For each region, we develop a baseline surrogate model using execution traces. At the beginning of the tuning phase, the surrogate model is fine-tuned for the specific query signature [30] (each corresponds to a distinct query execution plan), leveraging both query-specific observations and benchmark workload data. To ensure data privacy, information sharing between users is restricted. Models are trained exclusively with baseline data and query traces originating from the same user and query signature.

4.3 Online Phase: Centroid Learning

In this section, we introduce the Centroid Learning algorithm, a novel approach that seamlessly combines the strengths of two prominent configuration optimization strategies: greedy search (e.g., hill-climbing [26], FLOW2 [44], and OPPerTune [35]) and model-guided search (e.g., Bayesian Optimization [4]). The algorithm is designed to enhance the online tuning process.

Figure 5 depicts the online tuning process facilitated by the Centroid Learning algorithm. Upon user submission of a query (Step 0), the algorithm begins by leveraging the *baseline model*, trained offline as a surrogate model (see Section 4.1), to select the best configuration for the first iteration of tuning, where no query-specific data is yet available to train ML models. Various acquisition

]

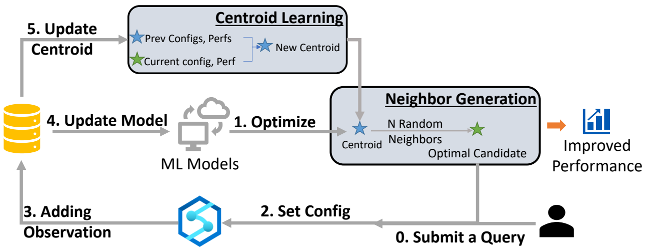


Figure 5: CBO with centroid learning.

functions, such as Expected Improvement (EI) [4], can be used as selection criteria [7, 34, 46]. The search subspace is defined as the neighborhood around the default configuration based on a step size (Step 1). Using the surrogate model, the candidate configuration with the highest acquisition function score is selected and applied to the submitted query (Step 2).

After the query execution, the observed performance is recorded as new entries in the storage system (Step 3), and the surrogate model is subsequently updated with this new data (Step 4). Based on the updated observations, the algorithm computes a new centroid using *statistical insights* derived from the collected data (Step 5), where the optimal descending direction is inferred statistically from a group of observations. This centroid serves as the anchor for generating the next set of configuration candidates for subsequent query submissions. This iterative tuning process continues with each new submission of the same recurrent workload, progressively improving the configuration selection and performance outcomes.

Algorithm 1: Centroid Learning

```
Input: Initial centroid $e_0$, centroid update step size $\alpha$, candidate
       generation step size $\beta$, observed performance $r$ of candidate
       $c$ with data size $p$, acquisition function $f$, at iteration $t$ latest
       $N$ observations for centroid updates
       $\Omega(t,N) = \{(c_i, p_i, r_i) \mid t+1-N \le i \le t\}$
$e_t \leftarrow e_0$ ; while not stopping criterion do
    Generate candidate set in the neighborhood of centroid $e_t$
      based on $\beta$: $C(e_t) = \{c^{(1)}, ..., c^{(n)}\}$ ;
    Use surrogate model to select the best candidate:
      $c_{t+1} = \arg\max_{c \in C} f(c)$ ;
    Execute query with config $c_{t+1}$, obtain $p_{t+1}$, and $r_{t+1}$;
    Find best candidates in the latest $N$ iterations:
      $c^* = \text{FIND\_BEST}(\Omega(t+1, N))$;
    Using the latest $N$ observations, obtain the gradient for each
      config dimension to descend towards:
      $\Delta = \text{FIND\_GRADIENT}(\Omega(t+1, N))$;
    Update centroid: $e_{t+1} \leftarrow c^* - \alpha \cdot \Delta$ ;
    Update time stamp: $t \leftarrow t+1$ ;
```

Algorithm 1 provides a detailed overview of the Centroid Learning process. Starting from the default configuration, each iteration involves selecting candidates within the neighborhood $C(e_t)$ that maximize the acquisition function f, denoted as c_{t+1}. After executing c_{t+1}, its performance r_{t+1} and corresponding data size p_{t+1} are recorded. The new centroid is determined based on two components: (1) c^*, the *best configuration* observed over the latest N iterations, evaluated using performance and associated data sizes, and (2) the *gradient* Δ, learned by fitting a regression model to the latest N observations. A linear surface is employed to approximate the small region explored in these iterations, enabling robust gradient calculation that mitigates noise while excluding the effects of data size variations. Specifically, the features include the configuration parameters in c and the data size p. The next centroid is derived by advancing from the current optimal configuration c^* in the direction of the learned gradient, scaled by a factor α. This approach deliberately *overshoots* the gradient direction at each iteration, drawing inspiration from the momentum effect in gradient descent algorithms for Deep Neural Networks (DNNs) [29, 37]. The overshooting strategy enhances the algorithm's ability to escape local minima, improving the likelihood of converging to a globally optimal configuration. While this paper focuses on continuous configurations, categorical configurations can be handled by employing embedding algorithms that map categorical values into a continuous space to enable tuning [50].

FIND BEST. Algorithm 1 incorporates the FIND_BEST function, which has undergone three iterations of refinement. Given the latest N observations, $\Omega(t,N)$, the simplest approach selects the candidate with the shortest execution time. While straightforward, this approach is inadequate when data sizes vary, as it may favor candidates with minimal data sizes, even if their overall performance is suboptimal. To address variations in data size, the second version of the function selects the candidate with the shortest *normalized* execution time:

$$c^* = \arg\min_{c_i \in \Omega} \frac{r_i}{p_i}. \tag{3}$$

This normalization penalizes candidates with smaller data sizes, providing a fairer comparison. However, we observed that for the same configuration, the ratio $\frac{r_t}{p_t}$ often decreases as p_t increases, leading to biased comparisons across observations with different data sizes. The final version employs machine learning (ML) to model the relationship between configuration, data size, and performance, enabling performance predictions under a uniform data size. A regression model H is trained using the observations in Ω:

$$r_i = H(c_i, p_i) + \epsilon_i, \tag{4}$$

where ϵ_i represents the error term. Using the fitted model H, performance predictions for each candidate are made by fixing the input data size to a constant value, such as p_t. The best candidate is then identified based on the predicted performance:

$$c^* = \arg\min_{c_i \in \Omega} H(c_i, p_t). \tag{5}$$

FIND GRADIENT. The intuition behind the gradient is to estimate whether increasing or decreasing a specific configuration value will lead to performance improvement or degradation based on observed trends. It is important to emphasize that the derived gradient at this step indicates only the direction of change (increase or decrease), not the magnitude. To control the adjustment's scale, we introduce a parameter, α, which can be fine-tuned to determine the step size in the descent.

Figure 6 illustrates a simple example based on observations from the last six iterations, Ω. In this example, we learn the performance trend with respect to spark.sql.shuffle.partitions. The observed trend shows that increasing the value of this configuration results in performance degradation. As a result, the centroid should move

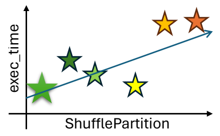


Figure 6: Learning the trend for the gradient.

in the direction of decreasing this configuration in subsequent iterations. Learning performance trends in this manner helps mitigate the impact of significant noise in the data, particularly when configuration changes (x-axis) are relatively small.

To relax the assumption about the relationship between data size and performance, we introduced a non-linear regression approach. This method enables the reuse of the ML model fitted in Equation (4) to learn the local gradient around the best candidate identified using Equations (3), or (5):

$$\Delta = \arg\min_{\delta \in \mathcal{D}} H\left(c^*(1-\alpha\delta), p_{t+1}\right), \tag{6}$$

$$\mathcal{D} = \{(1,1,1), (1,1,-1), \ldots, (-1,-1,-1)\}, \tag{7}$$

where Δ represents the "candidate gradient", determined based on predictions from the ML model. This gradient reflects the local trend around the best configuration identified from prior observations.

In production scenarios with varying data sizes, we observed that the use of a learned ML model consistently outperforms linear regression by avoiding additional assumptions about how data size impacts performance. Furthermore, the number of observations N should be sufficiently large (e.g., 10 or 20) to mitigate the influence of significant noise often present in production data. This ensures more reliable gradient estimation and robust tuning outcomes.

Additional guardrail. In the production system, we implemented a guardrail mechanism to monitor performance changes for each query over time. If continuous performance regression is detected over several consecutive iterations, the system identifies the query as unsuitable for autotuning. In such cases, the autotune feature is disabled, and the default configuration is reinstated.

Building on the insights from Figure 6, we developed a simple regression model to predict performance based on the *iteration number* and *input cardinality*. Starting at iteration 30, the model predicts the execution time for the next iteration. If this predicted value exceeds the execution time of the previous iteration by more than a predefined threshold, autotuning is deactivated for the query. This approach strikes a balance in the exploration budget for each query, ensuring that every query undergoes at least 30 iterations of tuning, even in the presence of performance regressions. If performance shows improvement over time, the tuning process proceeds as usual, allowing the system to optimize configurations effectively.

Summary. The Centroid Learning algorithm, as outlined in this section, provides several key advantages:

De-noising. The Centroid Learning algorithm leverages statistical insights to enhance exploration by utilizing the latest N observations to determine a statistically robust exploration direction, thereby effectively mitigating noise. Unlike methods such as hill-climbing [26], FLOW2 [44], or OPPerTune [35], which rely solely on the last two rounds of observations, our approach incorporates a broader set of observations to compute a descending gradient. In noisy environments, performance improvements observed in a single execution may be unreliable. By utilizing a larger pool of observations, as demonstrated in Section 6, the Centroid Learning algorithm achieves superior performance compared to traditional methods that depend exclusively on recent executions, effectively reducing the impact of noise.

Robustness: learning from failures. Through experimentation, we observed that even when the ML model fails to recommend an optimal candidate, the centroid update process still derives value from those observations by learning the correct gradient direction for the next iteration. This capability enables the algorithm to accelerate convergence, even when the surrogate model exhibits lower accuracy (see more discussion in Section 6).

Robustness: avoiding performance regression. The Centroid Learning algorithm restricts exploration to a smaller region defined by the step size β, minimizing the risk of significant performance regressions. This focused search space not only enhances robustness and safety but also reduces latency during model inference. By avoiding drastic jumps to entirely different regions, the algorithm prioritizes stability and ensures a more controlled and reliable optimization process, even if this approach sacrifices some potential performance gains.

Avoiding local minima. To prevent stagnation in local minima, the algorithm incorporates a momentum-like mechanism that overshoots in the exploration direction during updates as used in training deep neural networks, which has been shown to enhance optimization effectiveness [29, 37].

4.4 App-Level Configuration Optimization

Recall that in Spark, an *application* can execute one or more *queries* during its lifecycle. Tuning configurations at the application level (e.g., spark.executor.instances) differs fundamentally from query-level configuration optimization due to the following reasons:

- **Consistency requirements:** Application-level configurations must remain consistent across all queries within the application, as they are fixed at startup. In contrast, query-level configurations can be different across queries and are set at the query start time.
- **Limited information at startup:** Workload embeddings, which rely on input from the query optimizer (see Section 4.1), are only available after query submission. At application startup, details about upcoming queries and their embeddings are unavailable.

To address these challenges, we extend the existing tuning algorithm by developing a joint optimization framework for both application- and query-level configurations. This involves modifying Step 1 in Figure 5 and the argmax$f(c)$ computation in Algorithm 1. Additionally, we propose pre-computing application-level configurations after each application completes, enabling better initialization for future runs for the same recurrent workload.

Pre-compute app cache. We introduce a unique identifier, `artifact_id`, for each recurrent Spark application, derived from its artifact, such as a hash of a PySpark notebook or a Spark job description in JSON format. This identifier links each application to its historical observations within the same recurrent workload. At the end of each application run, the optimal application-level configuration

Algorithm 2: `App_cache` generation, modified argmax$f(c)$

```
Input: Number of app-level configuration candidates $M$, number of
       query-level config candidates $N$, set of queries $Q$, scoring
       function $f_q(c)$ for each $q \in Q$
Output: Best app-level configuration candidate
$V \leftarrow$ generate $M$ app-level config candidates around the
  neighborhood of the current setting;
for $q \in Q$ do
    $W_q \leftarrow$ generate $N$ query-level config candidates around the
      centroid of $q$;
    $C_q(v) \leftarrow \{v \times w_q \mid w_q \in W_q\}, \forall v \in V$;    // Cartesian
      product of app- and query-level config for each
      query
    $c_q^*(v) \leftarrow \arg\max_{c \in C_q(v)} f_q(c)$; // config candidate with
      the best acquisition function for query q given
      app-level config v
end
for $v \in V$ do
    $F(v) \leftarrow \sum_{q \in Q} f_q(c_q^*(v))$;    // Score app-level config
      candidate
end
return $\arg\max_{v \in V} F(v)$    // Return best app-level config
  candidate
```

is computed from the observed query embeddings and stored in `app_cache` under the corresponding `artifact_id`. When a new application is submitted, its `artifact_id` is used to retrieve the pre-computed `app_cache` for that job, bypassing additional computations and significantly reducing configuration inference latency.

***Joint optimization*.** For applications consisting of multiple queries, a joint optimization program is employed to compute the `app_cache` after the application has completed execution. Algorithm 2 describes the process: first, candidates for app-level configurations are generated. For each candidate, query-level configurations are created using the centroid of each query. The per-query surrogate model then selects the optimal query-level configuration, $c_q^*(v)$, for each app-level candidate v by maximizing the acquisition function f_q. The app-level configuration that achieves the highest total acquisition function score across all queries is selected. If the acquisition function is defined in terms of execution time, this approach minimizes the overall execution time of the application.

5 Implementation

Figure 7 presents an overview of Rockhopper's internals for the online phase. The Autotune Backend leverages cloud resources to process event files, train machine learning models, and infer optimal Spark configurations. The Autotune Clients run on customers' Spark clusters, retrieving model files and pre-computed configurations from the Autotune Backend and executing ML inference.

When a Spark job is submitted (Step 0), autotune-specific configurations, such as the application ID and `artifact_id`, are included in the job payload. These configurations enable Autotune Clients, including the model loader and query listener, to authenticate (e.g., via SAS URLs in Steps 1 and 2) and retrieve pre-computed optimal configurations, such as the number of executors and executor sizes, from the `app_cache` (Step 3) along with ML model files (Step 5). After query or application completion, Spark events are recorded (Step 6) to retrain ML models and refine app-level configurations. The Job Service applies these configurations to each Spark application (Step 4). For individual queries, Rockhopper's ML models guide the Autotune Config Inference module to determine optimal configurations before the physical planning stage. The following sections detail the implementation of each module.

Autotune Backend. The Autotune Backend securely manages storage for event files and models. Each Spark application is assigned a dedicated folder for event files, organized by its job ID, and another folder for its `artifact_id`, used for jobs with the same Spark definition. Restricted access is enforced through SAS URLs, ensuring data integrity across jobs. A Storage Manager oversees the cleanup of outdated event files to maintain GDPR compliance and prevent misuse [38]. The backend hosts three key streaming jobs: (1) the Embedding ETL, which processes Spark job logs; (2) the App Cache Generator, responsible for managing `app_cache`; and (3) the Model Updater, which updates query models and is triggered by new events in the Event Hub.

Authentication. To secure communication between Autotune and Spark jobs, the Autotune Backend generates SAS URLs for accessing models and writing event files. The Autotune Manager within Spark Core integrates seamlessly with Autotune by retrieving SAS URLs from the backend. It handles authentication, authorization, and request validation, ensuring that all requests originate from Spark clusters through the Fabric token service.

Autotune Client. Autotune Clients on Spark clusters manage configuration inference through the model loader and query listener components. These clients interact with Autotune storage to read models and write event files using SAS URLs. The AutotuneCredentialManager class handles SAS URL retrieval for both clients and the backend, leveraging the token library to communicate with the Autotune Manager. The URL of the Autotune Manager is provided as a Spark configuration during job submission, with SAS URLs being cached and refreshed as needed. Users can enable or disable the Autotune feature via the spark.autotune.query.enabled Spark configuration. The Autotune configuration inference client logs the suggested configurations along with their rationale, enhancing transparency and facilitating debugging.

6 Experiments

In this section, we present the experimental and deployment findings of Rockhopper, focusing on: (1) the evaluation with **synthetic functions** in Section 6.1, (2) the ablation study using **standard benchmark workloads** in Section 6.2, and (3) the deployment results with **production workloads** in Section 6.3.

6.1 Experiment with Synthetic Functions

To evaluate the tuning algorithms under significant noise levels, we design a synthetic optimization function that models the relationship between observed performance (e.g., execution time), data size, and three tunable configurations as a convex function. Figure 8 illustrates this function both before (dashed line) and after (solid line) adding noise for one of the configurations represented on the x-axis.

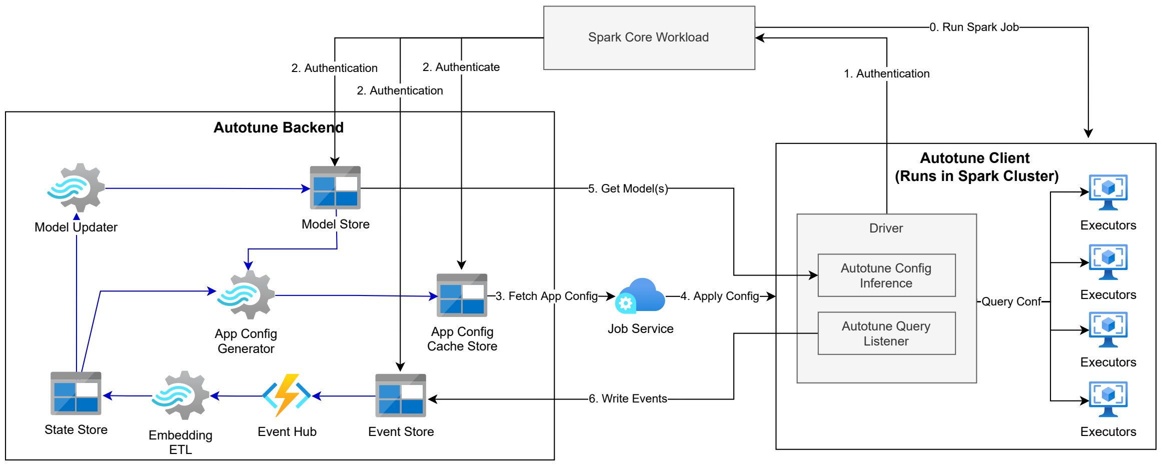
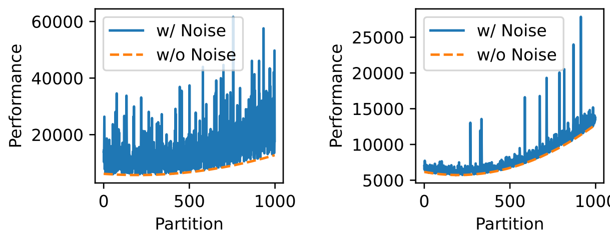


Figure 7: Architecture (online phase).



(a) High noise. **(b) Low noise.**

Figure 8: A synthetic function for optimization, where high noise corresponds to greater fluctuations and more frequent spikes, and vice versa.

The function exhibits a simple convex shape, reflecting typical relationships between configurations (e.g., spark.sql.shuffle.partitions) and execution time (y-axis), as shown in Figure 1.

We introduce two types of noise into the evaluation: (1) **fluctuation noise**, modeled as a Gaussian-distributed slowdown with a fluctuation level denoted by FL, and (2) **performance spikes**, where the execution time doubles with a probability determined by the spike level, denoted by SL. The observed execution time g for a configuration candidate c with data size p is modeled by injecting noise into the baseline execution time g_0. Drawing a random number $p \sim \mathcal{U}[0,1]$, the execution time is given by:

$$g = \begin{cases} g_0(1+|\epsilon|) & \text{if } p > \frac{\text{SL}}{10}, \\ g_0(1+|\epsilon|) \times 2 & \text{otherwise.} \end{cases} \tag{8}$$

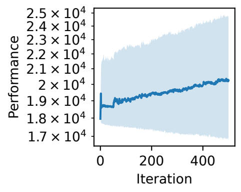
Here, $\epsilon \sim \mathcal{N}(0, \text{FL})$. For high noise levels, FL = 1 and SL = 1 (Figure 8a), whereas for low noise levels, FL = 0.1 and SL = 0.1 (Figure 8b).

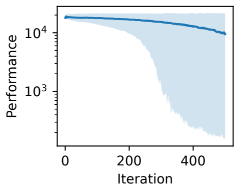
Impact of surrogate models. To assess the impact of surrogate models on the convergence to the optimal solution, we conduct experiments using pseudo-surrogate models with varying accuracy levels in predicting the "true" performance (i.e., performance in the absence of noise). An ideal surrogate model should identify the candidate within the set C that optimizes true performance as a function of data size. Conversely, an inaccurate surrogate model may select suboptimal candidates. We define a "Level 1" model as one that selects a candidate ranked approximately at the 10th percentile in true performance as optimal, whereas a less accurate "Level 8" model might recommend a candidate ranked near the 80th percentile. Such a decline in predictive accuracy can lead to significantly suboptimal decision-making outcomes.

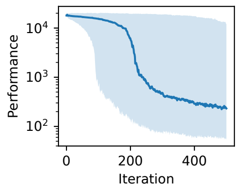
Constant workloads. We employ different pseudo-surrogate models that suggest candidates at various percentiles within the set C as specified by Algorithm 1. Figure 9 presents the convergence results when the surrogate model consistently selects candidates ranked at the $10 \cdot X^{\text{th}}$ percentile of the true performance for constant workloads with 100 runs. The solid line represents the median (50th percentile) of the true performance at each iteration across all runs. The shaded region illustrates the confidence interval, spanning from the 5th to the 95th percentiles. Notably, even when the surrogate model identifies candidates at Level 5 (selecting from the top 50% of the candidates), the algorithm demonstrates robust convergence (see Figure 9c), outperforming the baseline vanilla Bayesian Optimization algorithm (shown in Figure 2).

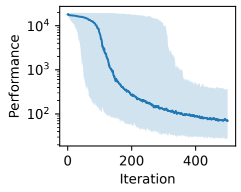
We replace the pseudo-surrogate model with a support vector machine regression model [31] trained on noisy data. Figure 10 shows the convergence results. Compared to Figure 9, the model tends to select candidates within the 30th to 50th percentiles for true performance, indicating moderate accuracy. Despite this, convergence remains satisfactory. The Centroid Learning algorithm consistently avoids poor performance throughout iterations and converges to the true optimum faster, as evidenced by the narrowing upper boundary of the shaded area. Compared to vanilla Bayesian Optimization (Figure 2a) and FLOW2 (Figure 2b), convergence improves significantly, demonstrating the robustness of the Centroid Learning algorithm even with a moderately accurate surrogate model. However, the results also suggest potential for further improvement in convergence by enhancing the surrogate model's accuracy (see Figure 9e).

Dynamic workloads. We simulate two types of dynamic workloads with high noise levels:


(a) Level 9.


(b) Level 7.


(c) Level 5


(d) Level 3.

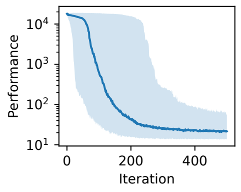
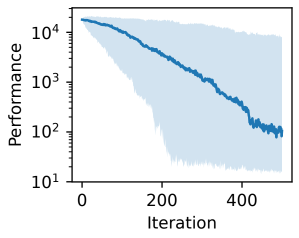

(e) Level 1.

Figure 9: Convergence results with varied surrogate models for constant workloads. Level X indicates that the model selects a candidate in the $10X^{\mathbf{th}}$ percentile within the set. Lower levels correspond to better performance (shown on the right).


(a) Performance.

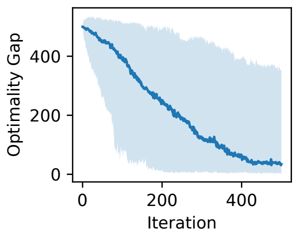
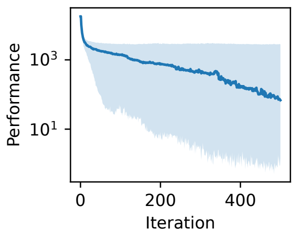
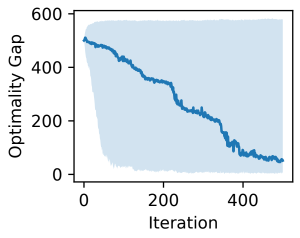
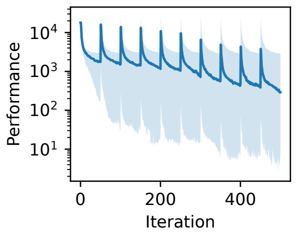

(b) Optimality Gap.

Figure 10: Convergence of the Centroid Learning algorithm for constant workloads, with accuracy comparable to Level 3 (Figure 9d) and Level 5 (Figure 9c). The results demonstrate significant improvements over existing algorithms (Figure 2).


(a) Linear increasing, normed-performance.


(b) Linear increasing, optimality gap.


(c) Periodic, normed-performance.

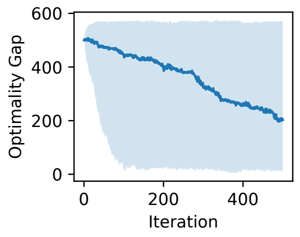

(d) Periodic, optimality gap.

Figure 11: With dynamic workloads.

- Workloads with data sizes increasing linearly over time;
- Workloads with periodic changes in data size, where the input data size follows $f(t) = t\%\%K$, a periodic function based on t with a period of K.

For both types of workloads, Centroid Learning converges to the optimal configuration (see Figure 11). To measure the optimality gap, we evaluate the absolute difference from the optimal value for the most impactful configuration, such as spark.sql.files.maxPartitionBytes as in Figures 10b and 11d.

6.2 Experiment with Standard Benchmark Workloads

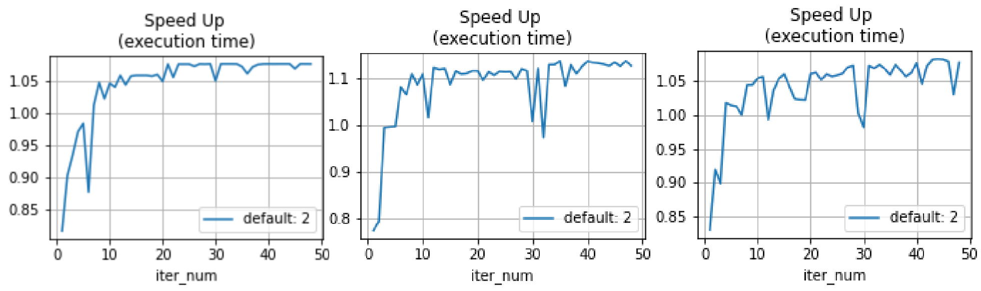
In this section, we discuss the performance of our algorithm on standard benchmark workloads like TPC-DS. In particular, we: (1) evaluate the impact of transfer learning, warm-starting from the baseline model (described in Section 4.2), as opposed to developing a new model based exclusively on the experimental data collected on the target query; (2) compare the performance of Centroid Learning and Bayesian Optimization algorithms; and (3) assess the impact of the new embedding based on virtual operators compared with the original embedding proposed by [53].

Transfer learning. We develop an experimental platform to test the algorithm's performance, specifically focusing on the vanilla Contextual Bayesian Optimization (CBO) to evaluate the impact of using the baseline model for a warm-start. The platform (V0) implements a synthetic evaluation method that proactively generates a large set of configuration performance data for each query. During inference, we restrict the candidate set to these pre-recorded configurations and use cached results without live query execution.

Figure 12 presents the results of tuning query-level configurations, where we evaluate over 275 configuration combinations per query. To account for noise in the tuning process, we assess the total execution time across all TPC-DS queries. We compare the results to the optimal configuration chosen by the Spark team through manual tuning (speedup = 1.0), which serves as the default for all users. The baseline model is trained on data sampled from all queries except the optimization target, using 100, 500, and 1000 random samples. We then fine-tune the baseline model by using it as a warm-start for Bayesian Optimization.

Interestingly, with 500 samples, the model converges to a better configuration than with 1,000 samples, resulting in performance improvements of 15% and 7%, respectively[1]. This indicates that additional samples beyond 500 reduce the model's adaptability, while insufficient samples limit the warm-start's effectiveness, making it harder to leverage existing insights. We also observe that

[1]Since we are tuning only three query-level configurations, this gain is expected.



(a) 100 samples. (b) 500 samples. (c) 1000 samples.

Figure 12: For Contextual Bayesian Optimization, with different baseline training sample sizes, the convergence is different.

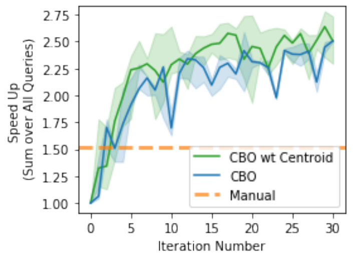


Figure 13: Centroid Learning vs. Bayesian Optimization (BO).

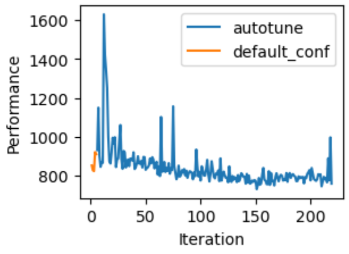


Figure 14: Total execution time for TPC-H queries.

the speedup is below 1 in early iterations, meaning the tuned performance is initially worse than the configuration hand-tuned by experienced Spark engineers. Our Centroid Learning approach (Section 4.3) avoids this problem by warm-starting the optimization from the best-known configuration.

Centroid Learning. Motivated by the suboptimal performance observed in the transfer learning experiments during early iterations as above, we develop the Centroid Learning algorithm (Section 4.3) to facilitate exploration from any configuration, ideally leveraging a known good configuration from manual tuning. This section compares the final convergence speed of Centroid Learning with that of the traditional Contextual Bayesian Optimization (CBO).

We introduce a new version of the evaluation platform (V1), referred to as the Lightweight Pipeline (LWP). The LWP integrates a Synapse [21] pipeline to submit and execute queries via notebooks, train machine learning models, recommend configurations, save configurations to a file, and apply the Spark configuration for query execution. Unlike the V0 platform, the LWP removes constraints on the candidate set, directly executes queries, and more accurately reflects the noisy environment of a real production setting.

Figure 13 illustrates the convergence results for both algorithms starting from an intentionally poor configuration (speedup=1.0), ensuring that the starting point does not influence the outcomes. The results confirm that Centroid Learning achieves significantly better final convergence than the CBO method, even under suboptimal starting conditions.

New workload embedding. We evaluate performance using (1) the workload embeddings proposed in [53], which are based on the count of different operator types (e.g., `join`, `scan`) as query features, and (2) the embedding method described in Section 4.1. The experiments involve executing 18 TPC-DS queries with a scaling factor of 1000G and assessing query-level configuration tuning using the LWP. Our results indicate that embeddings constructed from both regular and virtual operator counts result in significant performance gains for query-level configuration tuning. Starting from iteration 5, these embeddings yield an additional 5–10% improvement in performance consistently.

6.3 Deployment Results

In this section, we evaluate the configuration autotuning solution as deployed in Microsoft Fabric [23]. Starting very conservative about such large-scale ML system deployment, the team limit the configuration tuning to only three query-level configuration parameters: spark.sql.files.maxPartitionBytes, spark.sql.autoBroadcastJoinThreshold, and spark.sql.shuffle.partitions.

Benchmark workloads. In the production setting, we evaluate the algorithm using TPC-H workloads with a scale factor of 100 GB, while the baseline model is trained on TPC-DS data. Figure 14 shows the total query execution times across all 22 queries over multiple iterations. Each query is tuned independently, with configurations varying across iterations and between queries. Despite substantial noise and occasional runtime spikes, performance improves over time. For 10 queries, we observe performance gains exceeding 10%, with 6 of these showing improvements greater than 15%. Three queries exhibit minor regressions, with differences of less than 0.7 seconds, likely attributable to noise.

Internal customer workload—overall analysis. We also evaluate production performance using workloads from an internal customer, achieving an average performance improvement of 17% across more than 60 tested Fabric notebooks, with execution time improvements reaching up to 100%. The recurring workloads in production typically involve varying input sizes, adding complexity to the tuning process. Figure 16 illustrates the distribution of percentage speed-ups across all notebooks.

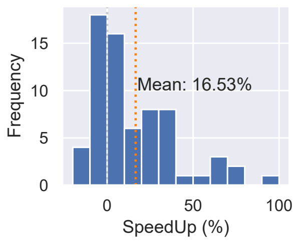


Figure 15: Percentage speed-up for customer workloads.

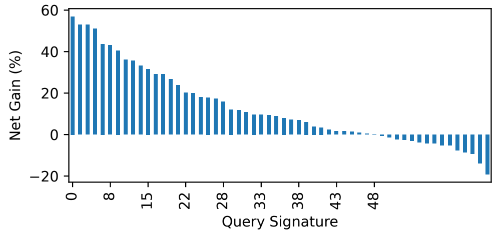


Figure 16: Speed-up for external customer workloads.

Internal customer workload—posterior analysis. A key component of Rockhopper is the monitoring dashboard, which facilitates real-time analysis of query tuning performance, enabling issue identification and deeper insights into workloads. The dashboard provides the following features:

- Visualization of configuration changes across iterations;
- Visualization of performance trends; and
- Evaluation of additional performance-impacting metrics.

Specifically, we collect metrics directly influenced by configuration suggestions, including: (1) partitions, (2) physical plans, (3) task numbers, and (4) input data sizes. This data enables us to explain performance changes, validate Rockhopper's configuration recommendations, and support Root Cause Analysis (RCA) for performance variations [1].

External customer workload—overall analysis. As the feature entered public preview [23], we analyzed Fabric Spark usage data from April 2024 to June 2024. During this period, over 300,000 query runs were recorded, each with more than 30 iterations per query signature. Autotuning was triggered in approximately 5% of these runs, accounting for over 10,000 instances. In total, we identified 416 unique query signatures, 73 of which consistently maintained autotuning throughout *all* iterations without being disabled by the extremely conservative guardrail settings (see Section 4.3).

Figure 16 shows the percentage speed-up distribution for all customer workloads with autotune enabled. The total execution time improves by approximately 20%. Among the queries exhibiting performance degradation exceeding 30%, one shows significant variance in execution time. Additionally, two queries exhibit regressions of more than 3× and 4×, likely due to factors unrelated to configuration changes[2]. With further iterations, the guardrail mechanism automatically disables autotuning for such queries. In production, we employ a conservative guardrail policy that enables autotuning only when query performance improves, which contributes to the overall performance gains observed.

[2]Although we attempt to exclude external impacts—such as changes in data size—by filtering queries with performance improvements associated with data size reduction, and vice versa.

7 Related

Black-box algorithms such as Bayesian Optimization and Reinforcement Learning [12] address the challenges of adapting to varying workloads and data sizes. LITE [18] and "You Only Run Once" [28] optimize Spark workloads using offline analysis, while UDAO [36], AutoExecutor [32, 33], and other studies [19, 45] focus on multi-objective tuning and parametric performance models. To reduce the dimensionality of tuning knobs, LOCAT [46] employs configuration-sensitive queries along with a data-size-aware Gaussian process surrogate model. Other approaches include Latin hypercube sampling [24], random sampling [11, 42], and SHAP-based configuration generation with workload similarity metrics [34]. OtterTune [39, 48] applies Bayesian Optimization and neural networks for performance prediction, while DBTune [51, 52] utilizes Lasso-based knob selection for efficient tuning.

Another line of research focuses on gradual tuning based on performance observations. CDBTune [49] used Reinforcement Learning with an actor-critic framework, while RFHOC [3] combined random forests and genetic algorithms for workload profiling. DAC [47] leveraged dataset size for workload comparison and genetic algorithms for candidate search. [17] integrated Bayesian Optimization with Gradient Descent and meta-learning for warm-starting surrogate models. OPPerTune [35] employed reinforcement learning with reward functions based on observed performance. Frameworks like FLAML [40, 41, 44] enable hyperparameter tuning with gradient search but face limitations in production due to their reliance on query execution and flighting. Our approach combines the strengths of Bayesian Optimization and gradual tuning by leveraging a surrogate model to guide exploration while using statistically derived gradients to ensure efficient convergence, even in the presence of noisy data and production constraints.

8 Conclusion

In this paper, we present Rockhopper, an end-to-end autotuning system integrated into the Spark platform on Microsoft Fabric. Rockhopper addresses key production challenges such as noisy data, dynamic workloads, and limited information at application startup. Leveraging the Centroid Learning algorithm, it minimizes performance regression, accelerates convergence, and provides a warm-start for customer workloads. Rockhopper also incorporates workload embeddings for transfer learning, enabling efficient tuning from benchmark to production workloads. Deployed in Fabric, Rockhopper achieves a significant 20% improvement in total execution time for Spark workloads, delivering these gains without requiring additional customer effort or compromising data privacy. Future work will focus on enhancing learning algorithms, refining workload embeddings, and developing adaptive strategies for dynamic workloads. Additionally, we aim to introduce more configurable parameters to further optimize performance. Rockhopper marks a major step forward in integrating autotuning into production Spark platforms, paving the way for more efficient and cost-effective big data processing in the cloud.

References

[1] Anonymous. 2024. AutoDebugger: Efficient Root Cause Analysis for Anomaly Jobs. (2024). Working paper.

[2] AWS. 2022. *Amazon.com, Inc.* Retrieved July 2, 2022 from https://aws.amazon.com/emr/features/spark/

[3] Zhendong Bei, Zhibin Yu, Huiling Zhang, Wen Xiong, Chengzhong Xu, Lieven Eeckhout, and Shengzhong Feng. 2015. RFHOC: A random-forest approach to auto-tuning hadoop's configuration. *IEEE Transactions on Parallel and Distributed Systems* 27, 5 (2015), 1470–1483.

[4] Christopher M Bishop and Nasser M Nasrabadi. 2006. *Pattern recognition and machine learning*. Vol. 4. Springer.

[5] Scikit-Optimize Contributors. 2023. *Scikit-Optimize.* Retrieved July 2, 2023 from https://scikit-optimize.github.io/stable/index.html

[6] Carlo Curino, Neha Godwal, Brian Kroth, Sergiy Kuryata, Greg Lapinski, Siqi Liu, Slava Oks, Olga Poppe, Adam Smiechowski, Ed Thayer, et al. 2020. MLOS: An infrastructure for automated software performance engineering. In *Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning*. 1–5.

[7] Ayat Fekry, Lucian Carata, Thomas F. J.-M. Pasquier, Andrew Rice, and Andy Hopper. 2020. Tuneful: An Online Significance-Aware Configuration Tuner for Big Data Analytics. *CoRR* abs/2001.08002 (2020). arXiv:2001.08002 https://arxiv.org/abs/2001.08002

[8] Johannes Freischuetz, Konstantinos Kanellis, Brian Kroth, and Shivaram Venkataraman. [n.d.]. Performance Roulette: How Cloud Weather Affects ML-Based System Optimization.

[9] Google. 2022. *Serverless Spark.* Retrieved July 2, 2022 from https://cloud.google.com/dataproc-serverless/docs

[10] Google. 2022. *Spark through Vertex AI.* Retrieved July 2, 2022 from https://cloud.google.com/vertex-ai-workbench

[11] Jing Gu, Ying Li, Hongyan Tang, and Zhonghai Wu. 2018. Auto-tuning spark configurations based on neural network. In *2018 IEEE International Conference on Communications (ICC)*. IEEE, 1–6.

[12] Xu Huang, Hong Zhang, and Xiaomeng Zhai. 2022. A Novel Reinforcement Learning Approach for Spark Configuration Parameter Optimization. *Sensors* 22, 15 (2022), 5930.

[13] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2011. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization: 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers 5*. Springer, 507–523.

[14] Alekh Jindal, Hiren Patel, Abhishek Roy, Shi Qiao, Zhicheng Yin, Rathijit Sen, and Subru Krishnan. 2019. Peregrine: Workload optimization for cloud query engines. In *Proceedings of the ACM Symposium on Cloud Computing*. 416–427.

[15] Konstantinos Kanellis, Cong Ding, Brian Kroth, Andreas Müller, Carlo Curino, and Shivaram Venkataraman. 2022. LlamaTune: sample-efficient DBMS configuration tuning. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2953–2965.

[16] Brian Kroth, Sergiy Matusevych, Rana Alotaibi, Yiwen Zhu, Anja Gruenheid, and Yuanyuan Tian. 2024. MLOS in Action: Bridging the Gap Between Experimentation and Auto-Tuning in the Cloud. *Proc. VLDB Endow.* 17, 12 (Nov. 2024), 4269–4272. https://doi.org/10.14778/3685800.3685852

[17] Yang Li, Huaijun Jiang, Yu Shen, Yide Fang, Xiaofeng Yang, Danqing Huang, Xinyi Zhang, Wentao Zhang, Ce Zhang, Peng Chen, et al. 2023. Towards General and Efficient Online Tuning for Spark. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3570–3583.

[18] Chen Lin, Junqing Zhuang, Jiadong Feng, Hui Li, Xuanhe Zhou, and Guoliang Li. 2022. Adaptive code learning for spark configuration tuning. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 1995–2007.

[19] Chenghao Lyu, Qi Fan, Philippe Guyard, and Yanlei Diao. 2024. A Spark Optimizer for Adaptive, Fine-Grained Parameter Tuning. *Proc. VLDB Endow.* 17, 11 (Aug. 2024), 3565–3579. https://doi.org/10.14778/3681954.3682021

[20] Microsoft. 2022. *Azure HDInsight.* Retrieved July 2, 2022 from https://docs.microsoft.com/en-us/azure/hdinsight/spark/apache-spark-overview

[21] Microsoft. 2022. *Azure Synapse.* Retrieved July 2, 2022 from https://docs.microsoft.com/en-us/azure/synapse-analytics/spark/apache-spark-overview

[22] Microsoft. 2023. *NimbusML.* Retrieved April 5, 2023 from https://learn.microsoft.com/en-us/nimbusml/overview

[23] Microsoft. 2024. *Configure Autotune for Fabric Spark.* Retrieved June 27, 2024 from https://learn.microsoft.com/en-us/fabric/data-engineering/autotune?tabs=sparksql

[24] Nhan Nguyen, Mohammad Maifi Hasan Khan, and Kewen Wang. 2018. Towards automatic tuning of apache spark configuration. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 417–425.

[25] ONNX.AI. 2023. *Open Neural Network Exchange (ONNX).* Retrieved April 5, 2023 from https://onnx.ai/

[26] Wiki Pedia. 2023. *Hill Climbing.* Retrieved Oct 18, 2023 from https://en.wikipedia.org/wiki/Hill_climbing

[27] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* 12 (Nov. 2011), 2825–2830. http://dl.acm.org/citation.cfm?id=1953048.2078195

[28] David Buchaca Prats, Felipe Albuquerque Portella, Carlos HA Costa, and Josep Lluis Berral. 2020. You only run once: spark auto-tuning from a single run. *IEEE Transactions on Network and Service Management* 17, 4 (2020), 2039–2051.

[29] Ning Qian. 1999. On the momentum term in gradient descent learning algorithms. *Neural networks* 12, 1 (1999), 145–151.

[30] Abhishek Roy, Alekh Jindal, Priyanka Gomatam, Xiating Ouyang, Ashit Gosalia, Nishkam Ravi, Swinky Mann, and Prakhar Jain. 2021. SparkCruise: workload optimization in managed spark clusters at Microsoft. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3122–3134.

[31] Scikit-learn. 2023. *Support Vector Machine.* Retrieved Nov 22, 2023 from https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html

[32] Rathijit Sen, Abhishek Roy, and Alekh Jindal. 2023. Predictive Price-Performance Optimization for Serverless Query Processing. In *Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023*. OpenProceedings.org, 118–130. https://doi.org/10.48786/EDBT.2023.10

[33] Rathijit Sen, Abhishek Roy, Alekh Jindal, Rui Fang, Jeff Zheng, Xiaolei Liu, and Ruiping Li. 2021. AutoExecutor: predictive parallelism for spark SQL queries. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2855–2858.

[34] Yu Shen, Xinyuyang Ren, Yupeng Lu, Huaijun Jiang, Huanyong Xu, Di Peng, Yang Li, Wentao Zhang, and Bin Cui. 2023. Rover: An online Spark SQL tuning service via generalized transfer learning. *arXiv preprint arXiv:2302.04046* (2023).

[35] Gagan Somashekar, Karan Tandon, Anush Kini, Chieh-Chun Chang, Petr Husak, Ranjita Bhagwan, Mayukh Das, Anshul Gandhi, and Nagarajan Natarajan. 2024. {OPPerTune}:{Post-Deployment} Configuration Tuning of Services Made Easy. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1101–1120.

[36] Fei Song, Khaled Zaouk, Chenghao Lyu, Arnab Sinha, Qi Fan, Yanlei Diao, and Prashant Shenoy. 2021. Spark-based cloud data analytics using multi-objective optimization. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 396–407.

[37] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. 2013. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*. PMLR, 1139–1147.

[38] European Union. 2022. *General Data Protection Regulation.* Retrieved Feb 23, 2022 from https://eur-lex.europa.eu/legal-content/EN/ALL/?uri=celex.

[39] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*. 1009–1024.

[40] Chi Wang, Qingyun Wu, Silu Huang, and Amin Saied. 2021. Economical Hyperparameter Optimization With Blended Search Strategy. In *ICLR*.

[41] Chi Wang, Qingyun Wu, Markus Weimer, and Erkang Zhu. 2021. FLAML: A Fast and Lightweight AutoML Library. In *MLSys*.

[42] 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.

[43] Lianggui Weng, Rong Zhu, Di Wu, Bolin Ding, Bolong Zheng, and Jingren Zhou. 2024. Eraser: Eliminating Performance Regression on Learned Query Optimizer. *Proc. VLDB Endow.* 17, 5 (May 2024), 926–938. https://doi.org/10.14778/3641204.3641205

[44] Qingyun Wu, Chi Wang, and Silu Huang. 2021. Frugal Optimization for Cost-related Hyperparameters. In *AAAI*.

[45] Yixin Wu, Xiuqi Huang, Zhongjia Wei, Hang Cheng, Chaohui Xin, Zuzhi Chen, Binbin Chen, Yufei Wu, Hao Wang, Tieying Zhang, et al. 2024. Towards Resource Efficiency: Practical Insights into Large-Scale Spark Workloads at ByteDance. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3759–3771.

[46] Jinhan Xin, Kai Hwang, and Zhibin Yu. 2022. LOCAT: Low-Overhead Online Configuration Auto-Tuning of Spark SQL Applications. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 674–684. https://doi.org/10.1145/3514221.3526157

[47] Zhibin Yu, Zhendong Bei, and Xuehai Qian. 2018. Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 564–577.

[48] Bohan Zhang, Dana Van Aken, Justin Wang, Tao Dai, Shuli Jiang, Jacky Lao, Siyuan Sheng, Andrew Pavlo, and Geoffrey J Gordon. 2018. A demonstration of the ottertune automatic database management system tuning service. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1910–1913.

[49] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. 2019. An end-to-end automatic

cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*. 415–432.

[50] William Zhang, Wan Shen Lim, Matthew Butrovich, and Andrew Pavlo. 2024. The Holon Approach for Simultaneously Tuning Multiple Components in a Self-Driving Database Management System with Machine Learning via Synthesized Proto-Actions. *Proc. VLDB Endow.* 17, 11 (July 2024), 3373–3387. https://doi.org/10.14778/3681954.3682007

[51] Xinyi Zhang, Zhuo Chang, Yang Li, Hong Wu, Jian Tan, Feifei Li, and Bin Cui. 2022. Facilitating database tuning with hyper-parameter optimization: a comprehensive experimental evaluation. *Proceedings of the VLDB Endowment* 15, 9 (2022), 1808–1821.

[52] Xinyi Zhang, Hong Wu, Yang Li, Jian Tan, Feifei Li, and Bin Cui. 2022. Towards dynamic and safe configuration tuning for cloud databases. In *Proceedings of the 2022 International Conference on Management of Data*. 631–645.

[53] Yiwen Zhu, Matteo Interlandi, Abhishek Roy, Krishnadhan Das, Hiren Patel, Malay Bag, Hitesh Sharma, and Alekh Jindal. 2021. Phoebe: a learning-based checkpoint optimizer. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2505–2518.

[54] Yiwen Zhu, Yuanyuan Tian, Joyce Cahoon, Subru Krishnan, Ankita Agarwal, Rana Alotaibi, Jesús Camacho-Rodriguez, Bibin Chundatt, Andrew Chung, Niharika Dutta, et al. 2023. Towards Building Autonomous Data Services on Azure. In *Companion of the 2023 International Conference on Management of Data*. 217–224.