Cluster Computing (2026) 29:52
https://doi.org/10.1007/s10586-025-05808-w

Pa-Stream: pattern-aware scheduling for distributed stream computing systems

Dawei Sun[1] · Yinuo Fan[1] · Ning Zhang[1] · Shang Gao[2] · Jianguo Yu[3] · Rajkumar Buyya[4]

Received: 15 March 2025 / Revised: 10 October 2025 / Accepted: 22 October 2025
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2025

Abstract

Adjusting operator allocations based on changes in system metrics is one of the key characteristic in distributed stream computing systems. However, Existing work often fail to detect potential pattern features within data streams, replying instead on outdated information for scheduling decisions. This results in delayed responses to data stream fluctuations, causing significant performance volatility. To address these challenges, this paper proposes a pattern-aware scheduling strategy called Pa-Stream. The main contributions of this work include: (1) Validation of performance issues: Through experiments conducted on Alibaba Cloud, we evaluate the performance of Storm's Resource Aware Scheduler under fluctuating data streams. The results demonstrate that variations in data streams degrade system performance and lead to resource waste. (2) Data stream prediction strategy: We introduce a data stream prediction algorithm based on the Long Short-Term Memory (LSTM) network to identify data stream patterns and predict system performance. (3) Initial scheduling strategy: A novel scheduling strategy based on bin-packing algorithms and multi-objective non-dominated sorting is proposed for the initial scheduling of operators. This approach addresses the limitations of traditional bin-packing algorithms in handling scheduling challenges in heterogeneous clusters. (4) Runtime scheduling strategy: For runtime scheduling, we design a strategy based on the Deep Q-network (DQN). This strategy incorporates DQN training, a scheduling scheme generation algorithm, and an online scheduling algorithm to optimize runtime decision-making. (5) Implementation and evaluation of Pa-Stream: We deploy Pa-Stream and validate its performance through extensive experiments. The results show that, compared to SP-Ant and R-storm, Pa-Stream reduces latency by up to 57.24%, increases throughput by up to 76.18%, and decreases system load by up to 52.91%.

Keywords Distributed computing system · Data stream pattern · Operator scheduling · Long short-term memory network · Deep Q-network

✉ Dawei Sun
sundaweicn@cugb.edu.cn

Yinuo Fan
fanyinuocn@email.cugb.edu.cn

Ning Zhang
zhangning@email.cugb.edu.cn

Shang Gao
shang.gao@deakin.edu.au

Jianguo Yu
yjg@zua.edu.cn

Rajkumar Buyya
rbuyya@unimelb.edu.au

[1] School of Information Engineering, China University of Geosciences, Beijing 10083, China

[2] School of Information Technology, Deakin University, Geelong 3216, Victoria, Australia

[3] School of Computer Science, Zhengzhou University of Aeronautics, Zhengzhou 450015, Henan, China

[4] Quantum Cloud Computing and Distributed Systems (qCLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Parkville 3010, Victoria, Australia

Published online: 21 November 2025

1 Introduction

Stream computing has emerged as a critical area of research as the scale and complexity of real-time data in various fields continue to grow [1, 2]. Distributed stream processing systems are specifically designed to handle real-time data streams, capable of receiving, processing, and analyzing continuously generated data. Unlike traditional batch processing systems, which handle static datasets offline, stream processing systems focus on the continuous, real-time processing of dynamic data streams. In distributed stream computing (DSC) systems, operators, implementing user-defined logic, perform real-time computations, transformations, and aggregations on the incoming data. The processed results are then passed to other operators for further analysis.

Operator scheduling is a crucial process in distributed stream computing systems. It determines how operators in stream processing applications are deployed across different computing nodes [3, 4]. To improve processing efficiency, each operator can contain multiple parallel executors or instances, which can be placed on different computing nodes. Additionally, multiple operators, whether identical or different, can run on the same computing node. Finding the optimal operator placement layout in such systems is an NP-hard problem, requiring approximation methods to enhance system performance [5]. Moreover, clusters are often heterogeneous [6, 7]. In heterogeneous clusters, nodes exhibit varying computational capabilities, and communication bandwidth and latency may differ significantly among them [8, 9]. The placement of executors with diverse computational and communication requirements onto nodes with varying performance characteristics further complicates the scheduling process.

Due to the real-time nature of stream computing, fluctuations in data stream can lead to node overloading or underutilization, causing increased data processing latency, reduced throughput, and wasted resources [10–12]. Therefore, a suitable reconfiguration plan is necessary to evaluate whether node overloading or resource wastage occurs under the current placement scheme when significant changes in the data stream arise. By employing appropriate allocation strategies, it is possible to generate more efficient operator allocation schemes that maintain performance metrics during sudden data stream fluctuations while ensuring optimal utilization of system resources.

Most existing approaches rely on passive adjustments based solely on current load conditions, showing clear limitations when faced with fluctuating data streams. (1) Poor predictive capability: They fail to accurately forecast future workload patterns and proactively adjust scheduling strategies. (2) Weak learning capability: Existing schedulers are mainly rule-based or heuristic-driven, with limited self-optimization ability during long-term operation. Therefore, there is a need for a scheduler that can accurately model future stream dynamics and proactively adapt resource allocation, ensuring robust performance under complex and fluctuating data stream scenarios.

To address the operator scheduling problem, this paper proposes a data stream pattern-aware scheduling strategy called Pa-Stream. By leveraging the Long Short-Term Memory (LSTM) network to identify pattern information in data streams, Pa-Stream monitors changes in real time during system operation, predicts system performance, and adjusts operator allocation schemes before performance declines. The strategy uses a variable-length action reinforcement learning (RL) algorithm, which is faster at solving the optimal scheduling problem compared to fixed-length action RL algorithms. As a result, Pa-Stream is more responsive in system scheduling, generates scheduling schemes more quickly, and operates smoothly and efficiently.

1.1 Paper contributions

The key research contributions of this paper are:

(1) **Validation of performance issues.** We set up a computing cluster on Alibaba Cloud servers to evaluate the performance of the widely-used Resource Aware Scheduler scheduler in Storm under fluctuating data streams. Experimental results reveal that rapid increases in data stream rates significantly raise system latency and load, degrading performance. Conversely, sharp decreases in data stream rates result in excessively low load, leading to resource waste.

(2) **Data stream prediction strategy.** We propose a data stream prediction algorithm based on the LSTM network to capture pattern features in data streams and predict system performance. This algorithm models the relationship between data stream velocity, operator allocation schemes in heterogeneous clusters, and system performance metrics.

(3) **Initial scheduling strategy.** A scheduling strategy based on bin-packing algorithms and multi-objective non-dominated sorting is proposed for initial operator scheduling. This algorithm performs multi-objective non-dominated sorting of nodes, subdivides nodes at each sorting level, and then applies the bin-packing algorithm to allocate operator instances efficiently to appropriate computing resources.

(4) **Runtime scheduling strategy.** We design a runtime scheduling strategy based on the Deep Q-network (DQN). This strategy includes DQN training, a scheduling scheme generation algorithm, and an online scheduling algorithm. The scheme generation algorithm utilizes

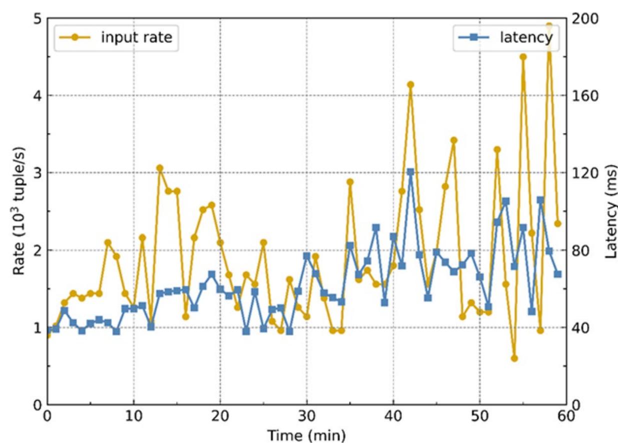

Fig. 1 System latency variations caused by data stream fluctuations

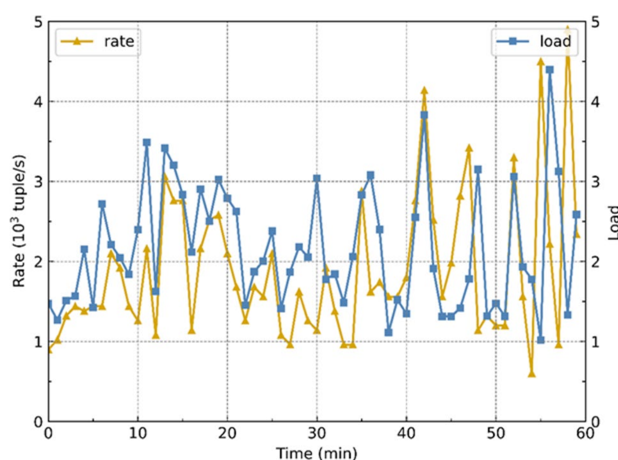

Fig. 2 System load variations caused by data stream fluctuations

the trained DQN to generate new scheduling schemes, while the online scheduling algorithm dynamically invokes the scheme generation algorithm as needed.

(5) **Implementation and evaluation.** We implement the proposed Pa-Stream framework on the Apache Storm platform and evaluate key system metrics, including latency, throughput, and resource utilization, under real-world streaming scenarios. The experimental results validate the effectiveness and robustness of Pa-Stream.

1.2 Paper organization

The rest of this paper is organized as follows: Section 2 examines the impact of data stream patterns on system performance. Section 3 models the operator scheduling problem for stream computing applications in heterogeneous clusters. Section 4 introduces the Pa-Stream scheduling strategy, detailing its framework and the design of its key components, including the initial scheduling strategy based on multi-objective bin packing, the LSTM-based data stream prediction strategy, and the reinforcement learning-based online scheduling strategy. Section 5 evaluates the performance of Pa-Stream through comparative experiments. Section 6 analyzes the current state of research and limitations of stream computing operator scheduling, and introduces related work on data stream pattern recognition algorithms. Finally, Section 7 summarizes the research findings, discusses limitations, and outlines future research directions.

2 Observations

We intuitively illustrate the impact of data stream volatility on system performance through experimental observations, emphasizing the necessity of adjusting scheduling strategy under fluctuating input streams. This experiment aims to discover potential issues of traditional schedulers, such as latency instability and load fluctuation, when handling fluctuating data streams in a real cloud environment. The effectiveness of Pa-Stream is to be validated through systematic experiments presented in Section 5. The observations are based on a 6-node Storm cluster (each node equipped with a 2 GHz, 2-core CPU and 2 GB memory). The input stream rate fluctuates uniformly between 500 and 5000 tuples/s. The application topology used is WordCount, which is representative of real-world streaming analytics tasks.

Significant changes in the input data stream of a stream computing system can lead to variations in system load. To verify the impact of data stream fluctuations on system performance, we deploy a heterogeneous cluster using Alibaba Cloud servers and evaluate the performance of Storm's widely used scheduler, the Resource Aware Scheduler (RAS), under varying data stream conditions. Figs. 1 and 2 illustrate the changes in system latency and load as the data stream rate fluctuates. In these figures, the yellow line represents the changes in data stream rate, while the blue lines show variations in system latency and load, respectively.

Analysis of the data reveals that when the data stream rate increases rapidly, the system's latency and load rise correspondingly, leading to a significant decline in performance. Conversely, when the data stream rate decreases sharply, system load becomes excessively low, causing resource wastage. It is necessary to evaluate and adjust existing allocation strategies in response to data stream fluctuations for system performance stability. An effective approach for addressing stream fluctuations is to predict the upcoming changes in the stream in advance. By forecasting these variations, potential performance impacts can be anticipated, allowing timely adjustments to the scheduling strategy to ensure stable system performance.

Table 1 List of Symbols

Symbol	Description
G	Directed acyclic graph
O	Set of operators
E	Set of the directed edges in the DAG
M	Set of machines
$I(o_i)$	Instance set of operator o_i
$v_{i,k}$	The k-th instance of operator o_i, where $k \in \{1, 2, \cdots, \lvert I(o_i)\rvert\}$
m	Number of operators in the topology
n	Number of machines in the cluster
$x_{i,j}$	Number of instances of operator o_i on machine M_j
o^i_{cpu}	CPU power required for each instance of operator o_i
o^i_{mem}	Memory required for each instance of operator o_i
C^j_{cpu}	CPU of machine M_j
C^j_{mem}	Memory of machine M_j
C^j_{band}	Bandwidth of machine M_j
C	Total CPU power of the cluster
Mem	Total memory of the cluster
n_s	Number of slots per machine
U	Vector containing the loads of all operators
U_{o_i}	Load of operator o_i
T_k	End - to - end latency of the kth tuple
Z	System load
Y	System latency

3 System model

To facilitate the discussion of Pa-Stream, we models the scheduling problem for stream computing applications in heterogeneous clusters. First, the topology of stream computing system is introduced and modeled, followed by the construction of a delay and load model for system runtime. Then, a resource model for heterogeneous stream computing clusters is established, and the resources are partitioned. Finally, based on the topology and resource models, an operator allocation model for heterogeneous clusters is constructed, along with a resource constraint model for operator allocation. Table 1 lists the key symbols used in this paper.

3.1 Topological logic model

In a distributed stream computing system, an application topology can be represented as a Directed Acyclic Graph (DAG), defined as $G = (O, E)$ [13, 14]. The topology specifies all the components required for the execution of the stream computing application, including data sources, data processing components, and the relationships between them. In the DAG, each vertex represents an operator in the topology. Let there be m operators in total, denoted as $O = \{o_1, o_2, \ldots, o_m\}$. These operators are responsible for receiving or processing data within the topology. The set E represents the directed edges in the DAG, where each edge signifies the flow of data between two adjacent operators.

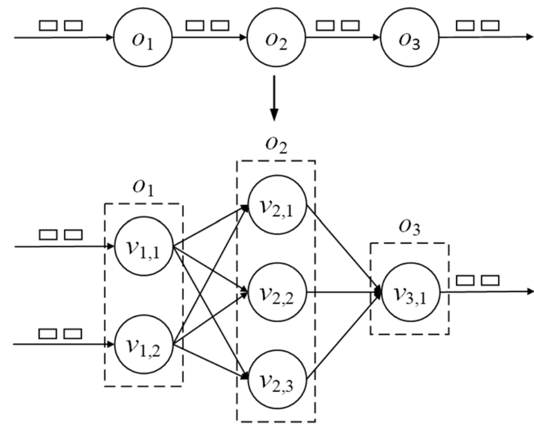

Fig. 3 Topological logic model

Each operator in a stream application consists of multiple parallel operator instances, and the execution of the application is carried out by these instances. The set of instances for an operator o_i is defined as $I(o_i) = \{v_{i,1}, v_{i,2}, \ldots, v_{i,|I(o_i)|}\}$, where $|I(o_i)|$ represents the number of instances of the operator. Due to differences in the processing logic of various operators, their computational complexities vary, resulting in heterogeneous CPU and memory resource requirements. Therefore, we need to clarify the resource requirements for each operator instance: the CPU demand for each parallel instance of operator o_i is represented as o^i_{cpu}, and the memory demand is represented as o^i_{mem}. To simplify the analysis, we assume that for a given operator, both the CPU and memory demands are the same across all its parallel instances.

As shown in Fig. 3, a simple topology includes three operators: o_1, o_2, and o_3. Operator o_1 functions as a Spout, responsible for receiving data from external sources and forwarding it into the topology, with a parallelism of 2 (i.e., 2 instances $v_{1,1}$ and $v_{1,2}$). Operators o_2 and o_3 are Bolts, tasked with receiving data from upstream operators and processing it according to user-defined logic, with parallelisms of 3 ($v_{2,1}$, $v_{2,2}$ and $v_{2,3}$) and 1 ($v_{3,1}$), respectively.

To facilitate a rigorous evaluation of the performance of Pa-Stream, we model several system performance metrics. One of the critical metrics is T_k, the end - to - end latency of the kth tuple, which refers to the time required for the k th tuple to be fully processed from its generation to completion. T_k is primarily composed of processing latency $T_{proc,k}$ and transmission latency $T_{tran,k}$, as shown in Eq. (1):

$$T_k = T_{proc,k} + T_{tran,k}, \tag{1}$$

where $T_{proc,k}$ represents the processing latency, which is the time taken to process the kth tuple across various components (including spouts and bolts). This latency includes both the waiting time in the tuple queue and the actual time spent processing this tuple. Additionally, $T_{tran,k}$ refers to the transmission latency of the kth tuple, which is the time taken to transmit the tuple between different operators.

Operator load is a crucial metric for assessing system performance. It is defined as the average load among all instances of an operator. The load of the kth instance of operator o_i is calculated as the product of the number of tuples awaiting processing within this instance during a specific detection interval and the average of tuple processing times, presented as a proportion of the detection interval. The load of the operator instance $v_{i,k}$ during system runtime is formulated in Eq. (2).

$$Load_{i,k} = \frac{(t_{i,k} \times Count_{i,k})}{Int_{i,k}} \tag{2}$$

where $t_{i,k}$ represents the average of tuple processing times of $v_{i,k}$, $Count_{i,k}$ refers to the number of tuples queued for processing in $v_{i,k}$, and $Int_{i,k}$ is the detection interval for $v_{i,k}$(e.g., 10 seconds in Apache Storm).

For an operator, a load value approaching 1 indicates that the operator is almost continuously processing tuples without idle time, which may become a bottleneck for improving system performance. If the load value exceeds 1, it suggests that the operator is overloaded, potentially leading to tuple backlogs. Conversely, a low load value means significant idle time, indicating underutilized resources. While this provides room for handling sudden increases in data streams, it also implies resource wastage due to under-utilization.

Based on the operator load and the tuple end-to-end latency, we define the system load Z as the average of all operator loads over a given time interval. The system latency Y is defined as the average end-to-end latencies of all tuples processed within the same time interval, as is shown in Eq. (3):

$$Y = \frac{1}{N}\sum_{k=1}^{N} T_k, \tag{3}$$

where N denotes the total number of tuples processed during the given time interval.

3.2 Computing resource model

In a heterogeneous cluster composed of n machines, denoted as $M_1, M_2, \ldots, M_n$, each machine M_j has distinct CPU and memory resources. The CPU resources of machine M_j are represented by C^j_{cpu}, defined as the product of the machine's processor frequency and the number of logical processors. The memory resources of machine M_j are denoted as C^j_{mem}, and its bandwidth as C^j_{band}. The total CPU resources C and memory M of the cluster are calculated by Eqs. (4) and (5).

$$C = \sum_{j=1}^{n} C^j_{cpu}, \tag{4}$$

$$Mem = \sum_{j=1}^{n} C^j_{mem}. \tag{5}$$

The operator instances in the topology are placed on the slots of machines during execution. Each slot accommodates one operator instance. To simplify the issue, we assume that each machine has the same number of slots, denoted as n_s, which indicates the maximum number of operator instances that can be placed on the machine, provided the CPU and memory resource requirements are satisfied.

To facilitate the initial scheduling of operators, machines in the cluster are categorized as either CPU-centric or memory-centric based on their configurations. A machine is defined as CPU-centric if the proportion of its CPU resources relative to the total cluster CPU resources exceeds the proportion of its memory resources relative to the total memory resources in the cluster. Otherwise, the machine is classified as memory-centric, as expressed in Eq. (6).

$$\frac{C^j_{cpu}}{C} > \frac{C^j_{mem}}{M}. \tag{6}$$

3.3 Operator allocation model

The operator scheduling problem involves assigning operator instances to machines within the cluster. Since operator instances run in parallel, instances of a single operator can be distributed across multiple machines, and a single machine can host multiple instances of either the same or different operators. Given the heterogeneous nature of the cluster, resource allocation and communication overheads differ depending on the machine to which an operator instance is assigned.

In a cluster with n machines (i.e., M_1, M_2, , M_n), let $x_{i,j}$ represent the number of instances of operator o_i placed on machine M_j. The scheduling problem in a heterogeneous cluster is to determine the values of $x_{i,j}$ for every operator o_i and machine M_j. Using Storm as an example, its scheduling process is dynamic and requires real-time adjustment of the operator instance allocation based on

the application topology's runtime status and system load. By adjusting operator parallelism and instance allocation, the system's performance and scalability can be improved, ensuring smooth operation of the system.

An effective scheduling strategy aims to maximize system performance under resource constraints. These constraints include CPU and memory resources and the number of available slots on each machine. The total CPU o^i_{cpu} and memory resources o^i_{mem} required by each operator instance $x_{i,j}$ placed on machine M_j must not exceed the machine's available CPU C^j_{cpu} and memory resources C^j_{mem}, as defined by constraints (7) and (8).

$$\sum_{i=1}^{m}(x_{i,j} \times o^i_{cpu}) \le C^j_{cpu}, \quad \forall j \in [1,n], \tag{7}$$

$$\sum_{i=1}^{m}(x_{i,j} \times o^i_{mem}) \le C^j_{mem}, \quad \forall j \in [1,n], \tag{8}$$

where *m* represents the number of operators in the topology. Furthermore, the total number of operator instances $x_{i,j}$ allocated on machine M_j must not exceed the machine's number of available slots n_s, as described in constraint (9).

$$0 \le \sum_{i=1}^{m} x_{ij} \le n_s. \tag{9}$$

4 Pa-Stream: architecture and algorithms

Pa-Stream is a data stream pattern awareness scheduling strategy designed for stream computing systems in heterogeneous clusters. It includes three strategies: an initial operator placement strategy, a data stream pattern awareness prediction strategy, and an operator migration strategy to handle variations in data streams during runtime.

4.1 System architecture

Pa-Stream is integrated into the open-source distributed stream computing platform Apache Storm. Please note that Apache Storm serves merely as the platform for our experiments, Pa-Stream is equally applicable to other stream processing platforms such as Spark Streaming [15] and Apache Flink [16]. Fig. 4 shows the architecture of Pa-Stream, which consists of the following key modules.

- **Metrics Collection Module**: This module consists of two components: Metrics Monitor and Database. Metrics Monitor collects runtime data from the stream computing system such as data stream rates and current operator placement schemes by leveraging the built-in `Metrics REST` API of Apache Storm, while Database persistently stores the data collected by the Metrics Monitor.
- **Prediction Module**: LSTM Network is the primary component of Prediction Module. Trained using historical data, this component models the relationship between data stream variations, operator placement schemes, and system performance in heterogeneous clusters. It predicts system performance changes to determine the timing of scheduling, and serves as the environment for the Agent to interact with during reinforcement learning.
- **Adaptive Scheduling Module**: There is two components in Adaptive Scheduling module: Agent and Scheduler. Agent is a DQN model that interacts with the environment, continuously learns, and generates new placement schemes. Scheduler executes both the

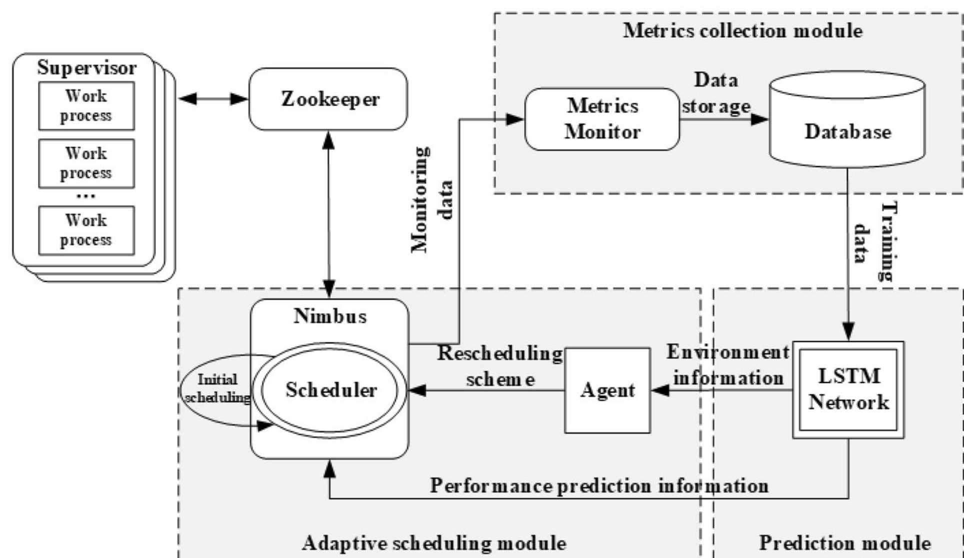

Fig. 4 System architecture of Pa-Stream

initial scheduling scheme and the online scheduling scheme. During online scheduling, it calculates the current system performance metrics and retrieves predictions of system performance changes from the LSTM network to evaluate whether rescheduling is necessary. If rescheduling is required, it obtains a new placement scheme from the Agent and executes it.

4.2 Bin-packing-based initial scheduling

To ensure optimal performance at the startup of the stream application, a multi-objective bin-packing algorithm [17] is employed for the initial scheduling of operators, allocating operator instances to computing resources in a reasonable manner. This algorithm employs multi-objective non-dominated sorting to solve scheduling problems within heterogeneous clusters.

First, each machine M_j in the heterogeneous cluster is evaluated based on its resources, including CPU, memory, and bandwidth, denoted as $(C^j_{cpu}, C^j_{mem}, C^j_{band})$. Utilizing metrics such as computing power, memory size, and network latency, multi-objective non-dominated sorting is conducted. The cluster machines are grouped into independent non-dominated layers, considered as Pareto fronts. A Pareto front represents the set of optimal trade-offs among conflicting objectives in a multi-objective optimization problem.

Next, Each machine is classified as either CPU-centric or memory-centric according to (6), and operators are classified as either CPU-sensitive or memory-sensitive. If the ratio of its CPU demand to the total CPU demand of all operators is higher than the ratio of its memory demand to the total memory demand of all operators, the operator is classified as CPU-sensitive. Otherwise, the operator is classified as memory-sensitive.

Then, the operators are topologically sorted according to their topological relationships, and their instances are placed according to this order. The operator instances will be placed on the machine within the optimal Pareto front. If there are multiple machines within the same Pareto front, the placement of the operator instances is determined according to the following rules.

1. CPU-sensitive operators: Place them on CPU-centric machines; if none are available, proceed to rule (3).
2. Memory-sensitive operators: Place them on memory-centric machines; if none are available, proceed to rule (3).
3. If no suitable machines are found in (1) or (2), or if multiple machines are available, calculate the communication cost to all upstream operator machines and select the machine with the minimum cost for placement.

Once an operator instance is assigned to a machine, the resources required by the instance are deducted from the machine's available resources, and the machine's remaining resource capacity and available slots are updated. Meanwhile, the machine is re-evaluated for its non-dominance relationship and adjusted to the appropriate Pareto Front layer. This process is repeated until the initial placement of all operator instances is completed.

4.3 LSTM-based data stream pattern prediction

In preliminary experiments, ARIMA and RNN models were tested but showed limited accuracy and stability, especially under fluctuating and non-stationary workloads. LSTM outperformed both models and was therefore adopted as the prediction component. We use an LSTM network to model the relationship between operator placement, data stream fluctuations, and system performance metrics. By training the LSTM on real-world data, we aim to rapidly predict changes in system performance during the early stages of stream processing applications, while maintaining good prediction accuracy. Additionally, during system operation, we perform incremental training with real-time data to adapt to long-term changes in data stream patterns.

The LSTM network is trained and evaluated using the **Twitter 2022 user behavior dataset** (30 GB), with **70%** used for training and **30%** for testing. The data is injected into the WordCount topology via **Kafka**. Two workload patterns are considered: (1) a steady stream at approximately 1000 tuples/s, and (2) a fluctuating stream uniformly sampled between 800 and 1200 tuples/s. The input features include historical stream rates and operator placement states, while the prediction targets are system latency and load. A **sliding window mechanism** is used to construct supervised samples, enabling the model to capture both short-term and long-term temporal dependencies.

The LSTM-based prediction model captures the complex relationships between data streams, operator allocation, and system performance. The network includes an input layer, an output layer, and four hidden layers. Fig. 5 illustrates the structure of our LSTM-based prediction model. The input layer corresponds to the input data rates and the allocation states of operators. The output layer predicts both the system latency and the system load. The hidden layers comprise three LSTM layers and one fully connected layer, effectively capturing the temporal information of data streams as they evolve over time. The fully layer maps the learned temporal information to the output layer.

Input layer: This layer is structured with a shape of (*TimeSteps*, *NumFeatures*), where $TimeSteps = 4$ specifies the number of time steps used as input to predict the system performance at the fifth time step. By limiting the input

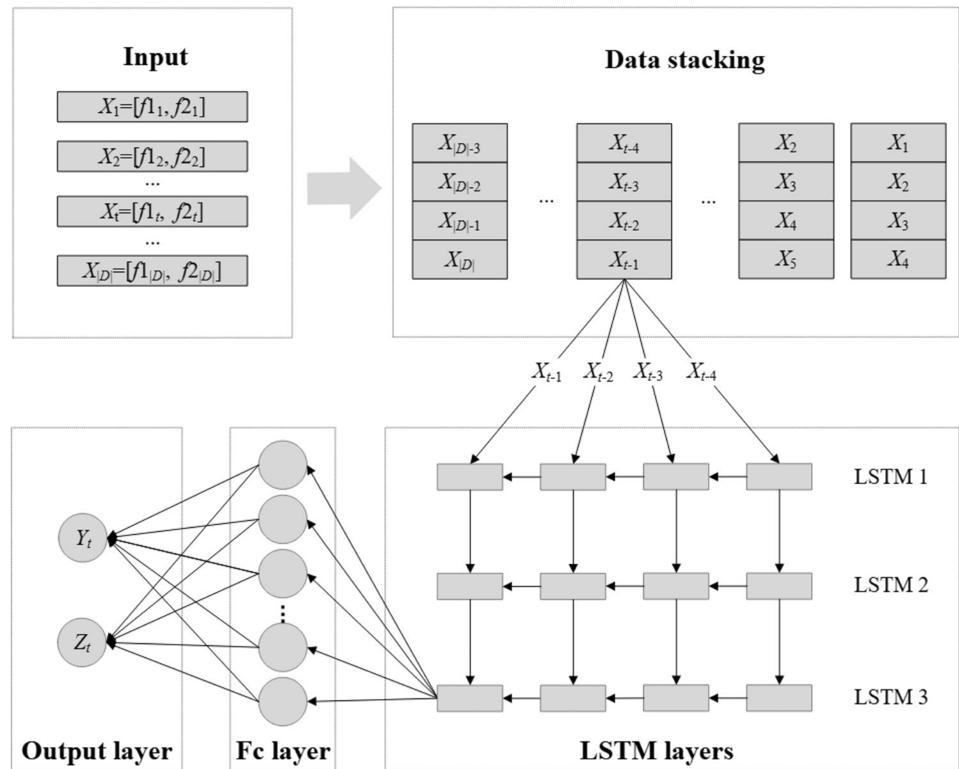

Fig. 5 LSTM-based prediction model

to four time steps, the model effectively captures short-term fluctuations in the data stream while maintaining manageable computational complexity.

The parameter *NumFeatures* represents the dimensionality of the feature vector for each time step and is based on two key metrics: data stream rate ($f1_t$) and operator placement scheme ($f2_t$).

(1) $f1_t$ is a scalar value representing the rate of incoming data at time *t*, reflecting the dynamic changes in the data stream.
(2) $f2_t$ is a vector that describes the allocation of operator instances across the cluster at time *t*. The length of $f2_t$ depends on the number of operators *m* and machines *n*, resulting in a vector of length $m \times n$.

These metrics are combined into a one-dimensional feature vector x_t for each time step, as shown in Eq. (10). Specifically, x_t is formed by concatenating the scalar $f1_t$ and the elements of the vector $f2_t$.

$$\mathrm{x}_t = [f1_t, f2_t], \tag{10}$$

The dimensionality of this feature vector, denoted as $NumFeatures$, is calculated based on the structure of $f2_t$ and the inclusion of $f1_t$. The value of $NumFeatures$ is given by Eq. (10).

$$NumFeatures = m \times n + 1, \tag{11}$$

where $m \times n$ corresponds to the length of $f2_t$, encoding the placement state of m operators across n machines, and the additional 1 accounts for the scalar $f1_t$. Although the input dimensionality varies with *m* and *n*, the core architecture of the LSTM, specifically its gating mechanism and hidden layer size, remains fixed and independent of the input size. Consequently, as the cluster scale changes, both the temporal modeling capability and the architectural stability of the LSTM are preserved.

Hidden layers: The hidden layers include four layers: the first layer with 256 LSTM units; the second with 1024 LSTM units; the third with 512 LSTM units; and a fully connected layer with 256 units. For the first two LSTM layers, $return_sequences$ parameter is set to True, allowing data to be passed to the next layer, thereby capturing detailed and implicit temporal information. In the third LSTM layer, $return_sequences$ is set to False, simplifying the information to facilitate network convergence and feature extraction.

We adopt the 256–1024–512–256 architecture based on the input dimensionality and required model capacity. This structure provides sufficient expressive power while maintaining generalization. Our experimental evaluation demonstrates that this configuration yields stable and accurate performance. In Fig. 5, each LSTM layer is expanded along the time dimension. Specifically, each layer contains four

time steps corresponding to input data from time step 1 to time step 4, denoted as t_1, t_2, t_3, and t_4. This design intuitively reflects the temporal dependency characteristics of the LSTM network when processing sequential data.

Output layer: The output layer consists of 2 units that output the predicted results: system latency and system load, as shown in Eq. (12). The output y_t at time t includes two parts, where Y_t and Z_t represent the prediction of system latency and system load at time t, respectively.

$$y_t = [Y_t, Z_t]. \tag{12}$$

In Fig. 5, the data stacking process employs a sliding window mechanism to reconstruct the time series. The window length is set to $s = 4$, and the original sequence data $\{x_1, x_2, \ldots, x_{|D|}\}$ is extracted using a stride of 1, generating the training sample set. Through this method, the original sequence is transformed into supervised learning samples with spatiotemporal correlations. For instance, the first two training samples are $([x_1, x_2, x_3, x_4], y_5)$ and $([x_2, x_3, x_4, x_5], y_6)$, effectively capturing both local patterns and long-term dependencies in the sequential data.

During training, the Mean Absolute Error (MAE), defined by Eq. (13), is used as the loss function. MAE is widely used to measure average prediction errors by calculating the sum of absolute differences between target values y_i and predicted values $\hat{y}_i$, emphasizing the magnitude of errors rather than their direction. In Eq. (13), n represents the total number of samples, referring to the number of target-prediction pairs used to compute the error. The ADAM gradient descent method is used to optimize the model for faster convergence and better performance.

$$\mathrm{MAE} = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i|. \tag{13}$$

The data stream prediction strategy consists of two key phases: training and prediction. Before system operation, raw data is collected to train the LSTM network. This raw data includes the data stream rate, operator allocation schemes, and corresponding system performance metrics at different time steps.

The training set cannot be directly used for training the LSTM network and must be processed using the sliding window method to transform it into feature-label pairs. Each sliding window encompasses data from five consecutive time steps. Specifically, the input features are derived from the first four time steps, which include the **data stream rate** ($f1_t$) and **operator placement scheme** ($f2_t$). The system performance metrics at the fifth time step, represented by **system latency** (Y_t) and **system load** (Z_t), serve as the corresponding labels for supervised learning. During system operation, the trained LSTM network is capable of predicting the system performance at the fifth time step based on the data from the previous four time steps.

After the LSTM model is trained, it can predict system performance for future time steps. Specifically, the model takes as input the data from the previous four time steps, including the data stream rate and operator placement scheme. It then outputs predictions for the system latency and load at the next time step. These predictions help evaluate the current system state and guide dynamic adjustments to optimize resource allocation and adapt to changes in data streams.

4.4 DQN-based runtime scheduling

To dynamically adapt to data stream fluctuations and efficiently utilize the resources of heterogeneous clusters, we propose an online scheduling strategy based on DQN, which integrates data stream prediction [18]. This strategy includes DQN training, a scheduling scheme generation algorithm, and an online scheduling algorithm.

To implement the proposed DQN-based runtime scheduling, we model the scheduling process as a Markov Decision Process (MDP). By leveraging the MDP framework, we can ensure that scheduling schemes are both adaptive to real-time changes in data stream patterns and aligned with long-term system performance goals. We design its state space, action space, reward function, and agent to align with the characteristics of stream computing scheduling.

State Space: A state describes a specific situation within the environment, while the state space defines the set of all possible states the environment can assume. The agent observes these states and makes decisions accordingly. As shown in Eq. (14), the state *st* represents the current state of the environment, comprising the allocation of operator instances on machines *X*, the system latency *Y*, and the vector containing the loads of all operators *U*.

For computational simplicity, the allocation of operators on machines is flattened into a one-dimensional representation, as shown in Eq. (15), where $x_{i,j}$ represents the number of instances of operator o_i placed on machine M_j. The vector containing the loads of all operators U is given by Eq. (16), where u_{o_i} indicates the load of operator o_i, *n* represents the number of machines, and *m* represents the number of operators.

$$st = (X, Y, U), \tag{14}$$

$$X = [[x_{1,1}, x_{1,2}, \ldots, x_{1,n}], \ldots, [x_{m,1}, x_{m,2}, \ldots, x_{m,n}]], \tag{15}$$

$$U = [u_{O_1}, u_{O_2}, \ldots, u_{O_m}]. \tag{16}$$

Action Space: The action space refers to the set of all possible actions that the agent can take in a given state. A scheduling scheme is generated through a series of actions, each defined as either increasing or decreasing the number of instances of operator o_i placed on machine M_j. The size of the action step determines the agent's exploration speed. Before execution, the model checks whether each scheduling action satisfies resource constraints. Actions that violate constraints are filtered out or replaced with the nearest feasible action. The reward function penalizes such actions, preventing the agent from repeatedly selecting infeasible scheduling decisions.

The step size, denoted as *l*, is defined as the number of instances of operator o_i that can be added to or removed from machine M_j, with the parameter *e* introduced to balance the step size. In this context, *e* is a small decimal in the range [0, 1]. With a probability of $1 - e$, the step size *l* is set to 1; with a probability of *e*, *l* is randomly selected from the remaining resources on the machine. The step size is given in Eg. (17), and the action *a* is given by Eq. (18). The size of the action space is $|A| = 2m \times n$, where *m* represents the number of machines and *n* represents the number of operators.

$$l = \begin{cases} 1, & \text{with probability } 1 - e, \\ [2, \text{remaining slots num}], & \text{with probability } e, \end{cases} \tag{17}$$

$$a = \langle x_{i,j} \pm l \rangle. \tag{18}$$

Reward Function: In reinforcement learning, the reward function assesses the quality of actions and plays a crucial role in guiding the agent's future action choices [19]. After the agent takes an action, if the action improves scheduling performance, the agent receives a positive reward, increasing the probability of choosing similar actions in the future. In stream computing, the primary objectives are to minimize system latency and maximizing processing capacity. Achieving these objectives requires an effective allocation scheme that reduces the end-to-end latency of data tuples while maintaining load balancing. Therefore, the reward function *R* incorporates both delay rewards and load rewards.

To address the dimensional differences between delay and load metrics, the arc-tangent function is applied to the delay, normalizing it to the range $[-1, 1]$. The delay reward is defined by Eq. (19). Here, $\Delta delay$ represents the change in system delay from the previous time step. A negative reward is given when the delay increases ($\Delta delay > 0$); conversely, a positive reward is assigned when the delay decreases.

$$R_{delay} = -\frac{2}{\pi} \arctan(\Delta delay). \tag{19}$$

The load reward is defined by Eq. (20), where Z represents the system load, and the parameter δ denotes the target system load. Runtime scheduling strategy aims to keep the system load close to this target. For instance, when $\delta = 0.8$, the system reserves some resources to accommodate sudden traffic spikes, while $\delta = 1.0$ implies full utilization of cluster machines. Excessive deviation of the system load from the target load δ results in a negative reward.

$$R_{load} = -(Z - \delta)^2 + 1. \tag{20}$$

To balance multiple optimization objectives (latency, load, and stability) during scheduling, the reward is formulated as a weighted combination, as shown in Eq. (21). A larger weighting coefficient b emphasizes delay-sensitive, optimizing end-to-end latency in the stream computing system. Conversely, a larger coefficient *c* prioritizes load-sensitive, focusing on optimizing the average load of operators in the system. The overall reward function is given by Eq. (22):

$$R = b \times R_{delay} + c \times R_{load}, \tag{21}$$

$$R = -\frac{2b}{\pi} \arctan(\Delta delay) - c\left((load - 0.8)^2 - 1\right), \tag{22}$$

where parameters *b* and *c* control the relative importance of latency and load objectives, allowing flexible policy preference. The parameter δ defines the target load level, penalizing deviations from it to prevent overload and enhance stability. This formulation provides interpretability and tunability across different runtime environments. The parameter δ is set to 0.8 to reserve about 20% redundancy for handling traffic fluctuations; the weights *b* and *c* are configured to reflect the latency-first objective while also accounting for load balancing.

Exploration Strategy: In reinforcement learning, action selection primarily involves balancing exploration and exploitation. Exploration refers to trying new actions to discover potential rewards, while exploitation involves choosing the action known to provide the highest return.

We utilize a hybrid method that combines the ϵ-greedy algorithm with the Softmax approach. Specifically, instead of selecting the action with the maximum Q-value in the exploitation phase of the ϵ-greedy algorithm, we apply the Softmax method, as shown in Eq. (23). This modification transforms the Q-values output by the Q-network into probabilities between 0 and 1, ensuring that the sum of all probabilities equals 1. These probabilities are then used to guide

action selection, enabling a smoother and more informed decision-making process.

$$\mathrm{Softmax}(x) = \frac{e^{x_i}}{\sum_i e^{x_i}}. \tag{23}$$

In Eq. (23), i represents the i-th action and x_i represents the Q-value for the i-th action. This approach provides a more balanced way to managing exploration and exploitation. Even if an action does not have the highest Q-value, it still has a chance of being selected. For actions with high Q-values, they are not chosen blindly but rather with a higher probability, reflecting their relative advantage.

The Q-value is computed using the action-value function, or Q-function, which assigns a value to each state-action pair. The iterative formula for updating the Q-function is given in Eq. (24), where $Q'(st, a)$ is the updated Q-value for action a in state *st*, r is the reward received after taking action a, and $\max_{a'} Q(st', a')$ is the maximum Q-value across all possible actions a' in the subsequent state st'. The discount factor γ adjusts the importance of future rewards: a larger γ emphasizes future rewards, whereas a smaller γ prioritizes immediate rewards. In practice, the Q-function iterative formula is often simplified by setting $\alpha = 1$.

$$Q'(st, a) \leftarrow Q(st, a) + \alpha[r + \gamma \max_{a'} Q(st', a') - Q(st, a)]. \tag{24}$$

Input: State space S, action space A, num of training episodes E, num of max steps N, epsilon decay δ, minimum epsilon $\epsilon_{\min}$
Output: Trained DQN model Q

1: $Q \leftarrow$ Initialize Q-network with random weights
2: $\epsilon \leftarrow 1.0$
   // Set initial exploration factor to 1.0 for maximum exploration
3: **for** each episode $e \in \{1, \ldots, E\}$ **do**
4:   $st \leftarrow$ initial state (reasonable random state)
5:   **for** each step $t \in \{1, \ldots, N\}$ **do**
6:     **if** $random_number < \epsilon$ **then**
7:       $a \leftarrow$ randomly select an action from A
8:     **else**
9:       $a \leftarrow argmax_a(st, a)$
       // Select the action with the highest Q-value for the current state
10:    **end if**
11:    $(st', r) \leftarrow$ ExecuteActionAndGetReward(st, a)
       // Execute action and obtain the reward r and the new state st'
12:    Store transition (st, a, r, st') in replay buffer
13:    Sample random batch from replay buffer
14:    Set $y = r + \gamma \max_{a'} Q(st', a')$ for the batch transitions
15:    Perform a gradient descent step on $(y - Q(st, a))^2$ with respect to the Q-network weights
       // Use gradient descent to train Q-network weights
16:    $st \leftarrow st'$
17: **end for**
18: $\epsilon \leftarrow \max(\epsilon \times \delta, \epsilon_{\min})$
     // Perform decay on ϵ
19: **end for**
20: **return** Q

Algorithm 1 DQN Training Algorithm

Following the modeling phase, we design a series of algorithms to implement our runtime scheduling strategy. Algorithm 1 outlines the training process of the DQN, which utilizes the ϵ-greedy algorithm to accelerate the convergence of the deep reinforcement learning network. Additionally, Algorithm 2 is designed to execute a specified action and obtain the corresponding reward.

In Algorithm 1, the input includes the state space S, action space A, number of training episodes E, num of max steps N, and the ϵ decay factor δ. The output is the trained DQN model Q.

Initially, the algorithm randomly initializes all weights of the Q-network and sets the value of ϵ to 1.0 (lines 1–2). During each training episode, the state *st* is randomly initialized under the specified constraints (lines 3–4). The algorithm then performs *N* action steps. At each step, an action is selected based on ϵ: with a probability of ϵ, an action is randomly chosen from the state space, and with a probability of $1-\epsilon$, the action with the highest Q-value is selected (lines 6–10). The chosen action is executed using Algorithm 2, which returns the reward *r* and the new state st' (line 11). The experience tuple (st, a, r, st') is stored in the replay buffer, and a batch of experience tuples is randomly sampled from replay buffer for training (lines 12–13). We adopt a fixed-capacity replay buffer. When the buffer becomes full, a FIFO replacement strategy is used to maintain sample freshness and prevent memory overflow.

The target value *y*, which represents the updated Q-value for action *a* in state *st*, is calculated based on the reward *r*, the discount factor γ, and the maximum expected reward for the next state (line 14). The DQN weights are then updated using gradient descent to minimize the error between the current Q-value and the target *y* (line 15). The state *st* is updated using the selected action (line 16). After completing all *N* action steps in an episode, the value of ϵ is decayed, ensuring that it does not drop below the minimum value $\epsilon_{\min}$ (line 18). Finally, the trained DQN model *Q* is returned (line 20).

In Algorithm 2, the input includes the current state st, the selected action a, and the trained LSTM network, while the outputs are the next state st' and the reward value r. The algorithm first identifies the target machine M_j and operator o_i based on the action *a*, determining where the action will be applied (line 1). Next, the maximum step size $length_{max}$ is calculated based on the remaining slots of the target machine M_j with constraint (9) (line 2). Then, a random number $random_number \in [0,1]$ is generated, and if $random_number < 0.75$ or $length_{max} == 1$, the step size *length* is set to 1; otherwise, *length* is randomly chosen from the range $[2, length_{max}]$ (line 3–8). The step size *length* refers to the magnitude of adjustments made to the number of operator instances on the target machine. For example, a small step size ($length = 1$) indicates an adjustment of only one operator instance (either increasing or decreasing), while a large step size ($length > 1$) represents adjustments involving multiple operator instances (e.g., 2, 3, or more). We assign a probability of 0.75 to small step sizes to constrain the adjustment magnitude of operator instances, thereby maintaining system stability in most cases. Meanwhile, we reserve a probability of 0.25 for large step sizes, enabling the algorithm to quickly adjust the number of operator instances and avoid excessive conservatism.

After that, the action *a* is executed with the selected step size *length*, modifying the number of operator instances

Input: State st, action a, trained LSTM
Output: Next state st', reward r

1: Determine the target machine M_j and operator o_i based on action a.
2: Calculate the maximum step size $length_{max}$ based on the remaining slots of machine M_j with constraint (9)
   // Determine the maximum number of instances that can be modified based on resource constraints
3: Generate a random number $random_number \in [0,1]$.
4: **if** $random_number < 0.75$ **or** $length_{max} == 1$ **then**
5:   $length \leftarrow 1$
   // choose the step size
6: **else**
7:   $length \leftarrow$ randomly choose length $\in [2, length_{max}]$
8: **end if**
9: $st' \leftarrow$ execute action a with $length$
   // Execute the action to modify the operator instances and obtain the next state
10: $Y, Z \leftarrow$ predict using LSTM
11: $r \leftarrow$ calculate reward based on Y, Z by Eq. (22)
12: **return** st', r

Algorithm 2 Executing Action And Getting Reward Algorithm

on the target machine M_j and resulting in the next state st' (line 9). Subsequently, the trained LSTM model is used to predict the system latency Y and system load Z for the new state st' (line 10). Finally, the reward r is computed using Eq. (22), which based on the predicted metrics Y and Z, and the algorithm returns the next state st' and the reward r (line 11–12).

Building on the trained DQN, we propose a scheduling scheme generation algorithm, which combines the ϵ-greedy algorithm and the Softmax algorithm, as shown in Algorithm 3. Additionally, Algorithm 4 illustrates the Softmax action selection method.

In Algorithm 3, the input includes the trained DQN Q, the initial placement state of operator instances st_{init}, the action space A, the maximum decision step length *maxSteps*, the number of episodes E, and the parameter ϵ that controls the trade-off between exploration and exploitation. The output is the set of operator placement solutions P.

Input: Trained DQN model Q, initial state st_{init}, action space A, num of max steps $maxSteps$, num of episodes E, epsilon ε
Output: Solution set P

1: $P \leftarrow \emptyset$
2: **for** each $e \in \{1, \ldots, E\}$ **do**
3: $st \leftarrow st_{init}$
4: $p_{best} \leftarrow$ empty
5: **for** each $t \in \{1, \ldots, maxSteps\}$ **do**
6: **if** $random_number < \varepsilon$ **then**
7: **if** $random_number < \varepsilon/2$ **then**
8: $a \leftarrow$ select an action from A randomly
9: **else**
10: $a \leftarrow$ execute Softmax Action Selection
// Select action using the Softmax policy for diversity
11: **end if**
12: **else**
13: $a \leftarrow argmax_a(st, a')$
// Select the action with the highest Q-value for the current state
14: **end if**
15: Execute action a and obtain the next state st'
16: Calculate current reward $r_{st'}$
17: **if** $r_{st'} > r_{st_{init}}$ **then**
18: $p_{best} \leftarrow st'$
// Update the best solution for this decision
19: **end if**
20: $st \leftarrow st'$
// Update the current state to the new state
21: **end for**
22: Add p_{best} to solution set P
23: **end for**
24: **return** solution set P

Algorithm 3: Scheduling Scheme Generation Algorithm

The algorithm begins by initializing the solution set P as an empty set (line 1). In the outer loop, it iterates over the episodes, initializing the current state st to the initial state st_{init} for each episode e, and setting p_{best} to "empty" to track the best solution found during that episode (lines 2–4).

Within the inner loop, the algorithm iterates over the time steps t to choose actions. If a generated random number is less than ε, the agent enters the exploration phase. Within this phase, if the random number is also less than $\varepsilon/2$, the algorithm selects a random action from the action space A; otherwise, it uses Softmax to select an action (lines 5–11). Conversely, if the random number is greater than or equal to ε, the algorithm chooses the action a that maximizes the Q-value for the current state st (lines 12–14).

After the action a is selected, it is executed, resulting in a new state st'. The reward $r_{st'}$ for transitioning to state st' is then calculated (lines 15–16). If the current reward $r_{st'}$ is greater than the previous reward r_{st}, the algorithm

updates p_{best} to state st' (lines 17–19). Then the current state is updated to the new state st' (lines 21). At the end of the episode, p_{best} is added to the solution set *P*, and finally, the solution set *P* is returned (lines 12–24).

In Algorithm 4, the input includes the trained DQN model *Q*, the action space *A*, the initial state st_{init}, and the temperature coefficient τ. The output is the selected action *a*. The temperature coefficient τ is a user-adjustable parameter that controls the shape of the Softmax distribution. Specifically, a higher value of τ leads to a flatter probability distribution, reducing the differences between action probabilities and thus promoting exploration. Conversely, a lower value of τ results in a more peaked distribution, favoring actions with higher Q-values and thus prioritizing exploitation of existing knowledge.

To strike a balance between exploration and exploitation, we set $\tau = 0.5$ in this implementation. The algorithm computes a probability for each action *a* in the state st_{init}, ensuring that all probabilities lie within the range [0, 1] and sum to 1 across the action space *A*. Finally, an action is selected through random sampling according to the computed Softmax probabilities, ensuring a probabilistic decision-making process guided by the Q values.

Input: Trained DQN model Q, initial state st_{init}, action space $A = \{a_1, a_2, \ldots, a_{|A|}\}$, temperature parameter τ
Output: Next action a

1: $Z \leftarrow 0$
  // Initialize the sum of exponentiated Q-values
2: **for** each action $a_i \in A$ **do**
3:   $\text{exp_Q}[a_i] \leftarrow \exp(Q(st_{\text{init}}, a_i)/\tau)$
  // Compute exponentiated Q-value for action a_i in state st_{init}
4:   $Z \leftarrow Z + \text{exp_Q}[a_i]$
  // Accumulate the sum of exponentiated Q-values
5: **end for**
6: **for** each action $a_i \in A$ **do**
7:   $\text{Softmax_prob}[a_i] \leftarrow \text{exp_Q}[a_i]/Z$
  // Compute Softmax probability for action a_i
8: **end for**
9: $\text{random} \leftarrow \text{Rand}(0, 1)$
  // Generate a random number uniformly distributed in $[0, 1]$
10: $\text{total} \leftarrow 0$
  // Initialize the cumulative probability for action selection
11: **for** each action $a_i \in A$ **do**
12:   $\text{total} \leftarrow \text{total} + \text{Softmax_prob}[a_i]$
  // Accumulate the probability of action a_i
13:   **if** $\text{total} \geq \text{random}$ **then**
  // If the cumulative probability exceeds the random number
14:     **return** a_i
  // Select and return action a_i
15:   **end if**
16: **end for**

Algorithm 4: Softmax Action Selection Algorithm

Table 2 Hardware configuration

Parameter	Master Node	RL Node	Worker Node 1	Worker Node 2	Worker Node 3
Quantity	1	1	3	3	3
vCPU	2GHz, 2 cores	2GHz, 8 cores	2GHz, 1 core	2GHz, 1 core	2GHz, 4 cores
Memory	4G	8G	2G	4G	4G
Disk	40G	40G	40G	40G	40G
Bandwidth	300Mbps	200Mbps	200Mbps	300Mbps	100Mbps

Table 3 Software and versions

Software	Version
Operating System	Ubuntu 20.04.1
Storm	Apache Storm 2.1.0
JDK	1.8
Zookeeper	3.6.3
Python	3.9
Kafka	2.1.2

Table 4 Experimental parameters

Parameter	Value
Learning Rate	0.001
Buffer Size	10,000
Experience Replay Buffer	64
Target Update Frequency	10 epochs
Training Episodes	1,000
Discount Factor γ	0.99
Dropout	0.5

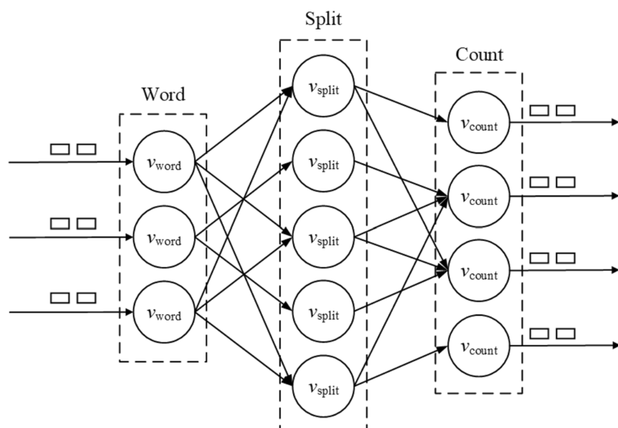

Fig. 6 A sample of WordCount operator instance topology

5 Performance evaluation

We evaluate the performance of Pa-Stream by comparing it with the widely used R-Storm [20] and the heuristic scheduling strategy SP-Ant [6] under two distinct data stream scenarios. R-Storm and SP-Ant are well-established Storm scheduling baselines [21, 22], representing resource balancing and task dependency optimization, respectively. They serve as suitable references to highlight the performance improvements achieved by Pa-Stream in dynamic scenarios.

Furthermore, we validate the accuracy of the proposed predictive algorithm. Although this paper does not separately present training convergence curves or sample efficiency comparisons, Pa-Stream inherits the convergence stability of DQN and improves sample efficiency through its prediction component. A more systematic analysis of convergence and sample efficiency will be conducted in future work.

5.1 Experimental setup

The experiments are conducted in a cluster deployed on the Alibaba Cloud computing platform, which consists of 11 machines. Among them, one master node hosts Nimbus and the Storm UI, 9 worker nodes handle job computation, and the remaining machine is designated for the DQN agent's calculations and online training of the LSTM network. To evaluate the algorithm's performance in a heterogeneous environment, the 9 worker nodes are selected from three different types of machines. The hardware configurations used in the experiments are detailed in Table 2, and the Software versions are listed in Table 3.

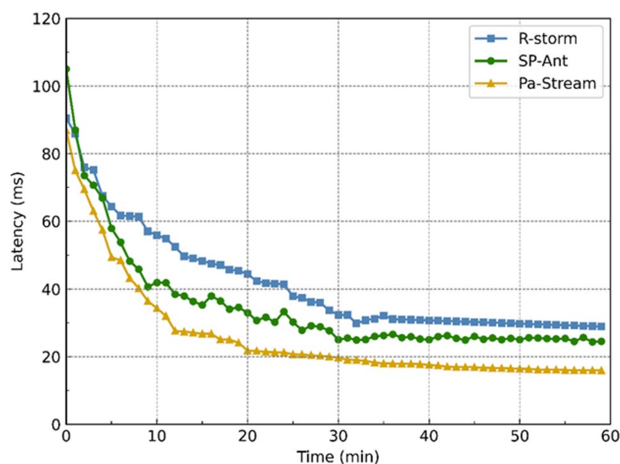

Fig. 7 Latency comparison of Pa-Stream with R-Storm and SP-Ant under steady data streams

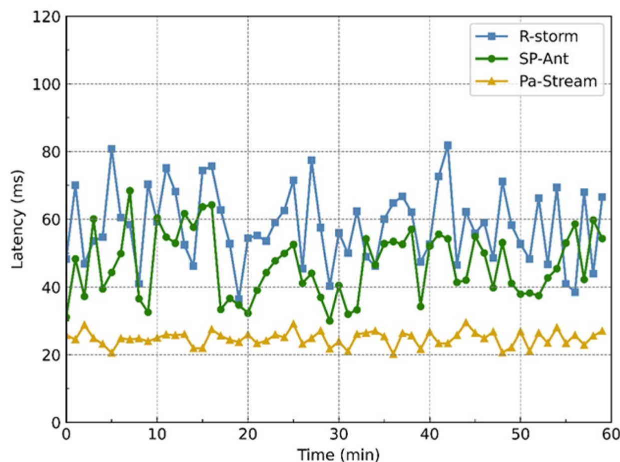

Fig. 8 Latency comparison of Pa-Stream with R-Storm and SP-Ant under fluctuating data streams

In our experiments, a dedicated node running the DQN agent and LSTM online inference/training is sufficient to support real-time scheduling decisions. For larger-scale topologies, however, computational and communication overheads may increase, in which case engineering optimizations such as model lightweighting or inference–training separation can be employed to maintain real-time performance.

The experiments use the WordCount application, a widely adopted benchmark in the stream processing community that has been extensively used in prior research [23–26]. The core scheduling mechanism of Pa-Stream is independent of specific application logic and can be applied to more

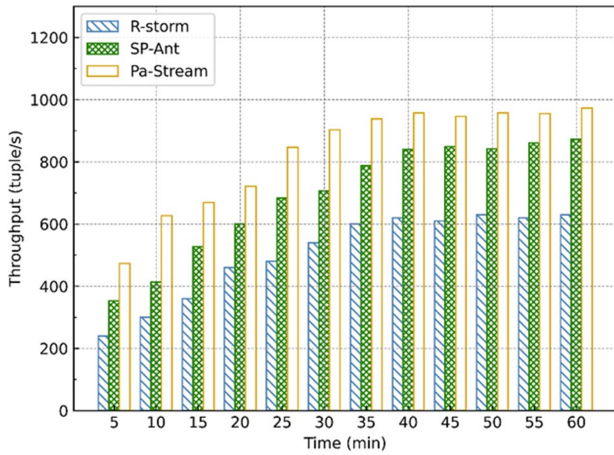

Fig. 9 Throughput comparison of Pa-Stream with R-Storm and SP-Ant under steady data streams

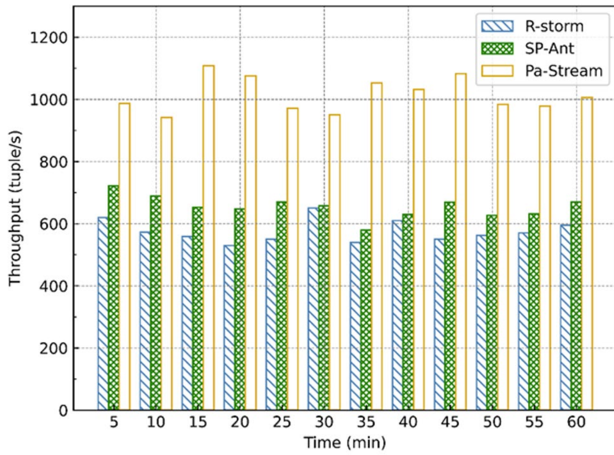

Fig. 10 Throughput comparison of Pa-Stream with R-Storm and SP-Ant under fluctuating data streams

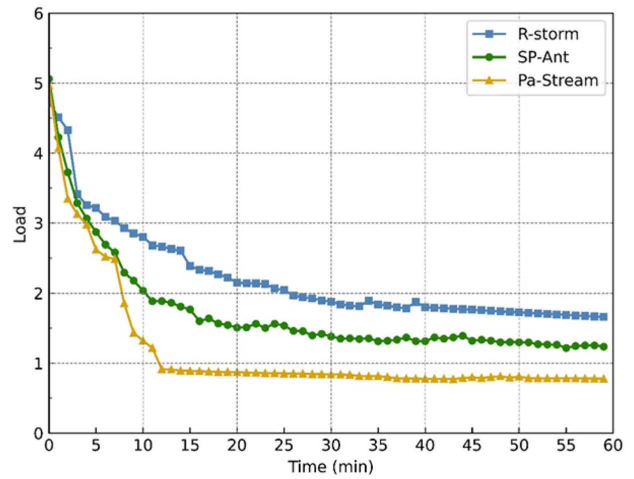

Fig. 11 Load comparison of Pa-Stream with R-Storm and SP-Ant under steady data streams

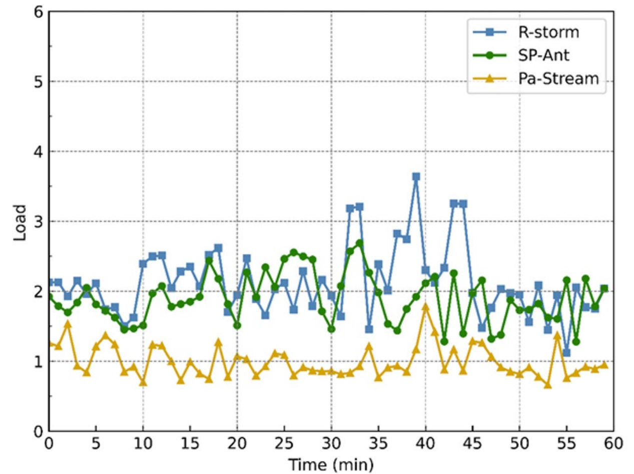

Fig. 12 Load comparison of Pa-Stream with R-Storm and SP-Ant under fluctuating data streams

complex topologies such as fraud detection or IoT pipelines, which we plan to explore in future work (see Section 7).

Due to space limitations, this section only presents the experimental data of the aforementioned three strategies on the WordCount application. Fig. 6 presents a sample of the WordCount operator instance topology, and experimental parameters are detailed in Table 4.

We use the Twitter 2022 user behavior dataset [27] as data source during our experimental evaluation. 70% of the data is extracted for training the prediction network, while the remaining 30% is extracted for testing in the experiments. The data extraction criteria ensure that the data stream speed matches the experimental preset and that the scale is within the carrying capacity of the experimental cluster. The experimental data size is 30GB, with two data stream scenarios: a steady stream at 1000 tuples/s and a fluctuating stream varying randomly between 800 tuples/s and 1200 tuples/s.

In the fluctuating stream scenario, the input rate is sampled from a uniform distribution between 800 and 1200 tuples/s with a fixed random seed. This ensures that Pa-Stream and all baseline methods are evaluated under the same workload sequence, guaranteeing fairness and reproducibility. Each experimental scenario is independently executed three times, and the figures report the mean values from these runs. Due to experimental cost constraints, error bars are not displayed; however, the performance trends remain consistent across repeated experiments, indicating stable system behavior.

5.2 System latency

System latency is defined as the average end-to-end latency of tuples, measured from the arrival of tuples at the application instance to the completion of their computation. This

metric reflects the average duration for which tuples are processed within the application instance, with lower latency indicating stronger real-time processing capabilities.

Figs. 7 and 8 present the system latency for Pa-Stream, R-Storm, and SP-Ant under steady and fluctuating data streams, respectively. Under steady streams, R-Storm exhibits a latency of approximately 41.64 milliseconds, while SP-Ant achieves 34.82 milliseconds. In comparison, Pa-Stream significantly reduces latency to 26.17 milliseconds, representing reductions of 37.17% and 24.86% compared to R-Storm and SP-Ant, respectively. Under fluctuating streams, R-Storm's latency increases to about 57.97 milliseconds, and SP-Ant achieves 46.61 milliseconds. Pa-Stream outperforms both, with a latency of 24.79 milliseconds, which is 57.24% lower than R-Storm and 46.81% lower than SP-Ant. The latency reduction observed in Fig. 8 primarily stems from the predictive capability of the LSTM module, which avoids scheduling delays under transient workload fluctuations.

These results demonstrate that the proposed Pa-Stream improves system latency under both steady and fluctuating streams, maintaining stable performance even in the presence of substantial stream variations. The above experimental results demonstrate that: whether under stable or fluctuating data rates, Pa-Stream's rescheduling remains stable. Under the stable stream rate, only a few initial migrations occur, while under the fluctuating stream rate, moderate migrations are triggered in response to load variations, but no frequent oscillations are observed.

5.3 System throughput

System throughput is defined as the number of tuples processed per second from the data source, reflecting the system's data processing capability. Higher throughput indicates stronger data processing performance.

Figures 9 and 10 illustrate the throughput variations for Pa-Stream, R-Storm, and SP-Ant under steady and fluctuating data streams, respectively. Under steady streams, the average throughput for R-Storm is approximately 507.50 tuples/s, while SP-Ant achieves an average throughput of 694.67 tuples/s. In comparison, Pa-Stream's average throughput is 830.50 tuples/s, representing an increase of about 46.56% over R-Storm and 19.55% over SP-Ant.

Under fluctuating streams, R-Storm's average throughput is approximately 575.75 tuples/s, while SP-Ant achieves 653.75 tuples/s. Pa-Stream outperforms both, with an average throughput of 1014.33 tuples/s, which is about 76.18% higher than R-Storm and 55.16% higher than SP-Ant. The throughput improvement in Fig. 10 results from adaptive decisions made by the DQN agent, which balance operator migration and resource utilization.

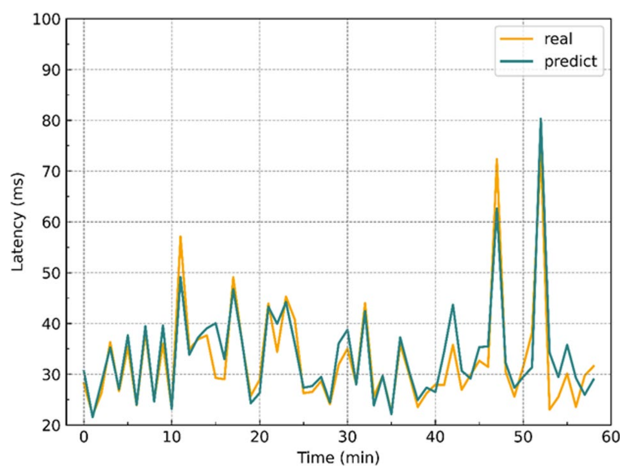

Fig. 13 Comparison of LSTM network predicted latency with actual values

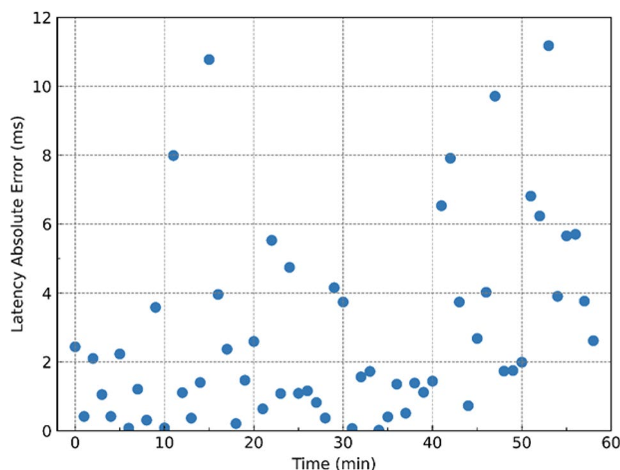

Fig. 14 Error analysis of LSTM network predicted latency compared to actual values

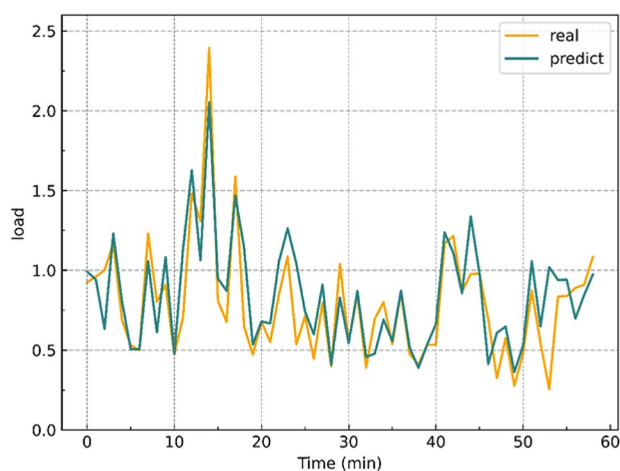

Fig. 15 Comparison of LSTM network predicted system load with actual values

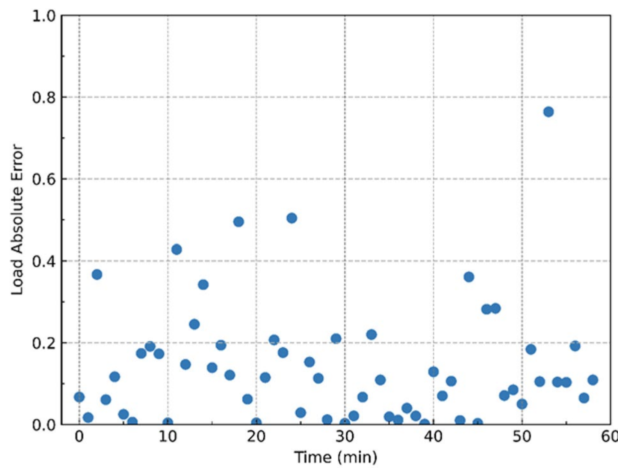

Fig. 16 Error analysis of LSTM network predicted system load compared to actual values

Table 5 Comparison of Pa-Stream and related work

Algorithm	Predictive	Learnability	Robustness	Notes
R-Storm [20]	×	None	Low	3D Bin Packing
I-Scheduler [7]	×	None	Low	Graph Partitioning
SP-Ant [6]	×	None	Medium	Ant Colony Algorithm
MT-Scheduler [29]	×	None	Medium	Dynamic Programming
ER-Storm [30]	×	Medium	Medium	Q-Learning
Pa-Stream (Ours)	✓	High	High	Deep Reinforcement Learning

These results demonstrate that Pa-Stream achieves substantial improvements in system throughput under both steady and fluctuating streams, with particularly notable gains under fluctuating conditions.

5.4 System load

System load is defined as the capacity utilization of operators, representing the proportion of time that operators are actively working within a given time frame. Excessive load can lead to increased processing latency and reduced system performance, whereas insufficient load indicates resource under-utilization within the cluster. An efficiently operating stream computing system should ideally maintain a load value between 0.8 and 1.0.

Figures 11 and 12 illustrate the load variations for Pa-Stream, R-Storm, and SP-Ant under steady and fluctuating data streams, respectively. Under steady streams, R-Storm's load averages around 2.23, while SP-Ant achieves an average load of 1.73. In contrast, Pa-Stream's load is approximately 1.19, representing a reduction of about 46.55% compared to R-Storm and about 31.25% compared to SP-Ant.

Under steady streams, Pa-Stream consistently maintains a stable load close to 0.8, effectively avoiding excessive system load and data backlog while preventing resource waste. In contrast, both R-Storm and SP-Ant exhibit significant load fluctuations under rapid data stream changes, leading to unstable performance, a critical factor behind increased latency and decreased throughput. Pa-Stream, however, exhibits only minor load fluctuations even under severe variations in data stream rates, maintaining a load close to 1. This balance ensures that the system avoids data backlog due to excessive load and prevents resource underutilization caused by insufficient load.

Under fluctuating streams, R-Storm's average load is approximately 2.12, and SP-Ant's average load is around 1.90. Pa-Stream achieves a significantly lower and more stable average load of 0.99, reflecting a decrease of about 52.91% compared to R-Storm and 47.47% compared to SP-Ant. The stable performance in Fig. 12 reflects the reward design, which penalizes frequent reconfigurations, thereby minimizing thrashing and enhancing stability.

The experimental results demonstrate that Pa-Stream maintains stable rescheduling behaviour under both steady and fluctuating data rates. Under a stable stream rate, as shown in Fig 7 and 11, due to only a few initial migrations occur, Pa-Stream exhibits relatively stable latency and resource utilization, without any significant fluctuations. While under fluctuating stream rates, as shown in Fig. 8 and 12, Pa-Stream exhibits smoother latency variations and more stable resource load compared to R-Storm and SP-Ant. This indicates that the system performs only limited operator reconfigurations during fluctuating workloads, effectively avoiding thrashing and maintaining stable scheduling behaviour.

5.5 Prediction accuracy

To assess the accuracy of the predictive algorithm for system performance forecasting, a portion of the dataset is used for training, and random samples from the remaining data are used for prediction. The prediction period is set to one minute, using data stream fluctuations from the first four minutes to predict system performance changes in the fifth minute. The predicted values for latency and load are compared to the actual values, and the MAE, as defined in Eq. (13), is used to measure prediction accuracy.

System Latency Prediction. Figs. 13 and 14 compare the predicted latency from the predictive algorithm with

actual latency values and analyze the MAE results. Fig. 13 indicates that the predicted latency trends closely match the actual values. The MAE analysis in Fig. 14 reveals an average error of approximately 2.809 milliseconds.

System Load Prediction. Figs. 15 and 16 compare the predicted load values with actual load values, and provide an MAE analysis. Fig. 15 shows that the predicted and actual load trends align closely. Fig. 16 shows an average MAE of approximately 0.144.

This experimental results indicate that the trained LSTM network exhibits low prediction errors for stream computing system performance metrics. This demonstrates the network's capability to predict system performance changes in real-time under varying data stream conditions.

5.6 Generalizability to complex topologies

Although our experiments focused on the relatively simple WordCount application, the insights gained are applicable to more complex real-world scenarios. In complex topologies, the core challenge remains the dynamic adjustment of operator placement schemes to accommodate fluctuations in data stream rates. The LSTM model and the DQN model are the main components of Pa-Stream. The LSTM model captures the relationship between the current system state and future system performance. This enables the model to effectively predict future outcomes, regardless of the complexity of the underlying topology. Similarly, the DQN model is not dependent on specific application structures. It generates scheduling schemes by using the LSTM model as the decision-making environment, making it suitable for both simple and complex topologies. Thus, Pa-Stream's core mechanisms are broadly applicable across diverse stream processing applications.

6 Related work

Learning-based algorithms directly optimize performance objectives, making them an increasingly popular research area in data analysis and processing [28]. We review and analyze existing studies on operator scheduling for distributed stream computing, identifying achievements and key limitations. Furthermore, we examine the state of research on data stream pattern recognition algorithms.

6.1 Operator scheduling for stream computing

Significant progress has been made in developing operator scheduling strategies for stream computing. The most popular approaches include model-based scheduling, heuristic algorithm-based scheduling, and reinforcement learning-based scheduling. Table 5 presents a comparison between existing scheduling algorithms and the proposed Pa-Stream.

In terms of static and heuristic-based schedulers, R-Storm [20] minimizes inter-node communication by precomputing placement plans but depends heavily on accurate user-provided resource requirements. I-Scheduler [7] leverages K-way graph partitioning to reduce communication costs but determines partition thresholds empirically. SP-Ant [6] combines heuristic and meta-heuristic algorithms (ant colony and bin-packing), yet its iterative optimization results in slow convergence. Similarly, MT-Scheduler [29] applies dynamic programming to map tasks based on static resource descriptions, but it lacks adaptability to runtime workload variations. Overall, these approaches assume relatively stable environments and do not adapt to dynamic workload fluctuations.

In terms of Learning-based or adaptive schedulers attempt to address dynamic workload changes using reinforcement learning or search-based algorithms. ER-Storm [30] employs tabular Q-learning combined with replication and migration (RSR) mechanisms to handle elasticity, but its reactive design and slow Q-learning convergence limit responsiveness in highly dynamic scenarios. The MCTS (Monte Carlo tree search)-based [31] approach achieves multi-objective optimization via stochastic sampling but suffers from slow convergence and limited precision due to sampling noise. The TBVI ((Trajectory-Based Value Iteration))-based method [32] improves sampling efficiency over traditional Q-learning by function approximation but simplifies the action space to single-operator scaling, leading to longer convergence trajectories and higher reconfiguration costs.

In terms of industry schedulers, Apache Beam and Google Dataflow provide unified programming abstractions for stream and batch data processing but rely on underlying cluster resource managers for runtime scheduling [33]. Industrial schedulers such as Kubernetes and YARN primarily focus on resource allocation and task isolation, rather than proactive scheduling for latency or throughput optimization [34].

Current distributed stream computing task schedulers aim to minimize inter-node communication, reduce latency, and improve throughput, but many limitations remain as discussed below:

(1) Model-based algorithms: When solving NP-hard problems, these methods may overlook useful information, such as data distribution during algorithm design, leading to suboptimal solutions. They lack adaptability, often performing poorly when applied to workloads different from those they were designed for. Their

development is time-consuming, requiring significant effort from developers to test and fine-tune numerous rules empirically. They struggle with heterogeneous clusters, as most schedulers designed for such environments reply on heuristic algorithms.

(2) Heuristic algorithms: While heuristic algorithms can produce optimal or near-optimal solutions for single-operator scheduling, they depend on iterative processes, which makes it difficult to obtain satisfactory solutions in the early stages of computation. Repeated iterations lead to slower solving speeds, restricting their applicability to offline scheduling. These algorithms are unable to reconfigure flexibly in response to dynamic changes in real-time data streams, such as input rate fluctuations or data skew, potentially leading to node overload. When clusters experience failures or scale up, heuristic-based offline schedulers struggle to adjust operator placement dynamically.

To address these challenges, runtime strategies are needed. These strategies allow dynamic operator migration to adapt to changes in both the cluster and data streams, ensuring flexibility, efficiency, and resilience in real-time stream computing environments. Unlike reactive schedulers such as ER-Storm, Pa-Stream integrates LSTM-based prediction with DQN-driven adaptive scheduling to proactively anticipate and mitigate the impact of fluctuating workloads. This design enables Pa-Stream to achieve more stable and forward-looking scheduling decisions.

6.2 Data stream pattern recognition

A data stream is defined as “an unbounded sequence of multidimensional, sporadic, and transient observations available over time” [35]. A data stream pattern refers to recurring, identifiable patterns or regularities within the data stream. These patterns help characterize the data stream and predict future changes, enabling informed decision-making. Trends in data streams over time can be periodic (e.g., seasonal variations or daily fluctuations), or non-periodic (e.g., long-term trends or sudden events). These trends can be identified and modeled using techniques such as time series analysis and statistical analysis.

Recent advances in stream learning and concept drift adaptation have introduced several algorithms that improve the accuracy and adaptability of models in evolving data environments. Representative approaches such as Adaptive Random Forest (ARF) [36], Streaming Random Patches (SRP) [37], and CS-ARF [38] employ ensemble-based strategies to maintain predictive accuracy under drift. ARF introduces adaptive resampling and operator adjustment to handle different drift types, while SRP combines random subspace and online bagging for fast and diverse ensemble learning. CS-ARF further integrates compressed sensing for dimensionality reduction, enhancing performance in high-dimensional streams.

Other studies have explored neural and hybrid stream learners, such as the Randomized Neural Network (RNN) [39] and the hybrid Hoeffding-tree-based methods [40], which use random feature filters and GPU acceleration to balance accuracy and efficiency. Frameworks such as River [41] provide unified toolkits for online learning and continual model evaluation, integrating algorithms from Creme and scikit-multiflow. In addition, Bahri et al. [42] proposed a time-weighted KNN method that adapts to recent data trends using sliding-window mechanisms.

These methods have demonstrated effectiveness in addressing data classification and concept drift in dynamic data streams. However, they tend to overlook the complex relationship between data stream fluctuations, operator placement strategies, and key system performance metrics such as throughput, latency, and resource utilization. This limitation hinders their ability to provide early-stage performance predictions, which are essential for optimizing resource allocation and task scheduling in real-time stream processing systems. Moreover, most algorithms are not designed to adapt to long-term evolutions in data stream patterns, reducing their robustness in continuously changing environments.

7 Conclusion and future work

This paper analyzes the impact of data stream variations on system performance and investigates scheduling challenges in distributed stream computing systems. It introduces a data stream pattern-aware scheduling strategy, Pa-Stream, for stream computing systems. Experimental comparisons show that Pa-Stream significantly enhances throughput, reduces latency, and optimizes resource utilization when processing fluctuating data streams, outperforming R-Storm and SP-Ant.

While Pa-Stream effectively addresses the rescheduling challenges of stateless operators in heterogeneous clusters, its current implementation is limited to relatively simple scenarios, such as the WordCount topology. Future work could explore its performance on more complex applications. Specifically, future research will focus on the following aspects.

(1) State management for stateful operators. Future work will address the state management challenges associated with stateful operators during runtime scheduling, extending the current Pa-Stream strategy. This includes designing algorithms for state transitions, such as mechanisms for

state backup, restoration, and consistency maintenance during runtime scheduling.

(2) Evaluation on complex topologies. Future work will evaluate Pa-Stream's performance on complex topologies, such as multi-stage ETL pipelines, real-time fraud detection systems, and large-scale graph processing tasks. These evaluations will provide deeper insights into the Pa-Stream's effectiveness under diverse and challenging scenarios.

Acknowledgements This work is supported by the National Natural Science Foundation of China under Grant No.62372419; the Fundamental Research Funds for the Central Universities, China under Grant No.265QZ2021001. The authors would like to thank YongKang Dang for his revisions to this paper.

Author contributions Dawei Sun: Conceptualization, Methodology, Validation, Writing - review & editing, Funding acquisition. Yinuo Fan: Validation, Investigation, Writing - original draft. Ning Zhang: Investigation, Data curation, Writing - original draft. Shang Gao: Formal analysis, Investigation, Writing - review & editing. Jianguo Yu: Investigation, Data curation, Writing - original draft. Rajkumar Buyya: Methodology, Writing -review & editing, Supervision, Funding acquisition.

Data availability No datasets were generated or analysed during the current study.

Declarations

Competing interests The authors declare no competing interests.

References

1. Fragkoulis, M., Carbone, P., Kalavri, V., Katsifodimos, A.: A survey on the evolution of stream processing systems. The VLDB Journal **33**(2), 507–541 (2024)
2. Hadian, H., Sharifi, M.: Gt-scheduler: a hybrid graph-partitioning and tabu-search based task scheduler for distributed data stream processing systems. Cluster Computing **27**(5), 5815–5832 (2024)
3. Hu, B., Yang, X., Zhao, M.: Online energy-efficient scheduling of dag tasks on heterogeneous embedded platforms. Journal of Systems Architecture **140**, 102894 (2023)
4. Tang, B., Han, H., Yang, Q., Xu, W.: Operator placement for data stream processing based on publisher/subscriber in hybrid cloud-fog-edge infrastructure. Cluster Computing **27**(3), 2741–2759 (2024)
5. Tan, J., Tang, Z., Cai, W., Tan, W.J., Xiao, X., Zhang, J., Gao, Y., Li, K.: A cost-aware operator migration approach for distributed stream processing system. IEEE Trans. Cloud Comput. (2025). https://doi.org/10.1109/TCC.2025.3538512
6. Farrokh, M., Hadian, H., Sharifi, M., Jafari, A.: Sp-ant: An ant colony optimization based operator scheduler for high performance distributed stream processing on heterogeneous clusters. Expert Systems with Applications **191**, 116322 (2022)
7. Eskandari, L., Mair, J., Huang, Z., Eyers, D.: I-scheduler: Iterative scheduling for distributed stream processing systems. Future Generation Computer Systems **117**, 219–233 (2021)
8. Sun, D., Cui, Y., Wu, M., Gao, S., Buyya, R.: An energy efficient and runtime-aware framework for distributed stream computing systems. Future Generation Computer Systems **136**, 252–269 (2022)
9. Eskandari, L., Mair, J., Huang, Z., Eyers, D.: T3-scheduler: A topology and traffic aware two-level scheduler for stream processing systems in a heterogeneous cluster. Future Generation Computer Systems **89**, 617–632 (2018)
10. Muhammad, A., Aleem, M., Islam, M.A.: Top-storm: A topology-based resource-aware scheduler for stream processing engine. Cluster Computing **24**, 417–431 (2021)
11. Li, B., Sun, D., Chau, V.L., Buyya, R.: A topology-aware scheduling strategy for distributed stream computing system. In: Broadband Communications, Networks, and Systems: 12th EAI International Conference, BROADNETS 2021, Virtual Event, October 28–29, 2021, Proceedings 12, pp. 132–147 (2022). Springer
12. Huang, X., Shao, Z., Yang, Y.: Potus: Predictive online tuple scheduling for data stream processing systems. IEEE Transactions on Cloud Computing **10**(4), 2863–2875 (2020)
13. Chang, S., Sun, J., Hao, Z., Deng, Q., Guan, N.: Computing exact wcrt for typed dag tasks on heterogeneous multi-core processors. Journal of Systems Architecture **124**, 102385 (2022)
14. Li, F., Bi, R., Wang, J., Sun, J., Sun, Z., Tan, G., Chen, M.: Vpss: A dag scheduling heuristic with improved response time bound. Journal of Systems Architecture **148**, 103084 (2024)
15. Apache: Spark (2025). http://spark.apache.org/ Accessed 2025-03-12
16. Apache: Flink (2025). http://flink.apache.org/ Accessed 2025-03-12
17. Mohiuddin, I., Almogren, A., Al Qurishi, M., Hassan, M.M., Al Rassan, I., Fortino, G.: Secure distributed adaptive bin packing algorithm for cloud storage. Future Generation Computer Systems **90**, 307–316 (2019)
18. Tan, F., Yan, P., Guan, X.: Deep reinforcement learning: From q-learning to deep q-learning. In: Neural Information Processing: 24th International Conference, ICONIP 2017, Guangzhou, China, November 14–18, 2017, Proceedings, Part IV 24, pp. 475–483 (2017). Springer
19. Ernst, D., Louette, A.: Introduction to reinforcement learning. Feuerriegel, S., Hartmann, J., Janiesch, C., and Zschech, P, 111–126 (2024)
20. Peng, B., Hosseini, M., Hong, Z., Farivar, R., Campbell, R.: R-storm: Resource-aware scheduling in storm. In: Proceedings of the 16th Annual Middleware Conference, pp. 149–161 (2015)
21. Kang, P., Khan, S.U., Zhou, X., Lama, P.: High-throughput real-time edge stream processing with topology-aware resource matching. In: 2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid), pp. 385–394 (2024). IEEE
22. Ecker, R., Karagiannis, V., Sober, M., Schulte, S.: Latency-aware placement of stream processing operators in modern-day stream processing frameworks. Journal of Parallel and Distributed Computing **199**, 105041 (2025)
23. Kalavri, V., Liagouris, J., Hoffmann, M., Dimitrova, D., Forshaw, M., Roscoe, T.: Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pp. 783–798 (2018)
24. Ching, C.-W., Chen, X., Kim, C., Wang, T., Chen, D., Da Silva, D., Hu, L.: Agiledart: An agile and scalable edge stream processing engine. IEEE Transactions on Mobile Computing (2025). https://doi.org/10.1109/TMC.2025.3526143
25. Maroulis, S., Zacheilas, N., Kalogeraki, V.: A holistic energy-efficient real-time scheduler for mixed stream and batch processing workloads. IEEE Trans. Parallel Distrib. Syst. **30**(12), 2624–2635 (2019)

26. Zapridou, E., Mytilinis, I., Ailamaki, A.: Dalton: Learned partitioning for distributed data streams. Proceedings of the VLDB Endowment **16**(3), 491–504 (2022)
27. Twitter, Inc.: Twitter Developer API Documentation (2022). https://developer.twitter.com/en/docs Accessed 2022-12-31
28. Cai, Q., Cui, C., Xiong, Y., Wang, W., Xie, Z., Zhang, M.: A survey on deep reinforcement learning for data processing and analytics. IEEE Trans. Knowl. Data Eng. **35**(5), 4446–4465 (2022)
29. Al-Sinayyid, A., Zhu, M.: Job scheduler for streaming applications in heterogeneous distributed processing systems. J. Supercomput. **76**(12), 9609–9628 (2020)
30. Hadian, H., Farrokh, M., Sharifi, M., Jafari, A.: An elastic and traffic-aware scheduler for distributed data stream processing in heterogeneous clusters. J. Supercomput. **79**(1), 461–498 (2023)
31. Silva Veith, A., De Souza, F.R., Assuncao, M.D., Lefèvre, L., Dos Anjos, J.C.S.: Multi-objective reinforcement learning for reconfiguring data stream analytics on edge computing. In: Proceedings of the 48th International Conference on Parallel Processing, pp. 1–10 (2019)
32. Russo, G.R., Cardellini, V., Presti, F.L.: Reinforcement learning based policies for elastic stream processing on heterogeneous resources. In: Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems, pp. 31–42 (2019)
33. Li, S., Gerver, P., MacMillan, J., Debrunner, D., Marshall, W., Wu, K.-L.: Challenges and experiences in building an efficient apache beam runner for ibm streams. Proceedings of the VLDB Endowment **11**(12), 1742–1754 (2018)
34. Li, W., Wang, Y., Ma, W., Wang, L., Lv, D., Liu, H.: Containerized scheduling method based on kubernetes and yarn in big data scenarios. In: Proceedings of the 11th International Conference on Computer Engineering and Networks, pp. 1339–1350 (2021). Springer
35. Bahri, M., Bifet, A., Gama, J., Gomes, H.M., Maniu, S.: Data stream analysis: Foundations, major tasks and tools. WIREs Data Mining and Knowledge Discovery **11**(3), 1405 (2021)
36. Gomes, H.M., Bifet, A., Read, J., Barddal, J.P., Enembreck, F., Pfharinger, B., Holmes, G., Abdessalem, T.: Adaptive random forests for evolving data stream classification. Mach. Learn. **106**, 1469–1495 (2017)
37. Gomes, H.M., Read, J., Bifet, A.: Streaming random patches for evolving data stream classification. In: 2019 IEEE International Conference on Data Mining (ICDM), pp. 240–249 (2019). IEEE
38. Bahri, M., Gomes, H.M., Bifet, A., Maniu, S.: Cs-arf: compressed adaptive random forests for evolving data stream classification. In: 2020 International Joint Conference on Neural Networks (IJCNN), pp. 1–8 (2020). IEEE
39. Pratama, M., Angelov, P.P., Lu, J., Lughofer, E., Seera, M., Lim, C.P.: A randomized neural network for data streams. In: 2017 International Joint Conference on Neural Networks (IJCNN), pp. 3423–3430 (2017). IEEE
40. Marrón, D., Read, J., Bifet, A., Navarro, N.: Data stream classification using random feature functions and novel method combinations. Journal of Systems and Software **127**, 195–204 (2017)
41. Montiel, J., Halford, M., Mastelini, S.M., Bolmier, G., Sourty, R., Vaysse, R., Zouitine, A., Gomes, H.M., Read, J., Abdessalem, T., et al.: River: machine learning for streaming data in python. J. Mach. Learn. Res. **22**(110), 1–8 (2021)
42. Bahri, M.: Effective weighted k-nearest neighbors for dynamic data streams. In: 2022 IEEE International Conference on Big Data (Big Data), pp. 3341–3347 (2022). IEEE

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.