Scarf: Self-Adaptive Tuning via Multi-Objective Reinforcement Learning for Apache Flink

Liu Liu
Zhejiang University
Hangzhou, Zhejiang
China
liu2@zju.edu.cn

Shenghao Gong
Zhejiang University
Hangzhou, Zhejiang
China
gongshenghao@zju.edu.cn

Ziquan Fang
Zhejiang University
Hangzhou, Zhejiang
China
zqfang@zju.edu.cn

Yunjun Gao
Zhejiang University
Hangzhou, Zhejiang
China
gaoyj@zju.edu.cn

ABSTRACT

Distributed stream processing systems (DSPSs) such as Apache Flink have become omnipresent for real-time data processing in e-commerce, finance, telecommunications, etc. The execution behavior of Flink is controlled by a vast and complex space of configuration knobs, necessitating automatic knob tuning to economize resource usage while maintaining sufficient processing capabilities for a given workload. Existing automatic methods largely adjust limited configuration knobs, respond slowly to dynamic workloads, and have difficulty transferring knowledge between heterogeneous jobs with diverse knob spaces.

To solve these problems, we present **Scarf**, a self-adaptive configuration tuning framework using multi-objective reinforcement learning (RL) for Apache Flink. Specifically, (1) we accelerate job-specific knob selection by clustering historical workloads according to their sensitivity to knob changes, dramatically reducing redundant sampling; (2) we formulate tuning as a multi-objective RL problem that jointly optimizes throughput and resource usage, learning a forest of RL models offline representing the Pareto front of the configurations, and dynamically selecting configurations from the Pareto front under fluctuating online workloads; (3) we enable rapid adaptation to new job topologies via a transferable actor–critic architecture based on graph neural networks (GNNs), complemented with a progressive neural-network (PNN) warm-up strategy. We implement **Scarf** on Apache Flink and evaluate it on a diverse range of streaming applications. Our framework significantly outperforms state-of-the-art DSPS tuning approaches, achieving up to 62.5% savings in CPU resources, 68.3% savings in memory usage, 77.1% reduction in online tuning time, while maintaining sufficient processing abilities throughout workload fluctuations.

PVLDB Reference Format:
Liu Liu, Shenghao Gong, Ziquan Fang, and Yunjun Gao. Scarf: Self-Adaptive Tuning via Multi-Objective Reinforcement Learning for Apache Flink. PVLDB, 19(7): 1516 - 1529, 2026.
doi:10.14778/3801059.3801066

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ZJU-DAILY/Scarf.

Proceedings of the VLDB Endowment, Vol. 19, No. 7 ISSN 2150-8097.
doi:10.14778/3801059.3801066

1 INTRODUCTION

With the explosive growth of online and real-time data, distributed stream processing systems (DSPSs) such as Apache Flink [2], Apache Storm [3], and Kafka Streams [5] have become widely adopted in domains including e-commerce [56], finance [40], and telecommunications [45]. These systems are designed to continuously process large volumes of data in real time, enabling timely data processing and analytics. Among existing DSPSs, Apache Flink has become the favored system for stream processing due to its high performance, exactly-once semantics, high-availability guarantees, and an extensive ecosystem [18, 19, 53].

Knob tuning is an essential step when deploying streaming applications on Flink, as knob values significantly impacts the runtime behavior of Flink, including throughput, latency, and resource consumption. For example, the WordCount workload [27] runs at 5.2 K queries per second (QPS) with 1 core and 1.7 GB memory under default settings, but reaches 90.7 K QPS with 32 cores and 63.1 GB memory after tuning. However, Flink's knob space is immense and complex. Flink exposes more than 200 knobs spanning operator-level settings (e.g., parallelism) and cluster-level parameters (e.g., network buffer size, checkpoint frequency, SQL optimizer flags) [11], making manual tuning difficult. Moreover, the interactions among knobs are often intricate. For instance, increasing operator parallelism may reduce latency, but it can lead deployment failures due to insufficient resources if the number of task slots is not increased accordingly. End users may lack the expertise or time to manually tune this large knob space. Consequently, there is a growing academic trend towards automated tuning for DSPSs [12, 23, 25, 31, 33, 35, 43, 44, 61, 77]. In the industry, vendors of Apache Flink (e.g., Alibaba Cloud [9], Ververica [8], Amazon [7]) now also offer out-of-the-box auto-tuning capabilities.

Existing approaches either narrow down the configuration scope by focusing solely on parallelism tuning [12, 33, 43, 44, 61, 77] or rely on manual, expert-driven knob selection [23, 25, 31, 35], which limits their effectiveness and generalizability. In contrast, we solve Flink tuning with intelligent **multi-objective optimization (MOO)**. This is essential due to the key difference between batch and stream processing. Batch systems receive finite data at once, so maximizing throughput is a dominant goal; whereas streaming systems ingest data incrementally, where provisioning resources more than needed to match the current input rate provides no performance benefit. Thus, the tuning goal must jointly (1) **minimize resource costs** (e.g., CPU, memory) and (2) **maintain required throughput and latency**. Moreover, workload intensity fluctuates over time, which shifts the trade-off between performance and cost. For example, Figure 1 (a) shows an example of workload trace [72],

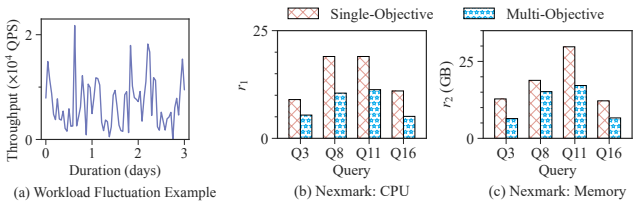

Figure 1: Research Motivation. (a) The per-hour throughput over three days for a real workload. (b, c) Average CPU (r_1) and memory (r_2) usage comparison between single-objective and multi-objective tuning under varying workloads.

which exhibits frequent fluctuations. In this setting, configurations optimized for peak load become wasteful under lighter load. Figure 1 (b,c) compare the performance of single-objective optimization (targeting throughput only) with dynamic MOO (balancing throughput and resource usage throughout the workload) under fluctuations. The results demonstrate that MOO enhances resource efficiency, reducing average CPU consumption by 40%–53% and memory usage by 20%–50%. Without MOO, existing method typically rely on two workarounds: (i) Retrain a model for each workload [27, 36, 77]. However, as workloads shift, previously trained models quickly become useless and must be retrained from scratch. (ii) Search the knob space using a strategy (e.g., heuristics or Bayesian optimization) guided by a performance model [33, 35, 43], yet it is challenging to accurately construct performance models across various workloads and the complex DSPS knob space. To this end, for the first time to our knowledge, we present a tuning framework for Apache Flink based on **multi-objective reinforcement learning (MORL)**. Specifically, by formulating Flink tuning as MOO, we can flexibly balance performance and resource consumption. Then, we employ a forest of RL agents to explore the vast knob space guided by multi-objective goals. This enables us to rapidly identify resource-efficient configurations across varying workloads. Nevertheless, several challenges must be addressed.

C1. Tedious job-specific knob selection. To reduce the search space, existing approaches require hand-picking a subset of parameters [23, 25, 31, 35], or focus on only tuning parallelism (i.e., *autoscaling*) [12, 33, 43, 44, 61, 77]. These methods may fail to identify knobs that are critical for a specific job. In contrast, to exploit Flink's richer configuration set, we perform automatic knob selection before tuning each job [37]. The selection process uses Latin hypercube sampling (LHS) [51] to generate knob samples, and then uses importance metrics such as SHAP [49] to identify the most influential knobs. However, because job characteristics vary widely, the importance of knobs also differs. For example, state backend settings are irrelevant for stateless jobs, and SQL optimizer flags do not affect jobs implemented directly with the DataStream API. This means that knob selection must be repeated for each individual job, requiring hundreds of samples each time. To address this challenge, we introduce **a knob selection acceleration method with workload clustering**. By clustering historical jobs based on their sensitivity to knob changes, we can identify a subset of knob vectors that best discriminates these clusters. As a new job arrives, we apply these knob vectors, assign the new job to the most similar cluster, and reuse the knobs selected for that cluster centroid. On top of that, we support dynamic fallback to full knob selection when no similar historical job exists. This approach significantly reduces redundant sampling and accelerates knob selection.

C2. Excessive reconfiguration under fluctuating load. Current methods [33, 36, 43] primarily rely on online training, which learns from feedback (e.g., job performance, resource usage, system metrics) based on actions taken in the production environment. Inevitably, these methods must perform exploratory actions. Each such exploration triggers a system reconfiguration, including job restart and state recovery, interrupting the data flow for approximately 1–2 minutes. Even worse, some explorations may yield poor knob values, causing severe performance degradation or even job failure, which is highly detrimental to production stability. To address this issue, we propose **a forest-based offline–online MORL approach for adaptive tuning under fluctuating load**. In the offline learning phase, we employ a forest of reinforcement learning (RL) models to identify a set of checkpoint-compatible Pareto fronts balancing throughput and resource usage. In the online phase, we directly select the knob values on the Pareto front that satisfy throughput requirements with minimal resource consumption. Compared with online training methods, our approach avoids excessive reconfigurations caused by disruptive explorations in the production environment.

C3. Cross-job knowledge transfer. Building a tuning model for a single job requires hundreds of samples. The samples are generated by job executions, which take hours to days in total. To speed up the tuning process, existing methods try to transfer the knowledge learned from previous tuning sessions [12, 33]. However, the heterogeneity of the jobs makes it difficult to transfer the knowledge learned from one job to another, because the different topology of job graphs will result in different action spaces and state spaces, impeding the transferability of RL models. To solve this problem, we present **a transferable GNN-based actor-critic network and a PNN-based warm-up strategy.** The actor-critic network uses state-of-the-art graph neural networks (GNNs) to model the execution graph of the streaming job. The operator-specific configurations are carefully handled at the node-level, enabling transferability across jobs with different topologies. Furthermore, we introduce a warm-up strategy based on Progressive Neural Networks (PNN) [62] to transfer the RL model forest and policy weights from similar jobs, thereby rapidly bootstrapping the tuning model for new tasks. This method significantly accelerates the tuning process for unseen jobs.

In summary, we introduce **Scarf**, a **s**elf-adaptive **c**onfigur**a**tion tuning framework via multi-objective **r**einforcement learning for Apache **F**link. The main contributions of this work are:

- We present, to our best knowledge, the first self-adaptive Flink tuning method using multi-objective RL (Sec. 3).
- We develop a workload clustering method based on sensitivity to knob changes that significantly reduces redundant sampling in the knob selection process (Sec. 4).
- We propose a forest-based RL model to adapt to the fluctuating load (Sec. 5).
- We design a transferable GNN-based policy architecture together with a PNN-based cross-job transfer mechanism,

enabling transferability and fast adaptation between heterogeneous job graphs (Sec. 6).
- We implement and comprehensively evaluate **Scarf** on Apache Flink, demonstrating up to 77.1% reduction in tuning time and up to 62.5% and 68.3% savings for CPU and memory resources compared to prior methods (Sec. 8).

The remainder of the paper is organized as follows. Section 2 introduces the preliminaries and formalizes the problem. Section 3 provides an overview of our solution. Section 4 details the workload clustering. Section 5 presents the MORL process. Section 6 describes the transferable GNN-based actor–critic network. Section 7 discusses how our tuner impacts critical DSPS functionality. Section 8 outlines the experimental setup and results. Section 9 compares related work. Finally, Section 10 concludes the paper.

2 PROBLEM STATEMENTS

Apache Flink utilizes the Dataflow model [13] to handle boundless streaming data. In this section, we first introduce the key concepts of Apache Flink, then formally define the knob tuning problem.

2.1 Key Concepts of Apache Flink

Streaming Job Representation. In Flink, a **logical job graph** defines the streaming job as a Directed Acyclic Graph (DAG) $G = (V, E)$, where V represents operators and E represents data flow. Data enters through source operators from external services (e.g., message queues), flows downstream for per-tuple computing, and exits via sink operators into external systems (e.g., OLAP warehouses). In Flink, the job graph can be defined using the DataStream API or generated by the Flink SQL optimizer.

Upon deployment, the job graph is transformed into a physical **execution graph**, which specifies the parallelism (i.e., number of instances) per vertex. During execution, these operator instances are scheduled on task managers, processing data partitions in parallel. Parallelism can be set globally or for each operator separately.

Configurations and Metrics. Flink offers numerous **configuration knobs** that impact performance and resource utilization. The knob set C can be divided into per-operator parallelisms C_o and global knobs C_g. Per-operator parallelisms enables job elasticity, while global knobs affect various system behaviors, including memory allocation, fault tolerance, state access, job scheduling and execution, and SQL optimization.

Common **performance metrics** for a streaming job include **throughput** (records processed per second) and **latency** (time for a record to travel from source to sink). Following previous works, we choose throughput as the primary performance metric, because latency is acceptably compromised in the distributed scenario [16].

Reconfiguration Overhead. Streaming jobs are typically long-running and stateful. Applying new configurations requires triggering a **savepoint** to persist operator states externally, restarting the cluster with new settings, and reloading the states from the savepoint. This process will halt data consumption for several minutes, causing degraded performance and increased latency. Thus, reconfigurations should be minimized in production.

An critical issue for reconfiguration is **savepoint compatibility**. During savepoint recovery, Flink maps state to operators using IDs derived from the job graph structure. If configuration changes (e.g., SQL optimizer flags, execution strategies) alter the resulting job graph, operator IDs may change, preventing state restoration. Therefore, online tuning must ensure different configurations generate identical job graphs to avoid state loss. To our knowledge, this issue has not yet been addressed in existing work.

2.2 The Auto-Tuning Problem of Apache Flink

With the core concepts introduced, we now present the formal definition of the auto-tuning problem of Apache Flink.

Definition 2.1 (Auto-Tuning Problem of Apache Flink). Given a streaming job J and a set of configuration knobs $C = C_o \cup C_g$, where C_o denotes the operator parallelism settings and C_g denotes the global configurations, let the throughput be $T(J, \lambda, C)$ and the total acquired resources be $R(J, C) = R_1(J, C) + \omega R_2(J, C)$. Here, λ represents the source rate of the entire job, R_1 denotes the number of acquired CPU cores, R_2 denotes the acquired memory space, and ω is a weighting factor chosen to normalize the two components to a comparable scale. The auto-tuning problem is to identify the optimal configuration:

$$c^* = \arg\min_{c \in C} R(J, c), \quad \text{s.t. } T(J, \lambda, c) \geq \lambda$$

In other words, the goal is to determine a configuration c^* that minimizes the total acquired resources while guaranteeing that the throughput of the job is no less than the source rate λ.

3 FRAMEWORK OVERVIEW

In this section, we provide an overview of **Scarf**, our self-adaptive configuration tuner for Flink. Figure 2 shows the architecture of the tuner comprising three key components:

① **Knob selection speedup via workload clustering**. This component addresses the challenge of vast knob spaces by knob selection, and accelerates the selection process when historical data is available. In cold-start scenarios, we sample N_c knob vectors from the search space to be used across all jobs. Then, we utilize SHAP values to quantify knob importance, and select the top K knobs with the highest importance. In warm-start scenarios, we construct performance vectors from previous knob selection sessions to cluster historical jobs with similar sensitivities to knob changes. Next, we identify a subset of knob vectors that best discriminate these clusters. New jobs are evaluated using only this subset, mapped to the closest historical cluster, and reuse the selected knobs from its centroid job. This significantly reduces redundant sampling during knob selection and ensures that the knob subspaces are consistent for similar jobs, facilitating knowledge transfer.

② **Multi-objective RL for adaptive tuning**. The MORL module focuses on the multi-objective nature of the tuning problem, while minimizing online tuning overhead. In the offline phase, we initialize a forest of multi-objective RL policies trained with varying weights for performance and resource usage. Next, we iteratively evolve these policies to construct a Pareto front, representing the best trade-offs between throughput and resource consumption. In the online phase, we generate configurations from these policies and map them to the corresponding job graph topologies. We then select the optimal Pareto front based on hypervolume and sparsity

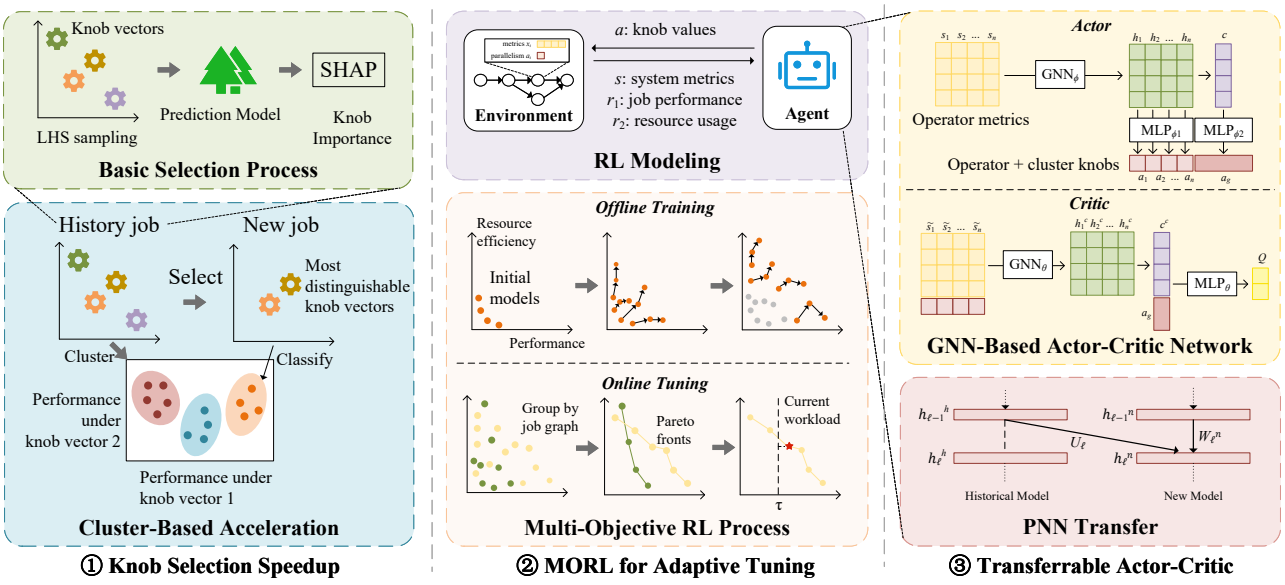

Figure 2: An overview of SCARF: a self-adaptive configuration tuning framework for Apache Flink.

metrics. This yields savepoint-compatible configurations with better performance and resource efficiency. During the online phase, we monitor the workload intensity and system behavior, dynamically selecting the configuration that meets the throughput constraints with the lowest resource cost. This design ensures seamless adaptation to workload fluctuations with minimal reconfigurations.

③ **Transferable GNN-based actor-critic networks.** The final component enables knowledge transfer across heterogeneous jobs to speed up policy training. First, to address the variable state and action dimensionality caused by heterogeneous job graphs, we employ Graph Neural Networks (GNNs) in the actor and critic networks of the RL policies, and handle operator-level actions and states at the node level. The resulting network is transferable across jobs with different topologies. On top of that, we incorporate a Progressive Neural Network (PNN) based warm-up strategy that boostraps the new job's Pareto optimal policies. This significantly accelerates the tuning process for unseen jobs.

It is worth noting that **SCARF** is not tied to Flink and can be applied to other stream processing framework that uses DAG-based job graphs, such as Apache Storm and Spark Structured Streaming. Its learning mechanisms rely only on standard concepts in stream processing, including throughput, latency, resource usage, and common system metrics. Therefore, **SCARF** can be easily adapted to other systems by modifying the interface that interacts with the underlying system to collect metrics and apply configurations.

4 KNOB SELECTION SPEEDUP

Motivation. Existing knob selection methods are job-specific, which leads to redundant measurements for each job. This motivates a workload-aware knob selection strategy, where we utilize parameter-response relationships to cluster historical jobs, identify the most similar job for a new job, and reuse the high-impact knob subsets. In this way, we can dramatically reduce performance samplings of real workloads, speeding up knob selection.

4.1 Basic Process

We first introduce the basic process of knob selection consisting of two main steps: LHS sampling and importance estimation.

LHS sampling. The first step is to sample the whole configuration space. The naive random sampling method often creates non-uniform samples, leading to suboptimal results [26]. Therefore, we adopt the popular Latin Hypercube Sampling (LHS) [51] method for sample generation. Assuming we need to generate N_c samples, LHS divides each dimension of the configuration space into N_c equal intervals, and then generates a random permutation of the intervals. Each sample is then generated by randomly selecting one value from each interval in the order of the joined permutations. This ensures that the samples are uniformly distributed.

Importance estimation. After sampling the knob space using LHS, we need to evaluate the importance of each knob to job performance. Common approaches for estimating knob importance include Lasso [65], Gini score [55], and the SHapley Additive exPlanations (SHAP) values [49]. We choose SHAP values as the performance estimator, which is based on cooperative game theory and explains how much each knob contributes to the overall performance. SHAP can capture negative contributions, enabling us to avoid tuning knobs whose changes would degrade performance. In contrast, Lasso and Gini scores are variance-based and therefore cannot distinguish between positive and negative contributions [78]. Therefore, SHAP is theoretically more effective for identifying the knobs most worth tuning.

4.2 Cluster-based Speedup

The basic knob selection process described above is time-consuming, as it needs to collect hundreds of samples for each job to train a

Algorithm 1 Finding the Most Distinguishing Knob Vectors

Input: Historical jobs $F = \{f_1, \ldots, f_n\}$, sample size m, number of clusters c, distance threshold δ, selected sample size m'
Output: Clusters $C = \{C_1, \ldots, C_c\}$, centroids $\{\mu_i\}_{i=1}^{c}$, selected knob vector indices Π

1: $X \leftarrow \text{LHS}(m)$ ▷ Sample knob vectors $X = \{x_1, \ldots, x_m\}$
2: Compute $P \in \mathbb{R}^{n \times m}$ by $P_{i,\cdot} \leftarrow f_i(X)$
3: $C, \boldsymbol{\mu} \leftarrow \text{KMeans}\big(\{P_{i,\cdot}\}_{i=1}^{n},\ c\big)$
4: **for** $j = 1, \ldots, m$ **do**
5: $\quad s_j \leftarrow \text{Var}\big(\{\mu_{i,j}\}_{i=1}^{c}\big)$
6: **end for**
7: Let π be a permutation of $1, \ldots, m$ s.t. $s_{\pi_1} \geq \cdots \geq s_{\pi_m}$
8: $\Pi \leftarrow \emptyset$
9: **for** $k = 1, \ldots, m$ **do**
10: $\quad$ **if** $\forall \pi \in \Pi,\ \text{dist}(x_{\pi_k}, x_\pi) > \delta$ **then**
11: $\quad\quad \Pi \leftarrow \Pi \cup \{\pi_k\}$
12: $\quad$ **end if**
13: $\quad$ **if** $|\Pi| = m'$ **then**
14: $\quad\quad$ Break
15: $\quad$ **end if**
16: **end for**
17: **return** $C, \{\mu_i\}_{i=1}^{c}, \Pi$

Algorithm 2 Knob Importance Estimation for New Jobs

Input: New job f', historical jobs $F = \{f_1, \ldots, f_n\}$, knob vectors $X = \{x_1, \ldots, x_m\}$, selected indices $\Pi = \{\pi_1, \ldots, \pi_{m'}\}$, clusters and centroids $C = \{C_i, \mu_i\}_{i=1}^{c}$, similarity threshold δ
Output: Weights $\{w_i\}_{i=1}^{n}$ or fallback flag

1: Compute $\mathbf{p}' \in \mathbb{R}^{m'}$ by $\mathbf{p}' \leftarrow f'(X_\Pi)$
2: **for** $i = 1, \ldots, c$ **do**
3: $\quad d_i \leftarrow 1 - \dfrac{\mathbf{p}' \cdot (\mu_i)_\Pi}{\|\mathbf{p}'\|\ \|(\mu_i)_\Pi\|}$ ▷ Cosine distance
4: **end for**
5: i $\leftarrow \arg\max_{i \in I}\{d_i\}$ ▷ Set of indices $I = \{1, \ldots, c\}$
6: **if** $d_i \leq \delta$ **then**
7: $\quad$ **return** knob importance of the centroid job of cluster C_i
8: **else** ▷ No cluster is similar enough
9: $\quad$ **return** fallback
10: **end if**

performance model of relatively high accuracy. To speed up the process, we propose a cluster-based knob selection method that clusters historical jobs based on their performance patterns, selects the knob values that best distinguish these clusters, and assigns new jobs to the corresponding clusters using the selected knob values. The process is described in Algorithms 1 and 2.

Algorithm 1 outlines the procedure for finding the most distinguishing knob vectors from historical jobs. In line 1, we generate *m knob vectors* $X = \{x_1, \ldots, x_m\}$ using LHS sampling. A knob vector is an element in the configuration space consisting of operator parallelisms and cluster knobs. In line 2, for each historical job f_i, we retrieve its performance on the sampled knob vectors X from tuning logs, obtaining a performance vector $P_{i,\cdot}$. We use throughput as the performance metric, as stated in Section 2.2. In line 3, we cluster the historical jobs using these performance vectors with K-means, producing c clusters $C = \{C_1, \ldots, C_c\}$ and their corresponding centroids $\{\mu_i\}_{i=1}^{c}$.

In lines 4–5, for each knob vector index j, we compute a score s_j defined by the variance of the jth coordinate of all centroids, measuring how differently the clusters behave under knob vector x_j. In line 7, we then sort the knob vectors in descending order of the scores. Next, we iterate over this order and add each knob vector to the selected set Π if its distance from every knob vector already in the set is sufficiently large (lines 10–12). This enforces diversity among the selected knobs. The process stops once m' knob vectors have been selected (lines 13–15).

Algorithm 2 describes how to rank knobs by importance and select knobs for new jobs. In line 1, we evaluate only the m' selected knob vectors instead of the full set X. Since each evaluation takes 2–3 minutes, this substantially reduces the knob selection time. In lines 2–4, we compute the cosine distance between the new job's performance vector and the cluster centroids. In line 5, We identify the cluster whose centroid is most similar to the new job. If the distance is sufficiently small, we return the knob importance of the most similar centroid (line 7); otherwise, we return a fallback flag indicating that a full knob selection is required (line 9).

5 MORL FOR ADAPTIVE TUNING

Motivation. Single-objective tuning approaches focus solely on performance, which struggles to balance the tradeoff between throughput and resource consumption. Moreover, a single-objective model cannot effectively adapt to changing objective weights due to fluctuating workloads. Furthermore, the focus of online tuning leads to excessive online reconfigurations, halting data consumption. To this end, we model DSPS tuning as an MORL problem that explicitly balances throughput and resource efficiency, with a focus on offline training to enable fast, accurate online knob adjustments.

5.1 RL Modeling

We formally define the tuning process as a Markov Decision Process (MDP). An MDP is defined by a tuple (S, A, P, R), where S is the state space, A is the action space, $P(s'|s, a)$ is the state transition probability from s to s' given action a, $R(a, s)$ is the reward function for state s and action a.

5.1.1 State Space. We collect operator-level metrics as the state of the MDP. The Flink metric system [11] routinely collects various metrics at the operator level, which can be used to monitor the load and performance of each operator. We select the following metrics, which describe the operator's response to the current workload:

- *idleTimeMsPerSecond*: The time in milliseconds that the operator is not processing any records nor backpressured.
- *busyTimeMsPerSecond*: The time in milliseconds that the operator is processing records.
- *backpressureTimeMsPerSecond*: The time in milliseconds that the operator is blocked due to backpressure from downstream operators.
- *numRecordsInPerSecond*: The number of records processed by the operator per second.
- *numRecordsOutPerSecond*: The number of records emitted by the operator per second.

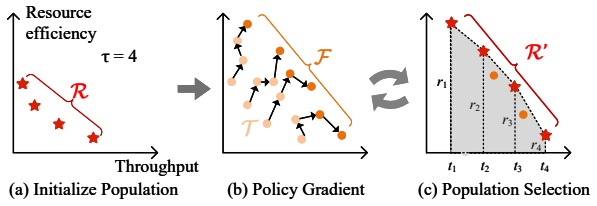

Figure 3: The Offline Training Process.

For each operator o_i, we collect the max and average values of these metrics across all parallel instances, and concatenate them to form the state vector $S = \{S_1, \ldots, S_N\}$, where N is the number of operators in the job. Therefore, the total state dimension is $10N$. The state space is agnostic to specific job topologies and operator types, facilitating transferability between different jobs.

5.1.2 Action Space. The action space is $A = A_o \times A_g$, where $A_o \subset \mathbb{R}^N$ is the space of operator parallelisms and $A_g \subset \mathbb{R}^{m'}$ is the space of selected global knobs.

5.1.3 Reward Function. The reward function is a 2-dimensional vector $\mathbf{r} = \{t, r\}$ consisting of job performance (i.e. throughput) and resource efficiency (i.e. CPU and memory). The throughput t is calculated as the sum of the *numRecordsInPerSecond* of all source operators. The resource efficiency r is defined as the inverse of $r_1 + \beta r_2$, where r_1 denotes the total number of CPU cores, and r_2 denotes the memory usage (in GB) of all operators. The parameter β is set to the average value of r_1/r_2 during knob selection, in order to keep the two components on a comparable scale.

Previous work usually transforms the reward vector into a scalar value by using a weighted sum of r and t. This means that the resulting model will only be optimized for a specific trade-off between throughput and resource usage. When the workload change requires a different trade-off, the model will become suboptimal and must be retrained. Consequently, we keep the reward vector as it is, allowing the agent to learn a policy that adapts to different trade-offs easily with MORL, as described in Section 5.2.

5.1.4 Optimization Objectives. We use the state-of-the-art framework, Soft-Actor-Critic (SAC) [32] to solve the MDP. SAC is an off-policy algorithm that maximizes expected reward while also maximizing entropy, which encourages exploration. Here, we change the optimization objective into a 2-dimensional vector $\mathbf{J}(\pi)$, where

$$J_i(\pi) = \sum_{t=0}^{T} \mathbb{E}\left[\mathbf{r}_i(s_t, a_t) + \alpha \mathcal{H}(\pi(\cdot|s_t))\right], i \in \{0, 1\}, \quad (1)$$

where π is the policy, $\mathcal{H}(\pi(\cdot|s_t))$ is the entropy of the π at state s_t, and α is a temperature parameter that controls the trade-off between exploration and exploitation.

5.2 The MORL Tuning Process

The MORL tuning process consists of two main stages: offline training and online tuning.

5.2.1 Offline Training. The offline training process is described in Figure 3. The main idea of the offline MORL training process is to maintain a population of τ policies, each of which is trained to optimize a different weighted sum of the reward function $\mathbf{w}_i \cdot \mathbf{r}$. Then, for each iteration, we select the next generation of policies and weights from the Pareto front of historical policies with the best quality. We first define the Pareto front of a set of history records $\mathcal{T} = \{(t_i, r_i)\}_{i=1}^n$, where t_i and r_i are the two components of the return indicating throughput and resource efficiency, respectively.

Algorithm 3 MORL: Offline Training Process

Input: Number of policies η, number of generations M, episode length N
Output: A set of policies Π

```
1:  R ← {π_i, ω_i}_{i=1}^η                          ▷ Initialize population
2:  T ← ∅
3:  for generation g = 1, ..., M do
4:      for task (π_i, ω_i) in R do                  ▷ Policy Gradient
5:          Run π_i for N steps
6:          π'_i ← perform policy gradient for task (π_i, ω_i)
7:          T = T ∪ {π'_i, (t_i, r_i)}
8:      end for
9:      ▷ Population selection
10:     R ← solve the dynamic programming transition in Eq. (5)
11: end for
12: return π_i for each (π_i, (t_i, r_i)) ∈ T
```

Definition 5.1 (Pareto Front). The Pareto front $\mathcal{F}$ of a set of records $\mathcal{T} = \{(t_i, r_i)\}_{i=1}^n$ is defined as the set of records that are not dominated by any other record in $\mathcal{T}$, i.e. there is no record (t_j, r_j) such that $t_j \geq t_i$ and $r_j \geq r_i$ for all $i \neq j$.

The detailed process is described in Algorithm 3.

Initializing population. In line 1, we initialize η RL policies π_i with corresponding objective weights ω_i. The policies are initialized randomly, while the objective weights are chosen uniformly.

Policy gradient. Then, for each generation, we collect M steps of experience for each policy (line 5) and update the policy using the collected experience (line 6). We store the updated policies and their return in a global history pool $\mathcal{T}$ (line 7).

Population selection. After that, we select η policies from the Pareto front $\mathcal{T}$ of the global history pool $\mathcal{T}$ with maximal quality (line 10). We define quality as a weighted sum of *hypervolume* and *sparsity*. Without loss of generality, we sort all candidates in $\mathcal{F}$ by throughput t in non-decreasing order. For the sorted subset $\mathcal{R} = \{(t_1, r_1), \ldots, (t_n, r_n)\} \subset \mathcal{F}$, we define:

Definition 5.2 (Hypervolume). Given the ordered set $\mathcal{R}$, the hypervolume is

$$\mathcal{H}(\mathcal{R}) = \sum_{i=1}^{n-1} (t_{i+1} - t_i)(r_i + r_{i+1}), \quad (2)$$

The hypervolume represents the area under the Pareto front. Larger hypervolume indicates that the policy can achieve better resource efficiency under various workload levels.

Definition 5.3 (Sparsity). Given the ordered set $\mathcal{R}$, the sparsity is

$$\mathcal{S}(\mathcal{R}) = \sum_{i=1}^{n-1} (t_{i+1} - t_i)^2, \quad (3)$$

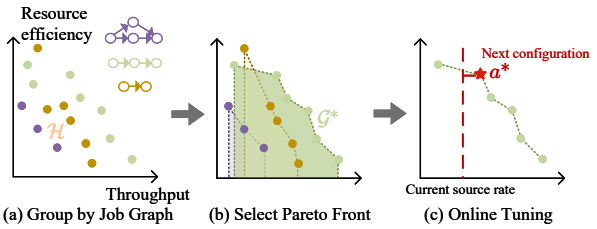

Figure 4: Online Policy Selection and Tuning.

Smaller sparsity means that policies are more evenly distributed in the throughput space, so that the agent can choose more suitable policies for specific throughput requirements.

The overall quality is defined as

$$Q(\mathcal{R}) = \mathcal{H}(\mathcal{R}) - \mathcal{S}(\mathcal{R}). \tag{4}$$

In order to select η points with the highest quality, we take advantage of the fact that both $\mathcal{H}$ and $\mathcal{S}$ involve only consecutive pairs in the sorted set $\mathcal{R}$. Therefore, we can solve the problem by dynamic programming. Let $F(i,k)$ be the maximum quality of any k-point subset ending at index i of $\mathcal{F}$, initialized with $F(i,1) = 0$. The transition for $k \geq 2$ is

$$F(i,k) = \max_{j<i} \left\{ F(j,k-1) + (t_i - t_j)(r_j + r_i) - (t_i - t_j)^2 \right\} \tag{5}$$

The optimal η-policy subset's quality is $\max_i F(i,\eta)$. The dynamic program is guaranteed to find the optimal subset in $O(|\mathcal{F}|^2\eta)$ time, making the total offline training time $O(M|\mathcal{F}|^2\eta)$.

After selection, we update the population with the selected policies. The weights w_i are the inverse of their returns, because they normalize the return values and encourage the model to balance the two objectives. The process is repeated for M generations until the population converges. All the policies and their corresponding returns are passed to the **online tuning** module.

5.2.2 Online Tuning. The online tuning process described in Figure 4, which is responsible for selecting a group of online policies and then adaptively choosing the best policy for the current workload. The process is described in Algorithm 4.

In lines 4-9, we obtain a set of optimal configurations by running each policy returned from the offline process. Theoretically, the Pareto front of these configurations represents the optimal settings for the current job. However, as discussed in Section 2.1, in order to preserve savepoint compatibility, the configurations used in the production environment must be restricted to those that generate an identical job graph. Thus, in line 7, we record the job graph associated with each configuration. Then, in line 11, for each job graph, we compute the Pareto front of its configurations using their multi-objective rewards and select the one with the highest quality according to Equation (4).

Once the system starts running, we continuously monitor the job's workload and the operators' performance. If the workload changes (line 14), we select the configuration with the lowest resource consumption (r_2) whose throughput (r_1) is sufficient to handle the current input rate (λ) (line 15).

Algorithm 4 MORL: Online Policy Selection and Tuning

Input: Offline policies $\Pi = \{\pi_1, \ldots, \pi_\eta\}$, episode length N
1: $\mathcal{H} \leftarrow \emptyset$ ▷ Optimal configuration set
2: $\mathcal{G} \leftarrow \emptyset$ ▷ Job graph set
3: ▷ Collect optimal configurations grouped by job graph
4: **for** each $\pi \in \Pi$ **do**
5: Run N steps and update policy for task (π_i, ω_i)
6: $a, \mathbf{r}, G \leftarrow$ the action, reward and corresponding job graph of the last step
7: $\mathcal{H} \leftarrow \mathcal{H} \cup \{(a, \mathbf{r}, G)\}$
8: $\mathcal{G} \leftarrow \mathcal{G} \cup \{G\}$
9: **end for**
10: ▷ Select the best job graph
11: $G^* \leftarrow \arg\max_{G \in \mathcal{G}} Q(\text{Pareto}(\{\mathbf{r} \mid (a, \mathbf{r}, G) \in \mathcal{H}\}))$
12: $\mathcal{P} \leftarrow \{(a, \mathbf{r}) \mid (a, \mathbf{r}, G^*) \in \mathcal{H}\}$
13: ▷ Online tuning
14: **for** each workload change **do**
15: $(a^*, \mathbf{r}^*) = \arg\min_{(a,\mathbf{r}) \in \mathcal{P}} r_2$ s.t. $r_1 \geq \lambda$
16: **end for**

6 TRANSFERABLE ACTOR-CRITIC

Motivation. Existing tuning methods cannot transfer knowledge between heterogeneous job topologies due to incompatible state spaces (operator metrics) and action spaces (operator-level parallelisms and global knobs), forcing expensive cold-start for every new job. To mitigate this, we design both actor and critic as GNNs that handle operator states and actions at the node level, and handle global state and action at the graph level. This architecture naturally handles variable-size DAGs and enables transferability across heterogeneous jobs. Next, we describe the transferable GNN-based actor-critic network, and provide a method for knowledge transfer between heterogeneous Flink jobs.

6.1 GNN-based Actor-Critic Network

Actor Network. Let $G = (V, E)$ be the job DAG with node features x_i. We run a GNN over $\{x_i\}$ to obtain node embeddings h_i:

$$\{h_i\}_{i \in V} = \text{GNN}_\phi(\{x_i\}). \tag{6}$$

Operator-level actions (i.e. operator parallelisms) are computed directly from node embeddings:

$$a_i = \text{MLP}_{\phi_o}(h_i), \quad \forall i \in V. \tag{7}$$

Global actions (i.e. cluster knobs) are computed by pooling the node embeddings and concatenating with a global MLP:

$$c = \frac{1}{N} \sum_{i \in V} h_i, \qquad a_g = \text{MLP}_{\phi_g}(c), \qquad \text{where } N = |V|. \tag{8}$$

Critic Network. Given a candidate action $(\{a_i\}, a_g)$, we first augment each node feature with its parallelism a_i:

$$\tilde{s}_i = [s_i \,\|\, a_i]. \tag{9}$$

We run a separate GNN to obtain critic node embeddings h_i^c:

$$\{h_i^c\}_{i \in V} = \text{GNN}_\theta(\{\tilde{s}_i\}). \tag{10}$$

We pool and concatenate with the global knobs a_g, then apply an MLP to get the joint Q-value:

$$c^c = \frac{1}{N}\sum_{i\in V} h_i^c, \qquad z = \left[c^c \,\|\, a_g\right], \qquad Q_\theta(s,a) = \mathrm{MLP}_{\theta_o}(z). \quad (11)$$

6.2 PNN-based Knowledge Transfer

In previous sections, we have made the following efforts to make knowledge transfer possible: **(i) Feature universality**: In Section 5.1.1, we incorporate universal and topology-agonistic RL states. With normalization, these states are consistent across jobs. **(ii) Structural transferability**: In Section 6.1, we design GNN-based actor-critic networks, which are permutation equivariant at the node level and invariant after graph pooling, naturally handling variable-size DAGs. In this section, we adopt Progressive Neural Network (PNN)-based knowledge transfer that reuses the most relevant historical policies and uses lateral connections to avoid catastrophic forgetting. In this way, we quickly obtain a Pareto front of configurations for the new job, reducing offline tuning cost.

Choosing the Historical Model. In Algorithm 1, each new job f' is assigned to a cluster, whose centroid job is f_c. By definition of the knob selection process, we will select the same knobs for tuning in the new job as f_c, while f_c has the most similar sensitivity to configurations as f'. Therefore, we can reuse the historical model trained on f_c to warm-up the new job f'.

PNN-style Layerwise Connections. For both the actor and critic networks, denote the output of the $(\ell-1)$-th layer in the *historical* model by $h^h_{\ell-1}$, and the corresponding output in the *new* model $h^n_{\ell-1}$. We define the ℓ-th layer of the new connected network as

$$h^n_\ell = \sigma\big(W^n_\ell\, h^n_{\ell-1} + U_\ell\, h^h_{\ell-1}\big), \quad \ell = 1,\dots,L, \qquad (12)$$

where W^n_ℓ are the trainable weights of the new model, U_ℓ is a small trainable matrix projecting the representation $h^h_{\ell-1}$ into the new feature space, and $\sigma(\cdot)$ is ReLU activation.

Analogous lateral connections are added to each layer of the critic network. To avoid catastrophic forgetting, all parameters of the historical model (ϕ^h, θ^h) remain frozen, and only the new parameters $\{W^n_\ell, U_\ell\}_{\ell=1}^L$ are updated during the warm-up phase.

Warm-up Training and Initial Pareto Front. We collect a small number of episodes N_{warm} on the new job f' by executing the combined actor–critic. Since the lateral connections already encode knowledge of good trade-offs from f_c, the new model converges quicker to a high-quality Pareto front. Finally, we perform online adaptation as described in Section 5.2.2.

7 IMPACT ON SYSTEM FUNCTIONALITY

Scarf addresses the following critical challenges in DSPSs:

❶ **Dynamic resource provisioning.** DSPSs dynamically allocate resources in response to workload fluctuations. **Scarf** uses MOO to learn a Pareto front of configurations balancing throughput and resource usage, and selects configurations based on workload changes to quickly adapt to dynamic resource demands. ❷ **Load balancing.** Data skew often causes resource inefficiency in DSPSs. **Scarf** does not directly modify data partitioning, but it can mitigate resource wastage by optimizing resource allocation and enabling query optimization flags (e.g., two-phase aggregation and skew join) to better handle skewed data. ❸ **Fault tolerance.** Flink achieves fault tolerance by persisting operator states through savepoints. However, existing tuning methods overlook savepoint incompatibility caused by heterogeneous job graphs. **Scarf** restricts online configurations to those that generate identical job graphs, ensuring correct state recovery during reconfiguration. ❹ **State management.** Efficient state management is crucial for low latency in stateful jobs. By optimizing state-backend-related knobs (e.g., memory allocation for RocksDB), **Scarf** improves state access times and reduces processing latency.

These challenges of DSPSs are highly interdependent. For example, under *dynamic resource provisioning*, it is common to increase operator parallelisms to handle higher loads, but data skew often becomes more prominent at high parallelisms, causing *load balancing* issues. Similarly, savepoint creation for *fault tolerance* consumes I/O bandwidth, contending resources with *state management*. Therefore, tuning individual components can result in local minima. In contrast, **Scarf**'s **end-to-end approach** optimizes global objectives to effectively manage these trade-offs.

8 EXPERIMENTS

8.1 Experimental Setup

We conduct experiments to answer the following questions:
Q1. How well does **Scarf** conserve CPU and memory resources?
Q2. Can **Scarf** allocate enough resources for job performance?
Q3. What is the overhead of **Scarf** compared to other baselines?
Q4. How does each component contribute to **Scarf**'s effectiveness?
Q5. How do parameters affect the tuning performance of **Scarf**?

8.1.1 Hardware and Software. The experiments are conducted on a cluster of 10 physical machines, each equipped with two 12-core CPUs (Intel Xeon E5-2620 v3, 2.40 GHz), 128 GB memory, and 1 TB HDD. All machines run CentOS 7.9 and are connected via 10 Gbps Ethernet. We deploy a YARN cluster with Hadoop 3.4.1, and run Apache Flink 2.0.0 in YARN application mode.

8.1.2 Workload Generation. To emulate the random fluctuations and periodicity of real-world streaming workloads, we vary the data input rate using a random permutation of predefined source rates. For each query, we determine a source rate unit W_u based on the query's complexity and generate the workload using a sequence of ten rate levels, e.g., [$9W_u$, $2W_u$, $3W_u$, $10W_u$, $1W_u$, $4W_u$, $5W_u$, $8W_u$, $6W_u$, $7W_u$]. We replicate this sequence to simulate periodicity. To eliminate possible bias caused by specific fluctuation patterns, we sample six random rate permutations for each query, resulting in 120 workload changes per query.

8.1.3 Queries. We evaluate the tuning methods using 32 queries from 3 benchmarks: Nexmark [6], PQP queries [12], and HiBench-stream [4]. Table 1 shows the source rate units (W_u) of each query.

- **Nexmark** [6]: a widely used streaming benchmark suite simulating an auction system. We use all 22 Nexmark queries, excluding Q6, which is not supported by Flink.
- **PQP queries** [12]: synthetic queries covering diverse scenarios, including linear queries, joins, and filters. We use the following query templates: linear (L), 2-way-join (2J),

Table 1: Evaluated Benchmarks and Queries

	Nexmark																						PQP Queries						HiBench			
Query	Q0	Q1	Q2	Q3	Q4	Q5	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22	L	2J	3J	2F	3F	4F	FW	ID	RP	WC
W_u	2M	2M	1M	1M	100K	1M	100K	1M	80K	500K	150K	500K	2M	2M	150K	20K	300K	200K	200K	100K	1M	1.5M	40K	25K	20K	40K	40K	30K	150K	3.5M	200K	200K

3-way-join (3J), 2-filter-chained (2F), 3-filter-chained (3F), and 4-filter-chained (4F).

- **HiBench-stream** [4]: a big-data benchmark suite with various workloads. We use all four streaming workloads provided in HiBench-stream: Fixwindow (FW), Identity (ID), Repartition (RP), and Stateful Wordcount (WC).

8.1.4 Baselines. We compare **Scarf** with the following state-of-the-art DSPS tuning methods and MOO-based tuning methods:

- **DS2** [36]: A linearity-based model assuming that system performance is proportional to operator parallelism.
- **Bayesian** [35]: A Bayesian optimization-based method that builds a performance model with a random forest and uses it to guide the exploration of the knob space.
- **ContTune** [43]: A hybrid model that employs conservative, heuristic explorations to prevent SLA violations, while accelerating tuning with Bayesian optimization.
- **ZeroTune** [12]: A zero-shot method that constructs a cost model to determine initial operator parallelisms and uses operator selectivity to guide the subsequent tuning process.
- **StreamTune** [33]: A GNN-based method that pretrains a series of encoders on historical jobs and refines the configurations online by predicting the bottleneck operator.
- **IAMA** [68]: An evolutionary MOO-based method that finds approximations of the Pareto front using dynamic programming and incrementally refines them.
- **UDAO** [75]: A gradient-based MOO-based method that generates the Pareto front using learned performance models and a multi-objective gradient descent solver.

8.1.5 Training Setup. Our method consists of three phases: historical workload collection, offline training, and online tuning. During historical workload collection, we train a model from scratch for 12 hours for each query in Section 8.1.3. In offline training, we first ensure the current query is unseen by excluding its previously trained model from the historical model set. Then, we assign the new job to the most similar cluster, transfer the RL model forest from the cluster centroid to the new job, and finetune the models with PNN for 100 steps with new and replayed experiences, which takes an average of 3.7 hours per query. Finally, we identify the Pareto front with the highest quality for online tuning.

For fairness, baseline methods that support workload transfer (i.e. ZeroTune and StreamTune) are also trained with the same historical workload. All baselines perform a warmup process for 100 steps before we start evaluating the results. All methods are limited to an online tuning budget of 10 steps, as excessive reconfiguration is impractical in production environments. Important parameter ranges and default values we use are shown in Table 2.

8.1.6 Evaluation Metrics. Throughout the evaluation, we measure the tuning performance using the following metrics:

(1) **Resource efficiency**: CPU cores (r_1) and memory (r_2) allocated by YARN to Flink according to the selected job configuration.

(2) **Throughput**: the job throughput after the tuning stabilizes under a specific source rate. If a tuning decision is optimal, the throughput should be equal to the source rate. Poor tuning decisions will result in significantly lower throughput than the source rate.

(3) **Watermark delay**: the largest difference between the watermarks of source and sink operators after stabilization. This delay captures the total time that a tuple travels along the job graph from source to sink. Due to batching and windowing, this delay is greater than zero and dependent on the specific workload.

(4) **Reconfiguration count**: the average number of reconfigurations following a workload change.

(5) **Total tuning time**: the elapsed time from a workload change to throughput stabilization. This includes model deduction, checkpoint creation, job restart, and state recovery.

8.2 Comparison with Baselines

We first compare **Scarf** with the baselines in terms of resource efficiency, throughput, watermark delay and tuning overhead.

8.2.1 Job Performance. We compare the average throughput and watermark delay throughout all 10 source rates to evaluate how effectively each method sustains the performance required by the source rates. The results are shown in Figures 5 (a) and 5 (b). Since some queries do not expose watermarks, Figure 5 (b) shows the watermark delay only for the remaining queries.

In theory, the average source rate over the entire tuning process is $5.5W_u$, as indicated by the gray dash line in Figure 5 (a). Overall, **Scarf** *consistently matches the source input rates*, demonstrating its capability to allocate sufficient resources for streaming jobs to run without back-pressure. This is attributed to the following reasons. First, **Scarf** identifies critical knobs that significantly impact performance, while some baselines (e.g., DS2, ContTune, ZeroTune, StreamTune) only tune operator parallelisms. This leads to failures in cases like Q15, which requires tuning the mini-batch size to achieve high throughput. Second, **Scarf** leverages direct system feedback via reinforcement learning, enabling it to capture accurate performance dynamics. In contrast, other baselines (e.g., Bayesian, IAMA, UDAO) rely on performance models to predict throughput, which are difficult to train within limited tuning budgets, often resulting in suboptimal configurations. Third, **Scarf**'s focus on offline training enables safe reconfigurations in production, while other baselines are prone to selecting poor configurations during online explorations, causing degraded performance.

Regarding watermark delay, **Scarf** maintains the lowest or near-lowest values for all workloads, with an average of 30.5%–84.6% decrease. Other methods incur higher latencies due to excessive online explorations (DS2, ContTune, ZeroTune, StreamTune) or convergence to suboptimal configurations (Bayesian, IAMA, UDAO).

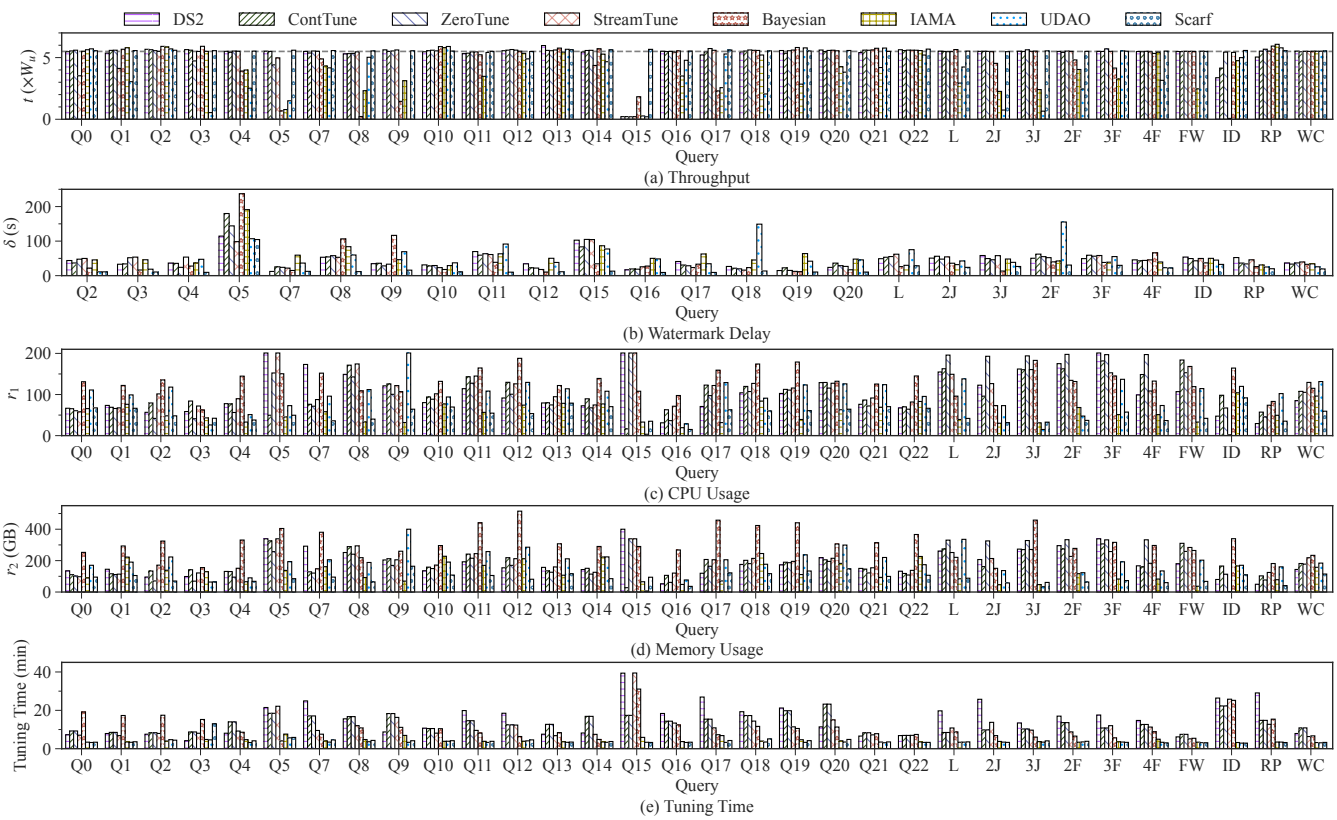

Figure 5: Average Throughput (t), Watermark Delay (δ), CPU Usage (r_1), Memory Usage (r_2) and Tuning Time Comparisons of Scarf and Baselines.

Table 2: Parameter Ranges and Default Values.

Parameter	Range
Number of Knobs Selected K	5, 10, **20**, 40
Population Size $\lvert\mathcal{R}\rvert$	1, **5**, 10, 20
Offline Training Steps T	20, 40, 60, 80, **100**
Actor Learning Rate η_a	1e-5, **3e-5**, 1e-4, 3e-4
Critic Learning Rate η_c	1e-4, **3e-4**, 1e-3, 3e-3

8.2.2 Resource Efficiency. We compare the average CPU and memory usage across all 10 source rates. The results are shown in Figures 5 (c) and 5 (d). Note that in some cases, certain baselines under-allocate resources (e.g., UDAO in Q3), resulting in lower usage than **Scarf** but failing to meet the throughput requirements. These cases are excluded from the subsequent analysis.

Overall, our method *delivers the most resource efficiency across workloads,* consistently outperforming baselines. Out of all workloads, **Scarf** achieves resource savings of 29.6%–62.5% (average 46.3%) for CPU and 27.8%–68.3% (average 48.7%) for memory compared to baseline methods. Its reinforcement learning approach explores the knob space without assuming any prior knowledge of knob impacts, enabling robust adaptation. By contrast, some baselines such as DS2 and ZeroTune assume linearity between operator parallelism and throughput, which is not guaranteed to be correct. ContTune, Bayesian, and StreamTune utilize Bayesian Optimization, which struggles to converge in large configuration spaces. IAMA and UDAO often recommend lower resource usage than required, because evolutionary methods and gradient-descent MOO are ill-suited for the complex knob space of DSPSs.

Specifically, ***Scarf** yields greater improvements on more complex queries.* **Scarf** brings an average of 55.5% CPU and 52.6% memory savings on queries with multiple operators, compared to 23.7% CPU and 39.2% memory savings on single-operator queries (i.e., Q0–Q2, Q13, Q14, Q21, Q22, L, ID). This is because complex queries have larger job graphs, where **Scarf**'s GNN-based job representation better captures the intricate relationships among operators.

8.2.3 Knob Tuning Overhead. Figure 5 (e) presents the average reconfiguration time per source-rate change (i.e., the total elapsed time from a source rate change to job stabilization). Note that we limit all methods to 10 reconfigurations per source-rate range. **Scarf** ranks among the fastest methods because its offline-online tuning design minimizes online exploration. In contrast, DS2, ContTune, ZeroTune, and StreamTune perform extensive online explorations, resulting in longer reconfiguration times, while Bayesian, IAMA, and UDAO require less reconfiguration time at the expense of sub-optimal job performance.

Figure 6: Ablation of Components over Different Job Graph Sizes (N) vs CPU (r_1) and Memory (r_2) Usage.

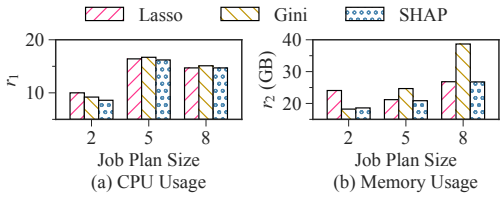

Figure 7: CPU (r_1) and Memory (r_2) Usage Comparisons of Different Knob Selection Methods.

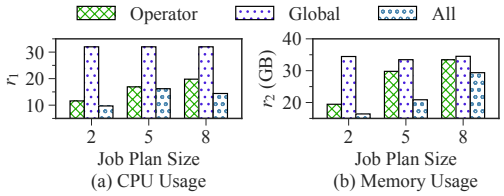

Figure 8: CPU (r_1) and Memory (r_2) Usage Comparisons Tuning Operator-Level, Global, and All Knobs.

8.3 Ablation Studies

8.3.1 Core Components. We test on all Nexmark queries with three ablated variants: w/o knob selection, w/o multi-objective learning, and w/o transfer learning. The results are shown in Figure 6.

Knob Selection. We remove knob selection by tuning 141 configurations, which include all Flink configurations except those irrelevant to performance (e.g., log paths, high-availability). The results show that knob selection accounts for a 0.8%–41.7% (average 11.6%) improvement in CPU efficiency and a 1.4%–50.4% (average 19.0%) improvement in memory efficiency. This demonstrates that knob selection effectively reduces the configuration space, making it easier for RL agents to learn the optimal tuning actions.

Multi-Objective Learning. We remove multi-objective learning by excluding the resource component from the RL reward. We find that multi-objective learning contributes 3.6%–119.3% (average 32.2%) improvement in CPU efficiency and 13.4%–140.6% (average 76.9%) improvement in memory efficiency. This is because multi-objective learning balances resource efficiency with job performance, enabling the model to avoid over-allocation under low workloads.

Transfer Learning. We disable transfer learning and train models from scratch instead of initializing with historic workloads. Within our 100-step offline training budget, transfer learning yields 2.4%–31.0% (average 14.2%) improvement in CPU efficiency and 1.5%–77.9% (average 28.5%) improvement in memory efficiency, proving that transferred models obtain Pareto fronts with better qualities.

8.3.2 Knob Importance Metric Ablation. To validate the effectiveness of SHAP values as the knob importance metric, we replace SHAP with Lasso and Gini score for knob selection. The results are shown in Figure 7. Lasso and Gini score both yield inferior performance in certain cases due to their inability to capture negative knob contributions, while SHAP consistently achieves the best CPU and memory efficiency across all queries.

8.3.3 Knob Space Ablation. We compare three knob spaces to tune: operator-level knobs (i.e., parallelisms), global knobs, and all knobs. The results are presented in Figure 8. It is evident that the operator parallelisms have a greater impact than global knobs. Nevertheless, global knobs contribute meaningful resource efficiencies when tuned together with operator-level knobs, yielding 4%–27% CPU savings and 12%–29% memory savings. Therefore, tuning operator-level and global knobs concurrently is more beneficial.

8.4 Parameter Studies

8.4.1 Number of Selected Knobs. We vary the number of selected knobs on Nexmark queries of different complexity. The results are shown in Figure 9. For simple queries, we find that the top-5 knobs already deliver satisfactory performance, while including more knobs makes it harder for the RL agent to learn effectively. However, for medium to complex queries, clear performance improvements are observed when the number of knobs increases from 10 to 20. Therefore, we recommend selecting the number of knobs in proportion to the job's complexity.

8.4.2 Number of Offline Training Steps. We compare the convergence process of new and transferred models by studying the relationship between the number of offline training steps and tuning performance. The results are shown in Figure 10. Some points are missing because the online tuning process fails to suggest a configuration that sustains the $10W_u$ throughput. It can be shown that new models typically have difficulty converging within 100 steps, while transferred models can converge within 60–80 steps, with comparable or better final tuning performances. This proves that knowledge transfer speeds up the offline training process, accelerating the adaptation to new workloads.

8.5 Cross-System Generalizability

We compare **Scarf** on Spark Structured Streaming with baseline methods using HiBench-stream workloads. We exclude DS2, ContTune, ZeroTune, and StreamTune as these methods rely on Flink-specific metrics to estimate operator processing abilities, making them incompatible with Spark. Consequently, we compare only against Bayesian, IAMA, and UDAO. As shown in Figure 11 (a), these baselines fail to sustain the required throughput, reflecting

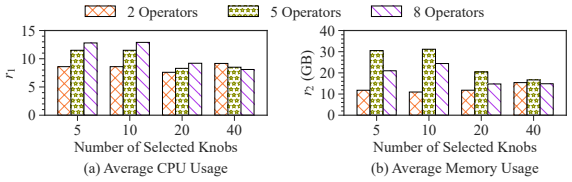

Figure 9: CPU (r_1) and Memory (r_2) Usage vs. Number of Selected Knobs.

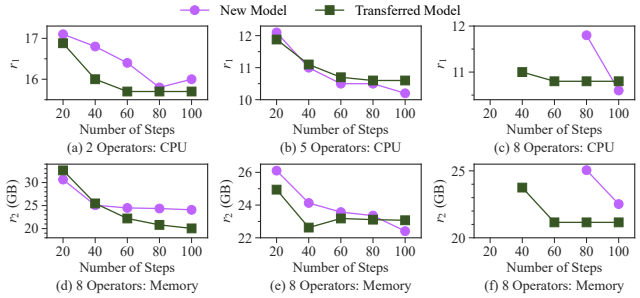

Figure 10: Convergence of New and Transferred Models In Offline Training.

the difficulty of building accurate performance models for Spark within a limited tuning budget. Although Figures 11 (b) and 11 (c) show that IAMA can use fewer resources, its suboptimal throughput indicates under-allocation. These results demonstrate **Scarf**'s ability to generalize to alternative DSPSs.

9 RELATED WORK

Distributed Stream Processing System Tuning. A subset of work on DSPSs focuses exclusively on parallelism tuning, also known as *autoscaling*. Pioneering approaches adopt threshold-based policies [21, 27, 30, 47, 52, 74] to determine when and how to perform scaling operations. However, these policies lack feedback mechanisms, which limits their effectiveness. Model-based methods rely on linear models [15, 36, 57], queuing theory [28, 46, 59], control theory [22], or game theory [54] to formulate the tuning problem. Such methods often depend on strong assumptions about data distributions, making them less robust in practice. More recent methods employ machine learning, e.g., Bayesian optimization [43, 44, 61, 77], deep learning [12, 33, 48], and reinforcement learning [20, 24, 34, 58, 60, 73]. Nevertheless, these methods continue to focus solely on operator parallelism and omit cluster-level knobs entirely. A smaller body of work include additional knobs [23, 25, 31, 35], but these knobs are manually selected without theoretical grounding, and are sensitive to breaking changes in DSPSs. *In contrast, our method considers both operator parallelism and cluster-level knobs, thereby unlocking the full potential of DSPSs.*

OLAP and Big-Data Framework Tuning. Another line of work investigates parameter tuning for OLAP databases. These studies employ heuristic approaches [10, 14], Bayesian optimization [38, 70], and reinforcement learning [17, 29, 41, 66, 71, 76]. Other methods

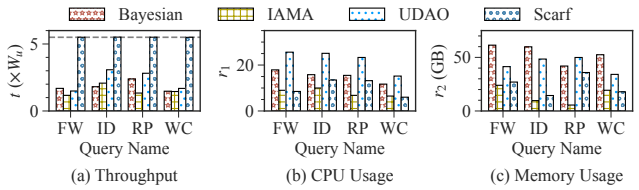

Figure 11: Average Throughput (t), CPU Usage (r_1) and Memory Usage (r_2) Comparisons of Scarf and Baselines on Spark Structured Streaming.

target parameter tuning for big-data frameworks, such as Apache Spark [1], in [42, 50, 63]. However, they are unsuitable for streaming scenarios. The core limitation is that they simply maximize throughput, without conserving resources under various workloads for streaming jobs. *In contrast, our approach formulates the problem as multi-objective optimization, thereby enabling dynamic configuration adjustments in response to workload shifts.*

Multi-Objective Optimization for Knob Tuning. Several other MOO methods have been proposed for knob tuning, which optimize a set of metrics (e.g., execution time, monetary cost). **Evolutionary methods** [39, 67–69] iteratively evolve a candidate population of configurations to approximate the Pareto front. These methods work well for decomposable problem spaces with near-optimal substructure, e.g., scheduling plans [39] and query plans [67–69]. However, the interdependencies among knob space dimensions prevent decomposability, making evolutionary techniques suboptimal. **Gradient-based methods** [64, 75] use gradient information to guide the search for optimal configurations. However, they require a subdifferentiable performance model, which is difficult to build for DSPSs under constraining budgets due to the high-dimensional knob space and complex dynamics. *In contrast, our RL-based method uses direct feedback from actual workloads, allowing effectively explorations of the complex knob space without relying on inaccurate assumptions or modeling.*

10 CONCLUSION

In this paper, we present **Scarf**, a self-adaptive configuration tuning framework for Apache Flink, with three key contributions: (1) knob selection speedup via workload clustering, (2) MORL for adaptive tuning, and (3) transferable GNN-based actor-critic networks. Extensive evaluations demonstrate that **Scarf** substantially improves tuning efficiency and resource utilization compared with state-of-the-art methods. In future work, we will further improve tuning stability in online environments by addressing job start-up failures, network jitter, misconfigurations, and related issues.

ACKNOWLEDGMENTS

This work was supported in part by the NSFC under Grants No. (62025206, 62402422, and U23A20296), Yongjiang Talent Introduction Programme (2024A-162-G), Zhejiang Provincial Natural Science Foundation of China under Grant No. LZ25F020001, and Zhejiang Province's "Lingyan" R&D Project under Grant No. 2024C01259. Yunjun Gao is the corresponding author.

REFERENCES

[1] 2018. Apache Spark. https://spark.apache.org/ [Accessed 03-09-2026].

[2] 2024. Apache Flink® — Stateful Computations over Data Streams. https://flink.apache.org [Accessed 03-09-2026].

[3] 2024. Apache Storm. https://storm.apache.org [Accessed 03-09-2026].

[4] 2024. GitHub - Intel-bigdata/HiBench: HiBench is a big data benchmark suite. https://github.com/Intel-bigdata/HiBench. [Accessed 03-09-2026].

[5] 2024. Kafka Streams. https://kafka.apache.org/documentation/streams/ [Accessed 03-09-2026].

[6] 2024. nexmark/nexmark: Benchmarks for queries over continuous data streams. https://github.com/nexmark/nexmark [Accessed 03-09-2026].

[7] 2025. Autoscaler parameter autotuning. https://docs.aws.amazon.com/emr/latest/EMR-on-EKS-DevelopmentGuide/jobruns-flink-autoscaler-parameter-autotuning.html [Accessed 03-09-2026].

[8] 2025. Autoscaling Apache Flink with Ververica Platform Autopilot. https://docs.confluent.io/cloud/current/flink/concepts/autopilot.html [Accessed 03-09-2026].

[9] 2025. Configure automatic tuning - Realtime Compute for Apache Flink. https://www.alibabacloud.com/help/en/flink/configure-autopilot-and-scheduled-tuning [Accessed 03-09-2026].

[10] 2025. *le0pard/pgtune*. https://github.com/le0pard/pgtune

[11] 2025. Metrics | Apache Flink. https://nightlies.apache.org/flink/flink-docs-release-2.0/docs/ops/metrics/ [Accessed 03-09-2026].

[12] Pratyush Agnihotri, Boris Koldehofe, Paul Stiegele, Roman Heinrich, Carsten Binnig, and Manisha Luthra. 2024. ZeroTune: Learned Zero-shot Cost Models for Parallelism Tuning in Stream Processing. In *2024 IEEE 40th International Conference on Data Engineering*. 2040–2053. https://doi.org/10.1109/ICDE60146.2024.00163

[13] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proceedings of the VLDB Endowment* 8, 12 (Aug. 2015), 1792–1803. https://doi.org/10.14778/2824032.2824076

[14] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An Extensible Framework for Program Autotuning. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques*. 303–315. https://doi.org/10.1145/2628071.2628092

[15] HamidReza Arkian, Guillaume Pierre, Johan Tordsson, and Erik Elmroth. 2021. Model-based Stream Processing Auto-scaling in Geo-distributed Environments. In *2021 30th International Conference on Computer Communications and Networks*. 1–11. https://doi.org/10.1109/ICCCN52240.2021.9522236

[16] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data*. 601–613. https://doi.org/10.1145/3183713.3190664

[17] Baoqing Cai, Yu Liu, Ce Zhang, Guangyu Zhang, Ke Zhou, Li Liu, Chunhua Li, Bin Cheng, Jie Yang, and Jiashu Xing. 2022. HUNTER: An Online Cloud Database Hybrid Tuning System for Personalized Requirements. In *Proceedings of the 2022 International Conference on Management of Data*. 646–659. https://doi.org/10.1145/3514221.3517882

[18] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *Proceedings of the VLDB Endowment* 10, 12 (Aug. 2017), 1718–1729. https://doi.org/10.14778/3137765.3137777

[19] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015), 28–38.

[20] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. 2018. Decentralized Self-adaptation for Elastic Data Stream Processing. *Future Generation Computer Systems* 87 (Oct. 2018), 171–185. https://doi.org/10.1016/j.future.2018.05.025

[21] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2013. Integrating Scale Out and Fault Tolerance in Stream Processing Using Operator State Management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 725–736. https://doi.org/10.1145/2463676.2465282

[22] Tiziano De Matteis and Gabriele Mencagli. 2017. Elastic Scaling for Distributed Latency-Sensitive Data Stream Operators. In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. 61–68. https://doi.org/10.1109/PDP.2017.31

[23] Isuru Dharmadasa and Faheem Ullah. 2023. Co-Tuning of Cloud Infrastructure and Distributed Data Processing Platforms. In *2023 IEEE International Conference on Big Data*. 207–214. https://doi.org/10.1109/BigData59044.2023.10386759

[24] Dong Nguyen Doan, Daniela Zaharie, and Dana Petcu. 2020. Auto-scaling for a Streaming Architecture with Fuzzy Deep Reinforcement Learning. In *Euro-Par 2019: Parallel Processing Workshops*. 476–488. https://doi.org/10.1007/978-3-030-48340-1_37

[25] Hui Dou, Shanshan Zhu, Yuxuan Zhou, Yiwen Zhang, Jicheng Mei, Yang Wu, and Jiaqing Dai. 2024. COTuner: Joint Optimization of Resource Configuration and Software Parameters for Recurring Streaming Jobs on the Cloud. In *2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing*. 87–96. https://doi.org/10.1109/CCGrid59990.2024.00019

[26] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning Database Configuration Parameters with iTuned. *Proceedings of the VLDB Endowment* 2, 1 (Aug. 2009), 1246–1257. https://doi.org/10.14778/1687627.1687767

[27] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. 2017. Dhalion: Self-regulating Stream Processing in Heron. *Proceedings of the VLDB Endowment* 10, 12 (Aug. 2017), 1825–1836. https://doi.org/10.14778/3137765.3137786

[28] Tom Z. J. Fu, Jianbing Ding, Richard T. B. Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. 2017. DRS: Auto-Scaling for Real-Time Stream Analytics. *IEEE/ACM Transactions on Networking* 25, 6 (Dec. 2017), 3338–3352. https://doi.org/10.1109/TNET.2017.2741969

[29] Jia-Ke Ge, Yan-Feng Chai, and Yun-Peng Chai. 2021. WATuning: A Workload-Aware Tuning System with Attention-Based Deep Reinforcement Learning. *Journal of Computer Science and Technology* 36, 4 (July 2021), 741–761. https://doi.org/10.1007/s11390-021-1350-8

[30] Bugra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. 2014. Elastic Scaling for Data Stream Processing. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (June 2014), 1447–1463. https://doi.org/10.1109/TPDS.2013.295

[31] Yijin Guo, Huasong Shan, Shixin Huang, Kai Hwang, Jianping Fan, and Zhibin Yu. 2021. GML: Efficiently Auto-Tuning Flink's Configurations Via Guided Machine Learning. *IEEE Transactions on Parallel and Distributed Systems* 32, 12 (Dec. 2021), 2921–2935. https://doi.org/10.1109/TPDS.2021.3081600

[32] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. In *Proceedings of the 35th International Conference on Machine Learning*, Vol. 80. 1861–1870. https://proceedings.mlr.press/v80/haarnoja18b.html

[33] Yuxing Han, Lixiang Chen, Haoyu Wang, Zhanghao Chen, Yifan Zhang, Chengcheng Yang, Kongzhang Hao, and Zhengyi Yang. 2025. Learning from the Past: Adaptive Parallelism Tuning for Stream Processing Systems. In *2025 IEEE 41st International Conference on Data Engineering*. Los Alamitos, CA, USA, 3535–3548. https://doi.org/10.1109/ICDE65448.2025.00264

[34] Thomas Heinze, Valerio Pappalardo, Zbigniew Jerzak, and Christof Fetzer. 2014. Auto-scaling Techniques for Elastic Data Stream Processing. In *2014 IEEE 30th International Conference on Data Engineering Workshops*. 296–302. https://doi.org/10.1109/ICDEW.2014.6818344

[35] Shixin Huang, Chao Chen, Gangya Zhu, Jinhan Xin, Zheng Wang, Kai Hwang, and Zhibin Yu. 2022. Resource Configuration Tuning for Stream Data Processing Systems via Bayesian Optimization. *Intelligent Computing* 2022 (Jan. 2022), 2022/9820424. https://doi.org/10.34133/2022/9820424

[36] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three Steps is All You Need: Fast, Accurate, Automatic Scaling Decisions for Distributed Streaming Dataflows. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. 783–798. https://www.usenix.org/system/files/osdi18-kalavri.pdf

[37] Konstantinos Kanellis, Ramnatthan Alagappan, and Shivaram Venkataraman. 2020. Too Many Knobs to Tune? Towards Faster Database Tuning by Pre-selecting Important Knobs. In *Proceedings of the 12th USENIX Conference on Hot Topics in Storage and File Systems*. Article 16, 8 pages. https://www.usenix.org/system/files/hotstorage20_paper_kanellis.pdf

[38] 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 (July 2022), 2953–2965. https://doi.org/10.14778/3551793.3551844

[39] Herald Kllapi, Eva Sitaridi, Manolis M. Tsangaris, and Yannis Ioannidis. 2011. Schedule Optimization for Data Processing Flows on the Cloud. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. 289–300. https://doi.org/10.1145/1989323.1989355

[40] Emmanouil Kritharakis, Shengyao Luo, Vivek Unnikrishnan, and Karan Vombatkere. 2022. Detecting Trading Trends in Streaming Financial Data Using Apache Flink. In *Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems*. 145–150. https://doi.org/10.1145/3524860.3539647

[41] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: a query-aware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment* 12, 12 (Aug. 2019), 2118–2130. https://doi.org/10.14778/3352063.3352129

[42] Yang Li, Huaijun Jiang, Yu Shen, Yide Fang, Xiaofeng Yang, Danqing Huang, Xinyi Zhang, Wentao Zhang, Ce Zhang, Peng Chen, and Bin Cui. 2023. Towards General and Efficient Online Tuning for Spark. *Proceedings of the VLDB Endowment* 16,

12 (Aug. 2023), 3570–3583. https://doi.org/10.14778/3611540.3611548

[43] Jinqing Lian, Xinyi Zhang, Yingxia Shao, Zenglin Pu, Qingfeng Xiang, Yawen Li, and Bin Cui. 2023. ContTune: Continuous Tuning by Conservative Bayesian Optimization for Distributed Stream Data Processing Systems. *Proceedings of the VLDB Endowment* 16, 13 (Sept. 2023), 4282–4295. https://doi.org/10.14778/3625054.3625064

[44] Yang Liu, Huanle Xu, and Wing Cheong Lau. 2022. Online Resource Optimization for Elastic Stream Processing with Regret Guarantee. In *Proceedings of the 51st International Conference on Parallel Processing*. 1–11. https://doi.org/10.1145/3545008.3545063

[45] Zishuo Liu, Jingzhi Yan, and Yixin Lao. 2025. Real-Time Thermal Defect Detection in Additive Manufacturing with Apache Flink. In *Proceedings of the 19th ACM International Conference on Distributed and Event-Based Systems*. 229–234. https://doi.org/10.1145/3701717.3735574

[46] Björn Lohrmann, Peter Janacik, and Odej Kao. 2015. Elastic Stream Processing with Latency Guarantees. In *2015 IEEE 35th International Conference on Distributed Computing Systems*. 399–410. https://doi.org/10.1109/ICDCS.2015.48

[47] Federico Lombardi, Leonardo Aniello, Silvia Bonomi, and Leonardo Querzoni. 2018. Elastic Symbiotic Scaling of Operators and Resources in Stream Processing Systems. *IEEE Transactions on Parallel and Distributed Systems* 29, 3 (March 2018), 572–585. https://doi.org/10.1109/TPDS.2017.2762683

[48] Federico Lombardi, Andrea Muti, Leonardo Aniello, Roberto Baldoni, Silvia Bonomi, and Leonardo Querzoni. 2019. PASCAL: An Architecture for Proactive Auto-scaling of Distributed Services. *Future Generation Computer Systems* 98 (Sept. 2019), 342–361. https://doi.org/10.1016/j.future.2019.03.003

[49] Scott M. Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 4768–4777. https://papers.nips.cc/paper_files/paper/2017/file/8a20a8621978632d76c43dfd28b67767-Paper.pdf

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

[51] M. D. McKay, R. J. Beckman, and W. J. Conover. 1979. A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code. *Technometrics* 21, 2 (1979), 239–245. https://doi.org/10.2307/1268522

[52] Yuan Mei, Luwei Cheng, Vanish Talwar, Michael Y. Levin, Gabriela Jacques-Silva, Nikhil Simha, Anirban Banerjee, Brian Smith, Tim Williamson, Serhat Yilmaz, Weitao Chen, and Guoqiang Jerry Chen. 2020. Turbine: Facebook's Service Management Platform for Stream Processing. In *2020 IEEE 36th International Conference on Data Engineering*. 1591–1602. https://doi.org/10.1109/ICDE48307.2020.00141

[53] Yuan Mei, Rui Xia, Zhaoqian Lan, Kaitian Hu, Lei Huang, Paris Carbone, Yanfei Lei, Vasiliki Kalavri, Han Yin, and Feng Wang. 2025. Disaggregated State Management in Apache Flink® 2.0. *Proceedings of the VLDB Endowment* 18, 12 (Aug. 2025), 4846–4859. https://doi.org/10.14778/3750601.3750609

[54] Gabriele Mencagli. 2016. A Game-Theoretic Approach for Elastic Distributed Data Stream Processing. *ACM Transactions on Autonomous and Adaptive Systems* 11, 2, Article 13 (June 2016), 34 pages. https://doi.org/10.1145/2903146

[55] Stefano Nembrini, Inke R König, and Marvin N Wright. 2018. The Revival of the Gini Importance? *Bioinformatics* 34, 21 (May 2018), 3711–3718. https://doi.org/10.1093/bioinformatics/bty373

[56] Gautam Pal, Katie Atkinson, and Gangmin Li. 2023. Real-time User Clickstream Behavior Analysis Based on Apache Storm Streaming. *Electronic Commerce Research* 23, 3 (2023), 1829–1859. https://doi.org/10.1007/s10660-021-09518-4

[57] Benjamin J. J. Pfister, Dominik Scheinert, Morgan K. Geldenhuys, and Odej Kao. 2024. Daedalus: Self-Adaptive Horizontal Autoscaling for Resource Efficiency of Distributed Stream Processing Systems. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*. 130–141. https://doi.org/10.1145/3629526.3645042

[58] Haoran Qiu, Weichao Mao, Chen Wang, Hubertus Franke, Alaa Youssef, Zbigniew T. Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. 2023. AWARE: Automate Workload Autoscaling with Reinforcement Learning in Production Cloud Systems. In *2023 USENIX Annual Technical Conference*. 387–402. https://www.usenix.org/conference/atc23/presentation/qiu-haoran

[59] Gabriele Russo Russo, Valeria Cardellini, Giuliano Casale, and Francesco Lo Presti. 2021. MEAD: Model-Based Vertical Auto-Scaling for Data Stream Processing. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing*. 314–323. https://doi.org/10.1109/CCGrid51090.2021.00041

[60] Gabriele Russo Russo. 2023. Using Reinforcement Learning to Control Auto-Scaling of Distributed Applications. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*. 137–138. https://doi.org/10.1145/3578245.3585427

[61] Gabriele Russo Russo, Valeria Cardellini, and Francesco Lo Presti. 2023. Hierarchical Auto-scaling Policies for Data Stream Processing on Heterogeneous Resources. *ACM Transactions on Autonomous and Adaptive Systems* 18, 4, Article 14 (Oct. 2023), 44 pages. https://doi.org/10.1145/3597435

[62] Andrei A. Rusu, Neil C. Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. 2016. Progressive Neural Networks. *CoRR* abs/1606.04671 (2016). arXiv:1606.04671 http://arxiv.org/abs/1606.04671

[63] 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. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 4800–4812. https://doi.org/10.1145/3580305.3599953

[64] 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*. 396–407. https://doi.org/10.1109/ICDE51399.2021.00041

[65] Robert Tibshirani. 2018. Regression Shrinkage and Selection Via the Lasso. *Journal of the Royal Statistical Society: Series B (Methodological)* 58, 1 (Dec. 2018), 267–288. https://doi.org/10.1111/j.2517-6161.1996.tb02080.x

[66] Immanuel Trummer. 2022. DB-BERT: A Database Tuning Tool that "Reads the Manual". In *Proceedings of the 2022 International Conference on Management of Data*. 190–203. https://doi.org/10.1145/3514221.3517843

[67] Immanuel Trummer and Christoph Koch. 2014. Approximation Schemes for Many-Objective Query Optimization. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. 1299–1310. https://doi.org/10.1145/2588555.2610527

[68] Immanuel Trummer and Christoph Koch. 2015. An Incremental Anytime Algorithm for Multi-Objective Query Optimization. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1941–1953. https://doi.org/10.1145/2723372.2746484

[69] Immanuel Trummer and Christoph Koch. 2016. Multi-Objective Parametric Query Optimization. *ACM SIGMOD Record* 45, 1 (June 2016), 24–31. https://doi.org/10.1145/2949741.2949748

[70] 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. https://doi.org/10.1145/3035918.3064029

[71] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. 2021. UDO: Universal Database Optimization Using Reinforcement Learning. *Proceedings of the VLDB Endowment* 14, 13 (Sept. 2021), 3402–3414. https://doi.org/10.14778/3484224.3484236

[72] Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Yuchu Fang, Yeju Zhou, Yang Zheng, Zhenheng Tang, Xin He, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. 2025. BurstGPT: A Real-World Workload Dataset to Optimize LLM Serving Systems. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*. 5831–5841. https://doi.org/10.1145/3711896.3737413

[73] Mingzhe Xing, Hangyu Mao, and Zhen Xiao. 2022. Fast and Fine-grained Autoscaler for Streaming Jobs with Reinforcement Learning. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence*. 564–570. https://doi.org/10.24963/ijcai.2022/80

[74] Le Xu, Boyang Peng, and Indranil Gupta. 2016. Stela: Enabling Stream Processing Systems to Scale-in and Scale-out On-demand. In *2016 IEEE International Conference on Cloud Engineering*. 22–31. https://doi.org/10.1109/IC2E.2016.38

[75] Khaled Zaouk, Fei Song, Chenghao Lyu, Arnab Sinha, Yanlei Diao, and Prashant Shenoy. 2019. UDAO: A Next-generation Unified Data Analytics Optimizer. *Proceedings of the VLDB Endowment* 12, 12 (Aug. 2019), 1934–1937. https://doi.org/10.14778/3352063.3352103

[76] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 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. https://doi.org/10.1145/3299869.3300085

[77] Liang Zhang, Wenli Zheng, Chao Li, Yao Shen, and Minyi Guo. 2021. AuTraScale: An Automated and Transfer Learning Solution for Streaming System Auto-Scaling. In *2021 IEEE International Parallel and Distributed Processing Symposium*. 912–921. https://doi.org/10.1109/IPDPS49936.2021.00100

[78] 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 (May 2022), 1808–1821. https://doi.org/10.14778/3538598.3538604