The Journal of Supercomputing (2024) 80:16546–16573
https://doi.org/10.1007/s11227-024-06085-x

TimeLink: enabling dynamic runtime prediction for Flink iterative jobs

Xiaofei Yue[1] · Qingyang Ding[2] · Jianming Zhu[3] · Yanbing Ding[4]

Accepted: 18 March 2024 / Published online: 13 April 2024

Abstract
With the increasing growth of data scale and computing complexity, Flink, a novel distributed computing system, has been applied in various scenarios (e.g., machine learning) due to its excellent iterative nature. Predicting the runtime of Flink iterative jobs is critical to optimizing their performance. However, existing offline works generally ignore relevant runtime information, such as cluster state variations and inter-iteration dependencies, resulting in high actual prediction errors. Online methods, on the other hand, have a non-negligible time overhead. In light of this, we propose *TimeLink*, a dynamic runtime prediction algorithm for Flink iterative jobs. Its key idea consists of three stages: (1) *TimeLink* incorporates both offline and online execution features during runtime to measure the fine-grained similarity of iterative jobs, (2) it matches historical jobs with similar performance consumption to the current running iterative job in real time, and (3) its remaining runtime is predicted by combining the continuity of runtime bias between completed supersteps of matched jobs and the current iterative job. We implement *TimeLink* and evaluate it using realistic iterative workloads. The experimental results show that *TimeLink* exhibits relative average prediction errors of 5.91–12.86%. Moreover, it outperforms existing solutions with an improvement of over 6.24% in prediction accuracy.

Keywords Iterative job · Runtime prediction · Fine-grained similarity · Job matching · Flink system

1 Introduction

Iterative computing poses the essential foundation for machine learning (ML) [18] and graph processing [16], serving as a critical enabler in numerous domains such as artificial intelligence, social networks, and data mining. As the volume of data and computing complexity continue to surge, distributed iterative computing has emerged as a powerful driver of productivity improvement [6]. Flink [5], a

Extended author information available on the last page of the article

cutting-edge distributed system renowned for its robust support of iterative computing, has gained immense popularity in recent years. Leading Internet enterprises, such as Amazon and Twitter, rely on Flink for performing extensive iterative analytics atop vast datasets (e.g., user and system session logs).

In practice, these batch processing jobs generally require adherence to the service level objectives (SLO) in terms of runtime; otherwise, the results obtained may be outdated [26]. User profiles (*i.e., Flink-conf.yaml*) that lack expert guidance are susceptible to mismatches between resource configuration and SLO demand. For instance, resource under-provisioning directly offends SLOs, whereas defensive over-provisioning wastes a significant amount of resources (due to early completion). This makes the system unable to guarantee continuous resource availability, which in turn affects the completion of other colocated jobs. It is reported that as much as 60% of analytics applications in operational clusters are recurring on various datasets [15], some of which are highly repetitive [1, 9]. Estimating the runtime of jobs in advance to guide their resource configurations has received widespread attention and recognition [13, 17, 31]. However, the majority of analytics applications operate for brief durations (e.g., 80% of jobs in Google's production cluster run for less than 10 min [23]), and timely and accurate runtime prediction becomes crucial for downstream optimization methods.

As for an iterative job, its runtime is influenced by many factors, including program parameters, input data sizes, cluster status, and inter-iteration dependencies. Accurately predicting the runtime of these jobs is a challenging task. Some studies [3, 11, 22, 28, 30] delve inside iterative jobs and propose fine-grained methods for job execution time prediction. They generally fit runtime functions based on key features (e.g., shuffling, parallelism, and resource allocation) obtained from sampling execution or job history. Several works [2, 10, 12, 32, 33] leverage recent execution logs to train ML models for black-box prediction of end-to-end job completion times. However, most of them are offline algorithms that lack real-time feedback and do not consider dynamic factors, such as changes in the cluster state, which may affect their performance. In addition, other works [12, 34] propose online prediction methods, which incorporate runtime features to train models, by utilizing real-time data from previous executions. However, online training is generally quite time-consuming, even with incremental techniques. To address these limitations, there is a growing need to adapt to dynamic cluster environments and provide accurate and timely feedback on the runtime of iterative jobs. Fortunately, we observe that certain features that affect the performance of iterative jobs can be obtained offline (e.g., resource allocation and input data volume), while others (e.g., resource utilization and convergence) need to be monitored at runtime. Thus, we are eager to incorporate the advantages of both offline and online prediction to provide accurate runtime prediction with low latency.

In this paper, we present ***TimeLink***, a novel dynamic runtime prediction algorithm for Flink iterative jobs. Unlike existing offline or online prediction algorithms, *TimeLink* commences upon receipt of a user's prediction request. It constructs a fine-grained job similarity model utilizing a combination of offline job features captured via early-stage sampling execution, and online features continuously collected during the execution of iterative jobs. Based on this model, *TimeLink* then performs real-time

matching to identify historical jobs that exhibit similar performance consumption as the current iterative jobs. Finally, the matched jobs are leveraged to dynamically predict the remaining runtime of the current iterative job.

Contributions: To the best of our knowledge, we are the first to consider the dynamic runtime prediction for Flink iterative jobs. This paper makes three main contributions in the following aspects:

- We present *TimeLink*, a dynamic prediction algorithm that offers real-time feedback on the remaining runtime of a Flink iterative job. It uses a similarity-based iterative job-matching algorithm to filter out historical jobs with similar performance consumption to the current job, and then predicts its remaining runtime based on the continuity of runtime bias within matched jobs.
- Based on the execution structure and characteristics of Flink iterative jobs, a combination of static and dynamic fine-grained metrics is carefully designed to measure the similarity in job performance consumption. Further, we propose evaluation criteria core to the similarity quality function to verify the validity of these similarities for runtime prediction.
- We develop a prototype system of *TimeLink* and evaluate it by executing realistic iterative workloads. The experimental results reveal that *TimeLink* exhibits relative average errors of 5.91% and 12.86% in the optimal and worst cases, respectively. Moreover, it outperforms state-of-the-art solutions in terms of the prediction accuracy improvement of over 6.24%.

Organization: The remaining sections are organized as follows. Section 2 provides a summary of the background knowledge and current related work is summarized and analyzed. Section 3 shows the similarity-based iterative job-matching algorithm, and further proposes the dynamic runtime prediction algorithm. The implementation of the prototype system for *TimeLink* is described in Sect. 4. Section 5 evaluates the proposed approach by conducting comparison experiments on multiple datasets. Finally, conclusions and future research are addressed in Sect. 6.

2 Background and related work

In this section, we provide a brief overview of the background of Flink iterative computing. Next, we dive into the related works to further motivate our proposed dynamic runtime prediction.

2.1 Iterative computing in Flink

Flink employs a distinctive approach to iterative computing, whereby it executes the same step function in a loop. As shown in Fig. 1, the step function containing user-defined iteration logic is embedded in the Flink iteration operator. Specifically, Flink establishes iteration “head” and “tail” tasks to optimize input and output efficiency,

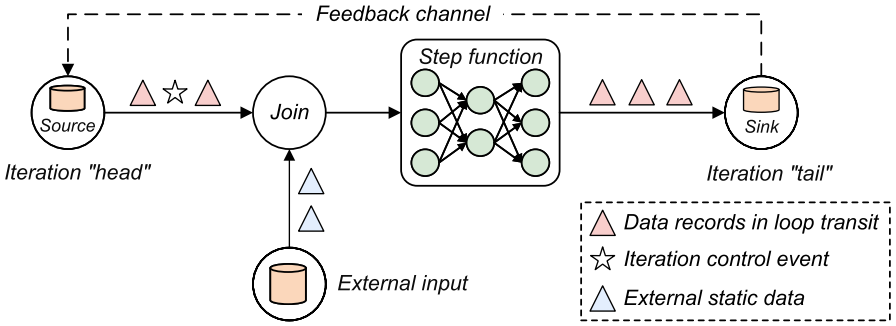


Fig. 1 Basic structure of Flink iterative computing

while also facilitating intensive intermediate data transfer between iterations through interconnected feedback channels. Moreover, Flink offers coordination in the feedback channel using control events that indicate the initiation and completion of an iteration, thereby ensuring a directed acyclic graph-based runtime [5].

2.2 BSP model

The bulk synchronous parallel (BSP) model serves as the fundament for distributed iterative computing in Flink. An example is the "vertex-centric" model, which is the most universal iteration model in Flink, and essentially is an implementation of the BSP-based Pregel [19] framework. As Fig. 2 illustrates, the BSP-based iterative computing process in Flink can be represented as a sequence of continuous supersteps. A superstep refers to the complete execution of all parallel subtasks in the current iteration, with synchronization barriers between adjacent supersteps to prevent overlapping executions. To better reflect the distributed nature of Flink iterative computing, this paper analyzes the iterative process using the superstep as the basic unit. Specifically, the runtime of a superstep is defined as the time interval between the earliest start time and the latest completion time of subtasks.

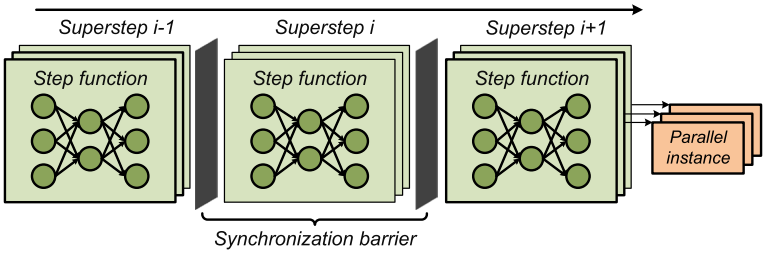


Fig. 2 Distributed iterative computing based on BSP model in Flink

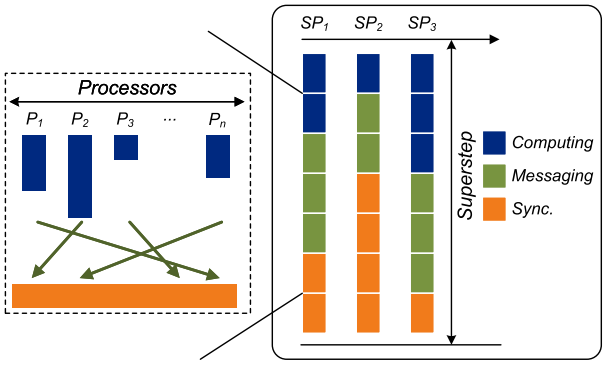


Fig. 3 Execution phase of a superstep in the BSP model

Each superstep consists of three phases, as shown in Fig. 3, and concludes upon satisfying the iteration termination condition.

- Local computing. Each processor independently performs a computing task using the data partitioned by the system.
- Messaging. Different processors communicate with each other to exchange data and control information.
- Barrier synchronization. All processors synchronize at the synchronization barrier to ensure data consistency and correctness.

2.3 Related work

Runtime prediction is widely used to optimize resource allocation and job scheduling strategies of distributed systems, as well as for feasibility analysis. In recent years, this area has gained a great deal of attention from researchers.

2.3.1 Offline runtime prediction

Offline runtime prediction methods consist of two categories: (*i*) fine-grained prediction methods that consider the job's internal structure; and (*ii*) black-box prediction methods that rely on ML models and do not need to delve into the job internals.

Fine-grained method: Starfish [11] is a self-tuning framework atop Hadoop that analyzes job runtime metrics obtained from sampling execution (i.e., rapid running on small sample datasets). It adjusts job configuration based on estimated job runtime. Similarly, Predict [22] captures the key features of Giraph iterative jobs by sampling execution to identify the critical path and predict the end-to-end completion time. Ernest [28] designs a performance prediction model to estimate the runtime of communication-intensive Spark jobs, relying on computing and communication modes observed in various numbers of cloud instances. Wang et al. [30] use partial

input data to simulate the actual job execution, and continuously collect fine-grained metrics (e.g., I/O overhead and resource consumption) to predict the runtime of each stage in the Spark job. Masha [4] enables the runtime prediction of big data applications in resource-constrained cluster environments through sampling execution. Juggler [3] considers the impact of cache constraints in each iteration and application parameters on runtime, configuring the cluster for optimal execution time and cost.

Black-box method: Benefiting from the flourishing of artificial intelligence, the ML models trained with recent job histories can make black-box predictions about the runtime via their powerful information abstraction and mining abilities. Sayeh et.al [2] propose a gray-box modeling approach, which includes a white-box model to predict the size of resilient distributed datasets and a black-box model to further estimate the end-to-end runtime of Spark jobs. Fu et al. [10] identify the limitations of prediction accuracy in applications by assessing the impact of multiple ML models on performance prediction, and enhanced average accuracy through system-level modifications. Yadwadkar et al. [33] develop a black-box performance modeling tool named Paris, which selected the best instance type by training a random forest model for each instance, and analyzed the cold-start test application on a small number of instances to obtain the model input. Xiao et al. [32] monitor the performance of simulation applications to calculate features such as CPU and memory utilization, which were then fed into an ML model with a stacking ensemble to make predictions.

2.3.2 Online runtime prediction

Since Dongen et al. [8] proposed a nonparametric regression method to predict the remaining runtime for a given situation, many researchers have extended this research using different ML algorithms (e.g., clustering techniques, individual and ensemble regressors, see [29]). Recently, more and more research has shifted its focus to deep learning algorithms (e.g., recurrent neural networks such as LSTM) to improve applicability to different jobs. Most relevantly, Hilman et al. [12] construct runtime feature sequences by monitoring the resource consumption of distributed jobs, and then used an LSTM network trained in an online manner to predict the runtime of scientific workflows in cloud environments. Pham et al. [21] propose a two-stage deep learning model-based prediction approach to predict the runtime of a continuous workflow task. Yue et al. [34] further extend this way to iterative jobs. When $(t-1)$th iteration ends, it first estimates the runtime feature of tth iteration, then predicts its runtime according to iterations in the latest sliding count window.

The existing research offers valuable insights into predicting the runtime of distributed iterative jobs. However, only a few works [3, 22, 28] focus on iterative jobs while considering their unique characteristics. Specifically, the offline methods among them primarily focus on resource consumption and execution structure, overlooking other relevant information crucial for dynamic prediction. The online methods still have not escaped the high latency of online training. In contrast, *TimeLink* overcomes these limitations by taking into account comprehensive offline and runtime information about the iterative job execution. Relying on the design

of fine-grained job matching, *TimeLink* eliminates the necessity for frequent model training, and the training overhead can be shared by subsequent executions.

3 Scheme of *TimeLink*

In what follows, we present the general procedure of *TimeLink*, which consists of similarity-based iterative job matching and dynamic runtime prediction algorithms.

3.1 Overview

We propose *TimeLink*, a dynamic runtime prediction scheme for Flink iterative jobs, as illustrated in Fig. 4. When a Fink user requests a prediction during the execution of iterative jobs, *TimeLink* immediately responds via the following four stages and outputs a prediction of its remaining runtime:

- Building the static and dynamic similarity models for iterative jobs, which are based on offline features captured through sampling execution and online features continuously collected during runtime, respectively (Sect. 3.2).
- Training the similarity-based job-matching algorithm that identifies historical jobs with comparable performance consumption for the current job, thereby forming a matched job set (Sect. 3.3).
- Given the continuity of runtime bias, the remaining runtime of the current iterative job can be estimated independently according to each historical job in the matched job set (Sect. 3.4).
- Deriving the final prediction by weight-averaging the individual predictions, with each weight reflecting the similarity between the corresponding historical job and the current job (Sect. 3.4).

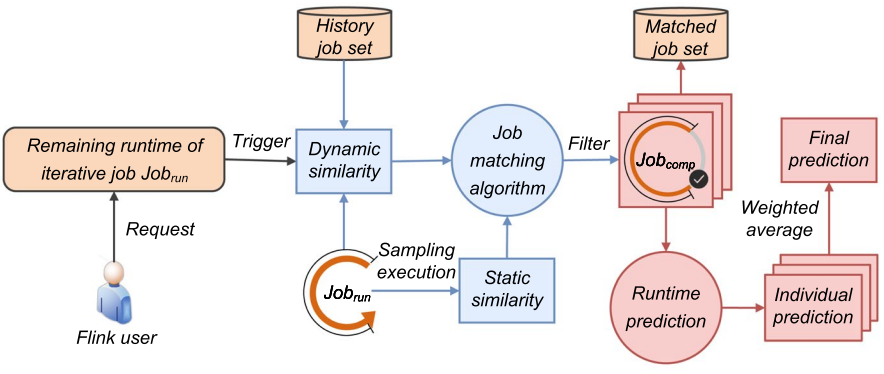


Fig. 4 Execution framework diagram of *TimeLink*

3.2 Job similarity

In fact, the more similar the historical and current iterative jobs are, the closer they are in terms of runtime consumption. The selection of similarity metrics is crucial for accurately predicting the runtime of the current job based on recent job history. We thus characterize the fine-grained similarity between historical and current jobs using several metrics, which are divided into static and dynamic similarities. For clarity, the important symbols are listed in Table 1.

3.2.1 Static similarity

Production analytics jobs are generally running periodically on an hourly or daily basis [15]. However, the resource allocation, program parameters, and the amount of input data are usually various for different executions of each job. This difference will affect the performance of each superstep in its lifecycle and will be exacerbated due to inter-iteration dependencies. These meta-data of job configurations are constant and their impact on each superstep is reflected in their processing power and is independent of the runtime environment.

Considering the BSP-based Flink iterative process, we capture the offline performance features of the current job through sampling execution. These features will be utilized to construct the static similarity that measures the extent of similarity between the current job prior to formal execution and the historical jobs. Specifically, for the graph datasets, we adopt the bias random jump (BRJ) sampling method [22] and set the sampling rate to 10% to reflect the distribution of the original dataset in the sample dataset as much as possible. For other datasets, we directly sample randomly according to the same ratio. Moreover, the parameters utilized in the execution sampling must account for scaling tuning to maintain a constant iteration number after sampling, as per the proposed scale function [22].

Table 1 Main notation

Symbol	Description
J_{run}	The current iterative job is being executed
J_{comp}	Any completed historical iterative job
m	The number of supersteps completed by the current iterative job
$[m]$	The form of set $\{1, 2 \ldots, m\}$
M	The total number of supersteps for the current iterative job
S_i	The ith similarity between the historical and current iterative job
DoS	The overall similarity of the historical and current iterative job

Table 2 Offline execution features for iterative jobs

Feature	Description
d^i_{act}	The Number of active data units in the ith superstep
d^i_{tot}	The Number of input data units in the ith superstep
msg^i_k	The Number of messages passed remotely in the ith superstep
μ^i_k	Data size passed by the kth message in the ith superstep

With the above sampling execution, the offline features shown in Table 2 are captured and combined with the BSP model to construct static similarities. The data unit in Table 2 represents a node (for graph data), or a data record (for structured data). Concretely, the static similarity is as follows:

Calculation similarity: The calculation similarity compares the number of active data units in each superstep. In the local computing phase, assuming uniform computing costs for each data unit, then the time spent on the computation is proportional to the number of active data units. The calculation similarity S_1 is defined as:

$$S_1 = \max\left\{0, \frac{\sum_{i\in[M]}\left(rate^i_{J_{\text{run}}} \cdot \text{rate}^i_{J_{\text{comp}}}\right)}{\sqrt{\sum_{i\in[M]}\left(\text{rate}^i_{J_{\text{run}}}\right)^2}\cdot\sqrt{\sum_{i\in[M]}\left(\text{rate}^i_{J_{\text{comp}}}\right)^2}}\right\}, \tag{1}$$

where $\text{rate}^i_J = d^{\,i}_{\text{act}}(J)/d^{\,i}_{\text{tot}}(J)$ denotes the fraction of active data units in the ith superstep of iterative job J.

Communication similarity: Communication similarity compares the amount of remotely transmitted data in each superstep. During the messaging phase, all nodes generate intermediate data that must be transmitted to the destination node via the network. Therefore, the duration is directly proportional to the amount of remotely delivered data. In particular, Flink employs an efficient shared memory mechanism for inter-process communication, resulting in negligible communication costs for local messages. The communication similarity S_2 is defined as:

$$S_2 = \max\left\{0, \frac{\sum_{i\in[M]}\left(\text{Msg}^i_{J_{\text{run}}} \cdot \text{Msg}^i_{J_{\text{comp}}}\right)}{\sqrt{\sum_{i\in[M]}\left(\text{Msg}^i_{J_{\text{run}}}\right)^2}\cdot\sqrt{\sum_{i\in[M]}\left(\text{Msg}^i_{J_{\text{comp}}}\right)^2}}\right\}, \tag{2}$$

where $\text{Msg}^i_J = \sum_{k\in[\text{msg}^i(J)]}\mu^i_k(J)$ denotes the total amount of data remotely transferred by the ith superstep of iterative job J.

Note that the synchronization phase is essentially an information interaction process among different nodes, which is implicitly covered in the communication similarity without explicit expression.

3.2.2 Dynamic similarity

To estimate the effect of the execution environment on job runtime, we continuously monitor runtime statistics at the granularity of supersteps to construct dynamic similarity. It measures the degree of similarity (DoS) between the currently running job and a historical job, which is shown below.

Runtime similarity: Runtime similarity compares the duration of each superstep from the beginning of job execution to the current state. If the completed supersteps of the current iterative job exhibit a high similarity in terms of duration to the historical jobs, it is reasonable to infer that the remaining supersteps are also likely to display similar performance characteristics. The runtime similarity S_3 is defined as:

$$S_3 = \max\left\{0, \frac{\sum_{i\in[m]}\left(\text{time}^i_{J_{\text{run}}} \cdot \text{time}^i_{J_{\text{comp}}}\right)}{\sqrt{\sum_{i\in[m]}\left(\text{time}^i_{J_{\text{run}}}\right)^2}\cdot\sqrt{\sum_{i\in[m]}\left(time^i_{J_{\text{comp}}}\right)^2}}\right\}, \tag{3}$$

where $time^i_J$ is the runtime of the ith superstep in iterative job J.

Convergence similarity: Convergence similarity compares the amount of data processed by each superstep from the beginning of job execution to the current state. The convergence behavior of Flink iterative jobs is influenced by various factors, such as data distribution, algorithm parameters, and cluster status. These factors are difficult to directly capture. However, the amount of processed data provides an intuitive expression. For instance, if a superstep processes only a small amount of data, it can be reasonably considered that the iterative job has converged [14]. The convergence similarity S_4 is defined as:

$$S_4 = \max\left\{0, \frac{\sum_{i\in[m]}\left(\text{data}^i_{J_{\text{run}}} \cdot \text{data}^i_{J_{\text{comp}}}\right)}{\sqrt{\sum_{i\in[m]}\left(\text{data}^i_{J_{\text{run}}}\right)^2}\cdot\sqrt{\sum_{i\in[m]}\left(\text{data}^i_{J_{\text{comp}}}\right)^2}}\right\}, \tag{4}$$

where $data^i_J$ is the amount of data processed by the ith superstep in iterative job J.

Resource utilization similarity. Resource utilization similarity compares the resource utilization of each superstep from the beginning of job execution to the current state. In Yarn [27], a popular resource management system, computing resources (e.g., CPU and memory) are represented abstractly as containers and utilized as basic units for resource allocation. Thus, we regard these containers as objects to gather information on resource utilization. These utilizations are stored in the array $Util_J[hr, sp, cid]$, where hr represents the type of computing resources, and sp and cid denote the index of supersteps and containers, respectively. For instance, $Util_J[i, n, k]$ denotes the utilization of computing resource i in kth container when

iterative job J executes the nth superstep. We classify resource utilization similarity into two categories:

A. The CPU utilization similarity S_5 is defined as:

$$S_5 = \max\left\{0, \frac{1}{\text{ctNum}} \cdot \sum_{j\in[\text{ctNum}]} \Delta_{\text{rsc}}(\text{cpu}, j)\right\}, \tag{5}$$

B. The memory utilization similarity S_6 is defined as:

$$S_6 = \max\left\{0, \frac{1}{\text{ctNum}} \cdot \sum_{j\in[\text{ctNum}]} \Delta_{\text{rsc}}(\text{mem}, j)\right\}, \tag{6}$$

where *ctNum* is the number of containers allocated by Yarn for iterative jobs, and $\Delta_{\text{rsc}}(hr, j)$ is calculated as:

$$\Delta_{\text{rsc}}(hr, j) = \frac{\sum_{i\in[m]} \left(\text{Util}_{J_{\text{run}}}[hr, i, j] \cdot \text{Util}_{J_{\text{comp}}}[hr, i, j]\right)}{\sqrt{\sum_{i\in[m]} \left(\text{Util}_{J_{\text{run}}}[hr, i, j]\right)^2} \cdot \sqrt{\sum_{i\in[m]} \left(\text{Util}_{J_{\text{comp}}}[hr, i, j]\right)^2}}. \tag{7}$$

Algorithm 1 Similarity-based iterative job-matching algorithm.

Input : current job J_{run}, historical job set H, weight vector $\mathbf{w}$, and threshold t.
Output : matched job set C and overall similarity set DOS.

```
Initialize the set C as an empty set;
Initialize the set DOS as an empty set;
for each historical job h in set H do
    Initialize the set S as an empty set;
    for i from 1 to 6 do
        Gather job runtime statistics;
        Calculate the independent similarity S_i between job J_run and h;
        Add S_i to the set S;
    end for
    Calculate the overall similarity DoS between job J_run and h using S and w;
    if DoS > t then
        Add the overall similarity DoS to the set DOS;
        Add job h to the set C;
    end if
end for
return C and DOS;
```

It is noteworthy that Eq. (5) and (6) provide a full measurement of resource distribution within the node by comparing the resource utilization of pairs of containers with equal numbers in each superstep, which solely applies to iterative jobs utilizing identical container quantities. In cases where the number of containers used by two

iterative jobs differs, we will directly compute the similarity between the average resource utilization of each iterative job across all containers.

3.3 Similarity-based iterative job matching

When a Flink user requests a runtime prediction for the current iterative job that has completed the mth superstep, *TimeLink* filters a batch of historical jobs with similar performance consumption to the current job. These matched jobs serve as the foundation for subsequent runtime predictions. The iterative job-matching algorithm based on similarity consists of three steps, as described in Algorithm 1.

Step 1: Multiple similarities $S = \{S_1, \ldots, S_6\}$ between each historical job and the current job are calculated using the job similarity model in Sect. 3.2 at the mth superstep.

Step 2: These discrete similarities are combined into a single metric, called *overall similarity*, to measure the value of similarity between two iterative jobs. It is defined as $DoS = \frac{1}{W}\sum_{i=1}^{6} w_i \cdot S_i$, where $W = \sum_{i=1}^{6} w_i$ and w_i is a nonnegative parameter that weights the priority of similarity S_i.

Step 3: Historical jobs that match the condition $DoS > t$, where t is a threshold, are screened and stored in the matched job set C. Meanwhile, their overall similarities compose the set DOS.

The performance of Algorithm 1 mainly depends on two hyperparameters in the job similarity model, i.e., weight vector $\mathbf{w} = (w_1, \ldots, w_6)$ and threshold t. They control the quality and quantity of matching results, respectively. The training process of these two parameters will be introduced below.

Dataset generation: Firstly, two iterative jobs j and k are randomly selected from the historical iterative jobs set H. Suppose job j is running and has just completed the mth superstep, and job k is a historical job that matches job j. Secondly, the random weight vector $\mathbf{w}$ is used to calculate the overall job similarity DoS between job k and j when their mth superstep is completed. Finally, *TimeLink* predicts the remaining runtime $\hat{\tau}$ of job j based on job k and its DoS. The accuracy of the prediction acc is determined by utilizing the actual value τ of the remaining execution time for job j. By repeating the above operation for each pair of historical iterative jobs $j, k \in H$, the parameter training dataset P with the element form of $\langle DoS, \hat{\tau}, \tau, acc \rangle$ can be obtained.

Parameter training: It is essential to identify the optimal weight vector and threshold, ensuring that a sufficient number of high-similarity matches are filtered out.

(i) weight vector. A set $P(\mathbf{w})$ with elements form of the $(\hat{\tau}(\mathbf{w}), \tau)$ is extracted from the set P as the weight training set. Additionally, the stochastic gradient descent (SGD) algorithm is employed to optimize the hyperparameters of weight vector $\mathbf{w}$ while minimizing mean square error (MSE) as the loss function:

$$MSE(\mathbf{w}) = avg\{(\hat{\tau}(\mathbf{w}) - \tau)^2 | (\hat{\tau}(\mathbf{w}), \tau) \in P(\mathbf{w})\}, \tag{8}$$

where $\hat{\tau}(\mathbf{w})$ is the remaining runtime predicted using $\mathbf{w}$ as the weight.

(ii) threshold. We utilize the optimal weight vector $\mathbf{w}_{opt}$ to construct the training data, and derive a sequence of points $P(\mathbf{w}_{opt})$ with the elements form of (*DoS*, *acc*) as the threshold training set. Further, we establish the subsequent function to guide the search of threshold t:

$$\begin{cases} h(t) = avg\{acc | (DoS, acc) \in P(\mathbf{w_{opt}}), DoS \geq t\}, \\ n(t) = \dfrac{|\{acc | (DoS, acc) \in P(\mathbf{w_{opt}}), DoS \geq t\}|}{|P(\mathbf{w_{opt}})|}, \end{cases} \tag{9}$$

where the optimal accuracy function $h(t)$ represents the mean accuracy corresponding to all overall similarities greater than the threshold t, while the optimal similarity ratio function $n(t)$ denotes the proportion of all points whose overall similarities exceed the threshold t.

Figure 5 illustrates an example of threshold selection using functions $h(t)$ and $n(t)$, with the scatter points representing the distribution of elements within set $P(\mathbf{w}_{opt})$. A high threshold value t_2 results in a mean accuracy close to 1, but limits the available iterations for matching and ultimately reduces the generalization capability of the prediction algorithm. A relatively low threshold t_1 ensures that sufficient historical jobs are selected to obtain a similar prediction accuracy, thereby improving the prediction algorithm's robustness. Based on these observations, we set the following threshold search objective:

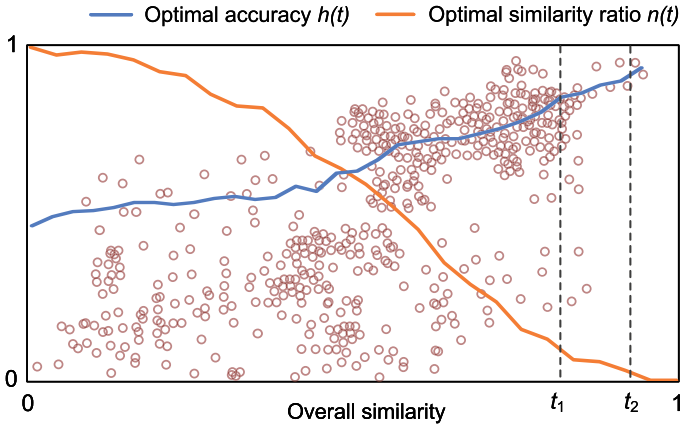


Fig. 5 An example of threshold selection using functions $n(t)$ and $h(t)$

$$\max \quad t$$
$$s.t. \quad \begin{cases} h(t) \geq h_{\min}, \\ n(t) \geq n_{\min}, \\ 0 \leq t \leq 1. \end{cases} \tag{10}$$

where the minimum average precision h_{min} (default of 0.8) is utilized to prevent excessively low similarity in matched jobs due to a low threshold. Similarly, the minimum proportion of similarity n_{min} (default of 0.2) is employed to avoid an insufficient number of historical jobs resulting from a high threshold. The highest threshold that satisfies constraints in Eq. (10) is selected, and if no historical jobs meet this criterion, the threshold is gradually lowered until a historical job is matched by Algorithm 1.

3.4 Runtime prediction

TimeLink utilizes a matched job set to predict the remaining runtime of the current iterative job. Specifically, it first uses each matched job to independently predict the remaining runtime, and then combines these individual predictions through weighted averaging to obtain a final prediction.

It is assumed that the historical job and the current job have comparable iteration numbers, which is based on the fact that iterative jobs or other batch jobs often require periodic analysis and processing of historical data with similar distribution in actual production [7]. Algorithm 2 demonstrates the dynamic runtime prediction for Flink iterative jobs. Upon completion of the mth superstep by the current iterative job, the following steps are executed.

Algorithm 2 Dynamic runtime prediction algorithm for Flink iterative jobs.

Input: current job J_{run}, historical job set H, the number of completed supersteps m, the total number of supersteps M.
Output: the remaining runtime $\hat{\tau}_{remain}$ of J_{run}.
1: $C, DOS =$ algorithm1(J_{run}, H); ▷ Get the matched job set and the overall similarity set by Algorithm 1
2: **for all** job c in set C **do**
3: **for** superstep k from 1 to m **do**
4: Calculate the bias of k-th superstep between jobs c and J_{run};
5: **end for**
6: Calculate the average bias δ_c between jobs c and J_{run} with Eq. (11);
7: **for** superstep k from $m+1$ to M **do**
8: Estimate the runtime of superstep k based on the average bias;
9: **end for**
10: Calculate the individual prediction result $\hat{\tau}_c$ with Eq. (12);
11: **end for**
12: Calculate the final prediction result $\hat{\tau}_{remain}$ with Eq. (13);
13: **return** $\hat{\tau}_{remain}$;

Step 1 - *TimeLink* first calculates the average bias of elapsed time for completed supersteps between each matched job and the current iterative job. Given that the cluster state and input data size of the current job may differ from those of historical jobs, even with high overall similarity, significant differences in runtime may exist. Hence, the average runtime bias δ_i between the current job and the ith matched job is utilized to quantify this gap.

$$\delta_i = \frac{1}{m} \cdot \sum\nolimits_{k\in[m]} \frac{\tau_{run}[k] - \tau^i_{comp}[k]}{\tau^i_{comp}[k]}, \tag{11}$$

where $\tau_{run}[k]$ and $\tau^i_{comp}[k]$ represents the actual runtime of the kth superstep in the current job and the ith matched job, respectively.

Step 2 - Relying on the continuity of runtime bias, *TimeLink* independently predicts the remaining runtime of the current iterative jobs using each matched job. Specifically, it assumes that the variances in runtime between completed supersteps will persist uniformly until the end of job execution. Thus, the individual prediction $\hat{\tau}_i$ for the ith matched job can be formulated as follows.

$$\hat{\tau}_i = \sum_{k=m+1}^{M} \tau^i_{comp}[k] \cdot (1 + \delta_i), \tag{12}$$

where M is the total number of supersteps in the current job. Then we can yield an estimate set of the same size as the matched job set, where the elements are a series of individual prediction results.

Step 3 - *TimeLink* obtains the final prediction through a weighted average of all individual results in the estimate set. Each utilizes the overall similarity between its corresponding historical job and current job as its weight, meaning that a historical job with higher similarity to the current job contributes more significantly to the final prediction. The final remaining runtime $\hat{\tau}_{remain}$ of the current iterative job is calculated as follows:

$$\hat{\tau}_{remain} = \sum\nolimits_{i\in[|\mathbf{C}|]} (DoS_i \cdot \hat{\tau}_i) / \sum\nolimits_{i\in[|\mathbf{C}|]} DoS_i, \tag{13}$$

where DoS_i represents the overall similarity between the ith matched job and the current job, $|\mathbf{C}|$ is the size of the matched job set.

Anomalous supersteps: The runtime of each superstep in the Flink iterative job should be non-absolutely decreasing in order, i.e., the closer to convergence, the shorter the runtime. However, there are inevitable conditions in the Flink cluster such as resource competition among multiple jobs or system failures, which make the runtime of some supersteps in historical jobs much higher than expected. This irregularity causes two types of anomalous supersteps that need to be preprocessed before algorithm execution.

For the historical job $j = \{sp_1, \ldots, sp_m, \ldots, sp_M\}$, if the average runtime bias of the top i supersteps is more than twice that of the top $i-1$ supersteps, then $sp_i (i \leq m)$ is the **anomalous superstep before the current iteration**. If we ignore the handling of such supersteps, the presence of just a few outliers with large

deviations will make the average bias highly inaccurate and thus indirectly affect the remaining supersteps to be predicted, and we thus eliminate these anomalous supersteps to solve it.

For the historical job $j = \{sp_1, \ldots, sp_m, \ldots, sp_M\}$, if the runtime of superstep sp_i is more than twice that of the sp_{i-1}, then $sp_i (i > m)$ is an **anomalous superstep after the current iteration**. Such supersteps are used for individual prediction in Eq. (12), so their anomalies can directly distort the results. However, they only interfere with the runtime prediction of a single superstep and have a small impact compared to the anomalous supersteps before the current iteration. Hence, *TimeLink* adjusts by reducing the overall similarity between the historical jobs with such anomalous supersteps and the current job, as follows:

$$\mathrm{DoS}_j^* = \mathrm{DoS}_j \cdot (1 - f_j), \tag{14}$$

where f_j is the percentage of such anomalous supersteps in job j and DoS_j^* indicates the adjusted overall similarity. Combined with Eq. (13), historical jobs with anomalous supersteps after the current iteration only have less contribution to the final prediction rather than being discarded.

Complexity analytics: Algorithm 2 outlines the dynamic prediction process of *TimeLink*. Let the set of historical jobs be H, the matched job set be C, the total number of supersteps be p, and the number of recently completed supersteps be q. Notably, Algorithm 1, a subalgorithm of Algorithm 2, only requires traversing the set of historical jobs to compute the job similarity and performing the job filtering, with a time complexity of $O(|H|)$. Therefore, the main performance bottleneck of *TimeLink* lies in the individual prediction part, which consists of two steps: (*i*) traversing each job in the set C to compute the average deviation coefficients of the q completed supersteps and (*ii*) making individual predictions for the $p - q$ unexecuted supersteps. Therefore, Algorithm 2 traverses the whole matched job set and the supersteps of all jobs in it, so its time complexity is $O(|C| \cdot (q + (p - q)))$. Notably, the algorithm only works on few but high-quality matched jobs after screening, with low space complexity and relatively small computation, so the overall time overhead is not high.

4 Implementation

In this section, we implement *TimeLink* atop Flink on Yarn by adding several components, which are seamlessly integrated into the JobManager and TaskManagers within Flink. Notably, Yarn is a popular resource manager who oversees the resource scheduling function of Flink. It utilizes containers (i.e., an abstraction of resources such as CPU, memory, and network) as the fundamental unit for resource allocation, which conveniently enables us to access resource statistics information in the shared cluster. Otherwise, accurately pinpointing the resource usage of each job is difficult, due to the presence of coexisting jobs within the multi-tenant Flink system.

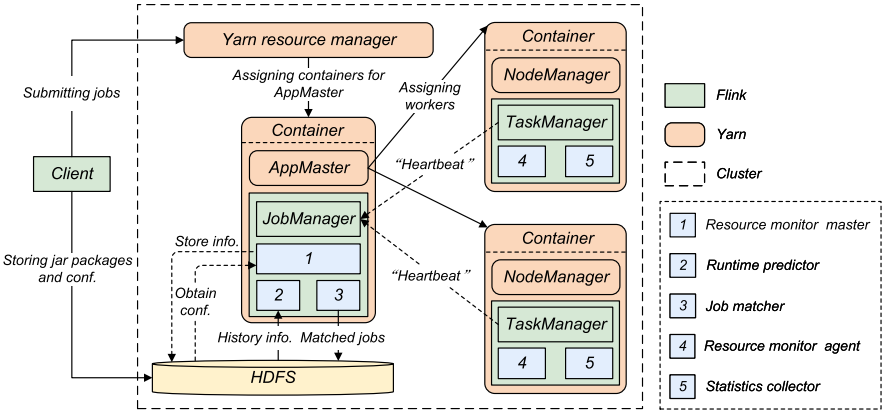


Fig. 6 Architecture overview of *TimeLink*

4.1 System design

As illustrated in Fig. 6, each component of *TimeLink* operates as an independent JVM class. Designed with modularity in mind, these components can be effortlessly adapted to other frameworks utilizing Yarn as a resource manager with minimal adjustments. Their specific functions is described below.

Job matcher: Job matcher implements the proposed job-matching algorithm that filters historical jobs with similar performance consumption for the current iterative job to be evaluated. It fetches the historical job information stored on HDFS and uses the *GradientDescentOptimizer* in the *Flink-ml* library to train the parameters within Algorithm 1. Note that the parameters are updated incrementally rather than being trained frequently, which occurs automatically once a certain amount of historical job information has been accumulated. In general, the job matcher calculates the similarity between the current job and each historical job separately. The static and dynamic similarities rely on offline statistics obtained from sampled executions and runtime statistics (e.g., resource utilization and convergence of supersteps) continuously acquired during formal executions. Finally, it outputs the matched job set filtered by the Algorithm 1 to the runtime predictor.

Runtime predictor: Runtime predictor implements the proposed dynamic runtime prediction algorithm. It triggers the job matcher to obtain the matched job set, and then runs Algorithm 2 to continuously output an estimate of the remaining job runtime. As the iterative process advances, the availability of runtime statistics for successive supersteps increases. Consequently, the matching of similar historical jobs becomes more accurate. In turn, these small but high-quality matched jobs contribute to increasingly accurate predictions of the remaining runtime.

Since these two core components rely on real-time statistics in terms of the system and the job, a distributed monitoring subsystem integrated with Flink needs to be deployed. It uses Akka for inter-node communication and HDFS for statistical information storage to ensure compatibility with Flink. Resource utilization of each container is obtained through *MetricReport* [20], a native performance monitoring

library provided by Flink, while job runtime statistics at superstep granularity are collected by embedding the statistics code in Flink's iteration operator. The monitoring subsystem consists of two components with the following design details.

Resource monitor: Resource monitor consists of multiple agent sides and a master side. Each agent side is deployed on a TaskManager (i.e., container) to collect its resource usage information, such as CPU and memory utilization. More specifically, each TaskManager configures *MetricReport* to fetch utilization information of the containers it belongs to on a second-by-second basis, and push it to the master side within the JobManager via the "heartbeat" mechanism continuously. At the end of a superstep, the master side aggregates equally the utilization records for each container in supersteps to obtain the desired *CPUMetric* and *MemoryMetric*.

Statistics collector: Statistics collector is essentially a code segment integrated into the step function in an iterative job to monitor the runtime statistics for each TaskManager. For instance, we log the commencement and conclusion time stamps for the parallel instance in each TaskManager, determining its runtime by evaluating the time difference. The processed data volume can be obtained in a similar way. These statistics are transmitted to the JobManager via Akka, where they are aggregated by superstep. Note that the overall runtime of a superstep is determined by identifying the maximum runtime among its parallel instances, and its actively processed data volume is obtained through summation.

The *jobListener* class is natively supported in the Flink system to obtain the contextual state of a job and trigger operations when the job satisfies a certain state. We thus design the sampling execution process as a *jobListener* and register it manually through the API when the user writes the job codes. When an iterative job is submitted, its sampling execution process is automatically triggered by notifying the *onJobSubmitted* method within the corresponding listener to quickly get and store the offline job information. Similarly, the runtime predictor component is also implemented as a *jobListener*. When a superstep is completed, it is triggered to work via the *onSuperstepEnd* method, thus providing immediate feedback on the prediction result.

4.2 Running process of *TimeLink*

We developed a job submission tool based on the "Yarn per-job" way. It simulates users submitting iterative jobs via invoking the "flink run" command within the client. The Flink on Yarn system starts as a short-standing cluster that provides isolation between jobs. Specifically, Yarn allocates a container to deploy AppMaster (master) responsible for initiating the Flink JobManager. It then assigns a preconfigured number of containers to deploy NodeManagers (slave), which, in turn, start the Flink TaskManagers. As shown in Fig. 6, the components of *TimeLink* are also launched in these two types of containers.

When a job is submitted, its sampling execution is automatically triggered by the *jobListener*. The tasks are scheduled to the corresponding TaskManager to execute based on the user-configured number of containers for this job. When a

superstep ends, the job matcher calculates the similarity of the current job to each historical job, based on offline statistics stored in HDFS and runtime statistics obtained in real time. It then outputs the matched job set via Algorithm 1. Finally, the runtime predictor feedbacks the user in real time the evaluated value of the remaining running time via Algorithm 2. Note that the model training process for the job matcher is done manually at regular intervals, and the automated process described above only requires the loading of updated parameters.

5 Experiment evaluation

In this section, we present the experimental methodology, and evaluate the prediction accuracy and time overhead of *TimeLink*.

5.1 Cluster settings

Experiment environment: Technical specifications outlined in the experiment were implemented in Flink 1.8.0 and coded in Java. The Flink cluster comprises nine homogeneous physical nodes, consisting of one master node for deploying the JobManager and eight slave nodes for deploying the TaskManagers, each configured with 16 slots. All nodes are connected with 10-Gigabit Ethernet, and each has two Intel Xeon Silver 4210 CPUs @ 2.20GHz (10 cores $\times$ 2 threads), 128 GB memory, and 1 TB SSD. Hadoop version 2.7.0 (for storing data on HDFS and managing resources through Yarn) is chosen. Each Yarn container is configured with 8 CPU cores (with hyperthreading technology) and 20GB memory, allowing up to 5 containers (i.e., TaskManager) to run simultaneously on each node. Consequently, our cluster can offer a maximum job parallelism of 640 across 40 TaskManagers. Moreover, all iterative jobs involved in the experiments were implemented using Flink's standard iteration operators, and their default parallelism is 128.

Baselines: Based on a comprehensive review of relevant works, we have selected the following three algorithms to compare their performance with *TimeLink*:

- **Juggler** [3]: the optimal approach for supporting iterative jobs is a pioneer in an offline prediction that takes into account cache limitations and the impact of application parameters on runtime.
- **OffOpt** [10]: the optimal offline approach that supports generic jobs identifies program-internal limitations on the predictive accuracy of ML models and then makes system-level modifications to improve average accuracy.
- **OnOpt** [21]: the optimal online method that supports generic jobs monitors their resource consumptions to construct runtime feature sequences and then uses an LSTM trained in an incremental learning manner to make online predictions.

Evaluation metrics: In this paper, the relative average error (RAE) is utilized to assess the accuracy of runtime prediction. RAE provides a more reliable reflection of the predictive effect and facilitates comparison across different methods. Let n denote the total number of predictions, and RAE is calculated method as follows:

Table 3 Iterative jobs and their datasets

Job	Dataset	Size [GB]	Parameter
Pagerank (PR)	Wikipedia	5.74	d = 0.85, 150 iterations
	Friendster	31.16	
	LiveJournal	10	
	UK-2006	4.7	
Connected Component (CC)	LiveJournal	10	150 iterations
	Wikipedia	5.74	
	Friendster	31.16	
	RoadNet-CA	6.9	
Single-Source Shortest Path (SSSP)	UK-2006	4.7	150 iterations
	RoadNet-CA	6.9	
	LiveJournal	10	
	Wikipedia	5.74	
K-Means (KM)	Points-1	10	5 clusters, 150 iterations
	Points-2	15	
	Points-3	20	

$$RAE = \frac{1}{n}\sum_{i=1}^{n}\frac{|actual_i - predict_i|}{actual_i} \cdot 100\% \tag{15}$$

Jobs and datasets: We select four typical iterative jobs, with each job evaluated on various datasets. The detailed information is presented in Table 3.

(i) Real dataset: The LiveJournal dataset is a large-scale collection of mutual relation data among users on the LiveJournal social network, encompassing both friend and interest connections. The UK-2006 dataset pertains to web link networks in the United Kingdom and contains key information such as web meta-data and jump link relationships. The WikiPedia dataset comprises articles, pages, and associated information from the Wikipedia website, which is maintained by the KONECT project [25]. The Friendster dataset encompasses a network structure of Friendster social network users and their friendship relationships, as well as group relationships involving more than three individuals. RoadNet-CA dataset is composed of road network data from southern California, including road network topology information such as road starting points and bifurcated intersections, which are maintained by Stanford large network dataset [24].

(ii) Simulated dataset: The Points-*n* dataset is a series of randomly generated point sets with different data sizes based on the Gaussian mixture model. The data in each point set conform to Gaussian distribution as a whole, with random cluster centers and an equal probability of mutation. Compared to other iterative algorithms, K-means has a relatively simple iterative process and deals with straightforward data relations, making it suitable for running on randomly generated Point-*n* datasets.

Table 4 RAE for different runtime prediction algorithms (%)

Job	OffOpt	Juggler	OnOpt # *TimeLink* (Ours)				
			k = 5%	k = 25%	k = 45%	k = 65%	k = 85%
PR	21.2	15.4	16.6 # 20.5	18.23 # 8.7	16.3 # 9.5	13.6 # 12.8	19.5 # 11.9
CC	19.9	15.7	17.3 # 18.5	13.3 # 15.4	15.2 # 6.9	13.1 # 11.5	16.5 # 12.4
SSSP	18.1	14.9	15.2 # 17.7	16.6 # 9.8	14.4 # 7.7	13.5 # 9.3	18.8 # 14.3
KM	16.3	12.3	16.9 # 17.3	10.1 # 11.3	13.5 # 9.3	13.7 # 7.6	15.2 # 8.5
Average	18.9	14.6	16.5 # 18.5	15.5 # 11.3	14.8 # 8.3	13.4 # 10.3	17.5 # 11.8

5.2 Overall performance

To evaluate the overall performance improvement of *TimeLink*, we compared its RAE with three baselines, maintaining identical cluster configurations and default parameters. Table 4 displays the RAE of various runtime prediction algorithms. Here, k represents the RIP of iterative jobs during online prediction requests, and "#" serves as a data separator.

When $k = 5\%$, *TimeLink* essentially operates in an offline prediction state, resulting in an RAE close to that of OffOpt, but notably distinct from Juggler. The reason is that we solely focus on critical features during sampling execution, contrary to Juggler's emphasis on the impact of fine-grained offline performance features (e.g., cache limitation and application parameters) in their presented execution time models. However, *TimeLink*'s performance does not necessitate reliance on full offline features, due to the inherent differences between offline and online prediction. When $k = 25\%$, *TimeLink* achieves its lowest average RAE, showcasing a 3.5% improvement in prediction accuracy over Juggler. This highlights the substantial potential for enhanced accuracy through the utilization of iterative job statistics during runtime. Notably, for Pagerank and SSSP, OnOpt, designed for general jobs, experiences more pronounced performance degradation. This is mainly attributed to the high complexity of the business logic and the increased number of iterations compared to other jobs, while OnOpt cannot consider fine-grained iterative features during runtime. When $k > 25\%$, the RAE of *TimeLink* experiences a modest increase but consistently remains within the range of 8% to 12%, consistently outperforming the baselines. As OnOpt treats iterative jobs as intricate batch jobs, its RAE decreases with accumulating runtime information (i.e., as k increases). However, the influence of the convergence nature becomes more pronounced during later iterations, resulting in an increase in RAE. *TimeLink* achieves its lowest average RAE at $k = 45\%$, presenting an improvement of approximately 6.24% in average prediction accuracy over Juggler, the best-performing baseline. This improvement is attributed to Juggler's reliance on a customized set of parametric models for predicting the running time of each iteration. Consequently, when the cluster environment changes, the prediction accuracy of all models is affected, even with cross-validation.

It is worth noting that OffOpt is not intended for offline prediction of iterative jobs, resulting in higher RAE compared to Juggler and *TimeLink* under all iterative

jobs. As such, predicting the runtime of iterative jobs is more intricate than that of regular batch jobs.

5.3 Deep dive of *TimeLink*

We conduct a deep dive into *TimeLink* to validate the effectiveness of the proposed job similarity. We also present the accuracy of *TimeLink* at various iteration moments to gain a better understanding of the benefits of dynamic runtime prediction.

Impact of job progress on prediction accuracy: *TimeLink* conducts the job matching at the end of each superstep and uses the obtained matched job set to predict the remaining runtime. The execution process of iterative jobs is denoted by relative iterative progress (RIP), which is defined as the ratio of the number of completed iterations to the total number of iterations.

As shown in Fig. 7, *TimeLink*'s relative error exhibits a stable trend when RIP ranges from 30% to 70%, maintaining a range between 7.57% and 13.21%. At the early stages of iterative job execution, the relative error shows a short and rapid upward trend due to the limited availability of runtime statistical information, coupled with a significant proportion of static similarity in *TimeLink*, resulting in an almost offline prediction state that causes the relative error to increase. As the RIP

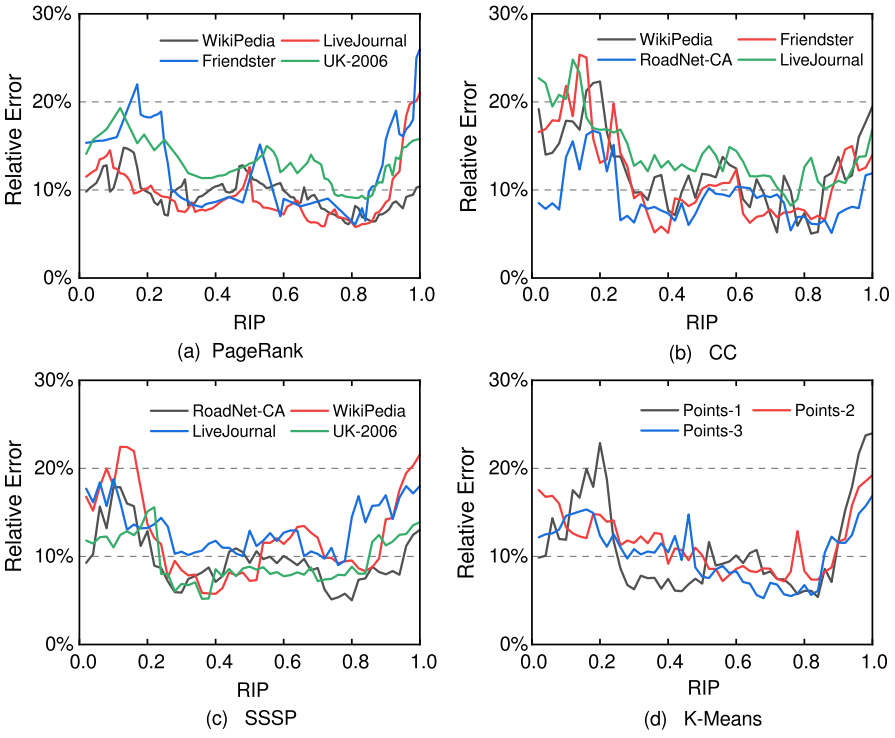


Fig. 7 Relative errors for iterative jobs with different RIPs

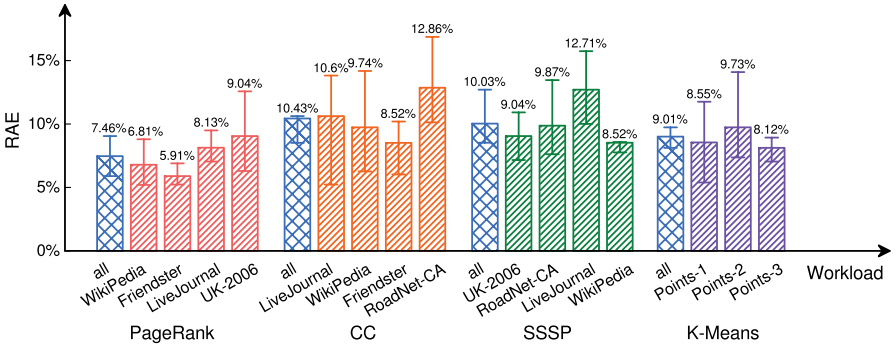
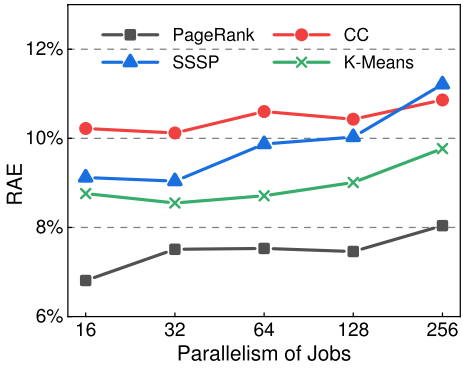


Fig. 8 RAE for iterative jobs under different datasets

Fig. 9 RAE for iterative jobs with different parallelisms



increases, more runtime statistical information is captured by *TimeLink* to build dynamic similarity, resulting in an improvement in the quality of the matched job set and a gradual decrease in relative error, reaching minimum values between 25% and 30%. However, toward the end of the iterative job execution, the relative error experiences a rapid increase due to the limited remaining runtime. As such, even a small absolute error could translate into a significant relative error. Furthermore, while differences in data distribution and sizes may cause slight variations in relative errors for these four iterative jobs when using different datasets, their overall trends remain consistent and insignificant.

Prediction accuracy with various workloads: For each workload (i.e., a combination of a dataset and an iterative job) in Fig. 7, we calculate its RAE using a series of representative samples. Specifically, we choose these samples according to specific criteria, starting at 5% of the total number of supersteps and taking five consecutive points at 20% intervals. Figure 8 shows the RAE comparison for the four iterative jobs across different datasets, where "*all*" represents the average RAE of the iterative job across all datasets. It can be seen that the RAE of all iterative jobs remains relatively stable, maintaining around 9%, with a fluctuation range of less than 5%. Specifically, the PageRank demonstrates the lowest RAE of 5.91% under the Friendster dataset. It should be noted that the CC has a short end-to-end runtime,

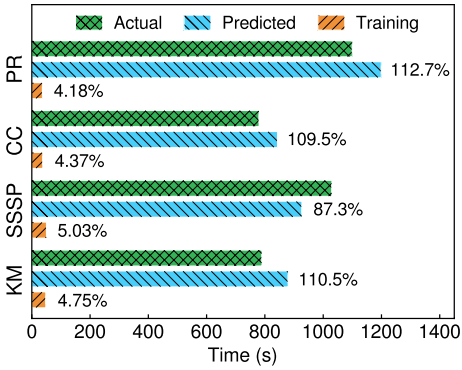


Fig. 10 Job-matching model building overhead of *TimeLink* for different jobs

even a low absolute error can lead to a high relative error, resulting in the highest RAE of 12.86% in the RoadNet-CA dataset.

Scalability evaluation: In practice, jobs are generally run at regular intervals. Due to the uncertain resource (e.g., slot) constraints available in the shared cluster, accurately predicting the runtime of jobs with different parallelism is crucial, especially when serving the performance optimization algorithms [3] (e.g., searching for optimal parallelism that can achieve a certain runtime limit). To evaluate such scalability of *TimeLink*, we submit an iterative job multiple times and vary its parallelism. The results are shown in Fig. 9. There is an increase in RAE attributed to the decrease in absolute time. Moreover, *TimeLink* exhibits no significant performance degradation as the job parallelism increases. This is because these meta-data of job configurations are constant and their impact on each superstep is reflected in their processing power and is independent of the runtime environment. Considering the BSP-based Flink iterative process, we have captured the static similarity of iterative job consumption at two levels: computation and communication, by sampling executions. In particular, they are designed in the form of ratios. As such, performance discrepancies resulting from variations in job parallelism have been quantified and integrated into *TimeLink*.

Table 5 Prediction overhead of *TimeLink* under different RIP

Job	Prediction Overhead			
	25%	45%	65%	85%
PageRank	25.6ms	23.6ms	17.8ms	15.8ms
CC	21.6ms	25.4ms	23.4ms	19.2ms
SSSP	24.3ms	24.7ms	23.8ms	21.3ms
K-Means	19.2ms	18.4ms	18.2ms	16.8ms

5.4 *TimeLink* overhead

Prediction overhead: The main goal of *TimeLink* is to provide runtime predictions with low time overheads online, as high prediction accuracy alone may be challenging to apply in downstream optimization algorithms. As mentioned in Sect. 3.4, *TimeLink* has a time complexity of $O(n^2)$. To further quantify its runtime overhead, we measure the prediction overhead of each job under RIPs from 25% to 85%. Table 5 shows that the prediction overhead for all jobs is in the millisecond range, which is negligible compared to the job runtimes that generally span hundreds of seconds. Specifically, as the iteration progresses, the matched job set of *TimeLink* becomes more accurate, and the number of them gradually converges to n_{min}. Thus, the time overhead of runtime prediction on this basis also decreases. Moreover, the prediction times of different jobs are very close to each other. This uniformity arises because *TimeLink*'s time complexity is independent of the job type and only related to the matching accuracy.

Building job-matching model: We assess the offline time to build the job-matching model, i.e., the relative training time spent on each iterative job. The results are depicted in Fig. 10, where the scales are presented as ratios to the actual job end-to-end runtimes. As shown, *TimeLink* is capable of maintaining the training time under all jobs within 5% of the actual runtime while ensuring high prediction accuracy. This suggests that *TimeLink* has a negligible impact on the end-to-end runtime (hundreds of seconds) of the iterative jobs. Additionally, the model training is infrequent, occurring only after a significant accumulation of job history. Therefore, *TimeLink* supports the prediction of the remaining runtime of an iterative job as it runs, and provides accurate predictions promptly.

6 Conclusion

This paper presents *TimeLink*, an advanced dynamic runtime prediction algorithm for Flink iterative jobs. Upon receiving a prediction request from the user, *TimeLink* utilizes historical iterative jobs matched in real time to predict the remaining runtime of the current job. The fine-grained similarity between iterative jobs is established via the captured offline and online features, serving as the foundation for job matching and final prediction. The parameters of the job-matching model were trained to enable the automatic adaptation of the matching algorithm to various iterative jobs. We conducted extensive experiments using various iterative algorithms and datasets within the Flink system. The results indicate that *TimeLink* has a relative average error of 5.91% and 12.86% in the best and worst cases, respectively. Compared with the optimal baselines, it improves the average prediction accuracy by 6.24%.

In the future, we will consider more complex scenarios, such as multi-job situations with interference and uncertainty. Meanwhile, this paper describes the important role of runtime prediction in resource provisioning, we thus plan to design a dynamic resource scaling mechanism for Flink iterative jobs based on *TimeLink* to maximize resource utilization and minimize operating costs.

Author Contributions All authors made equal contributions to the research design and analysis. Xiaofei Yue wrote the main manuscript text and developed the prototype system of *TimeLink*, Qingyang Ding wrote the main manuscript, performed the experimental analysis and plotted figures 7-10, Jianming Zhu directed the writing of the manuscript and the developing of the prototype system, and Yanbing Ding performed the experimental analysis and prepared figures 1-6. All authors read and approved the final manuscript, and Qingyang Ding is the corresponding author.

Funding This research was partially supported by the R&D Program of Beijing Municipal Education Commission (KM202211417008), the Beijing Social Science Foundation Program (23GLC037), the National Key Research and Development Program of China (2022YFC3300804).

Data Availability The data presented in this study are available upon reasonable request from the corresponding author.

Declarations

Conflict of interest All authors have no competing interests as defined by Springer, or other interests that might be perceived to influence the results and discussion reported in this paper.

Ethical approval Ethical review and approval were waived in this study because our research focused on the design and implementation of algorithms in the distributed system did not involve research with humans or animals, and was not within the scope of requiring ethical review.

References

1. Agarwal S, Kandula S, Bruno N, Wu M-C, Stoica I, Zhou J (2012) Reoptimizing data parallel computing. In: 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), pp 281–294
2. Al-Sayeh H, Hagedorn S, Sattler K (2020) A gray-box modeling methodology for runtime prediction of apache spark jobs. Distribut Parall Databases 38(4):819–839. https://doi.org/10.1007/s10619-020-07286-y
3. Al-Sayeh H, Memishi B, Jibril MA, Paradies M, Sattler K (2022) Juggler: Autonomous cost optimization and performance prediction of big data applications. In: SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, 12–17 June 2022, pp 1840–1854. https://doi.org/10.1145/3514221.3517892
4. Al-Sayeh H, Memishi B, Paradies M, Sattler K-U (2020) Masha: sampling-based performance prediction of big data applications in resource-constrained clusters. In: The 1st Workshop on Distributed Infrastructure, Systems, Programming and AI (DISPA). Very Large Data Base Endowment Inc. (VLDB Endowment)
5. Carbone P, Katsifodimos A, Ewen S, Markl V, Haridi S, Tzoumas K (2015) Apache flink™: stream and batch processing in a single engine. IEEE Data Eng Bull 38(4):28–38
6. Chen CLP, Zhang C (2014) Data-intensive applications, challenges, techniques and technologies: a survey on big data. Inf Sci 275:314–347. https://doi.org/10.1016/j.ins.2014.01.015
7. Cheng L, Wang Y, Liu Q, Epema DHJ, Liu C, Mao Y, Murphy J (2021) Network-aware locality scheduling for distributed data operators in data centers. IEEE Trans Parall Distribut Syst 32(6):1494–1510. https://doi.org/10.1109/TPDS.2021.3053241
8. Dongen BF, Crooy RA, Aalst WM (2008) Cycle time prediction: When will this case finally be finished? In: On the Move to Meaningful Internet Systems: OTM 2008: OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008, Monterrey, Mexico, 9–14 Nov 2008, Proceedings, Part I. Springer, pp 319–336
9. Everman B, Rajendran N, Li X, Zong Z (2021) Improving the cost efficiency of large-scale cloud systems running hybrid workloads-a case study of alibaba cluster traces. Sustain Comput Inf Syst 30:100528

10. Fu S, Gupta S, Mittal R, Ratnasamy S (2021) On the use of ML for blackbox system performance prediction. In: 18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, 12–14 Apr 2021, pp 763–784
11. Herodotou H, Lim H, Luo G, Borisov N, Dong L, Cetin FB, Babu S (2011) Starfish: a self-tuning system for big data analytics. In: Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings, pp 261–272
12. Hilman MH, Rodriguez MA, Buyya R (2018) Task runtime prediction in scientific workflows using an online incremental learning approach. In: 11th IEEE/ACM International Conference on Utility and Cloud Computing, UCC 2018, Zurich, Switzerland, 17–20 Dec 2018, pp 93–102. https://doi.org/10.1109/UCC.2018.00018
13. Hoseinyfarahabady MR, Jannesari A, Taheri J, Bao W, Zomaya AY, Tari Z (2020) Q-flink: A qos-aware controller for apache flink. In: 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGRID 2020, Melbourne, Australia, 11–14 May 2020, pp 629–638. https://doi.org/10.1109/CCGrid49817.2020.00-30
14. Imran M, Gévay GE, Markl V (2020) Distributed graph analytics with datalog queries in flink. In: Software Foundations for Data Interoperability and Large Scale Graph Data Analytics—4th International Workshop, SFDI 2020, and 2nd International Workshop, LSGDA 2020, Held in Conjunction with VLDB 2020, Tokyo, Japan, 4 Sept 2020, Proceedings. Communications in Computer and Information Science, vol 1281, pp 70–83. https://doi.org/10.1007/978-3-030-61133-0_6
15. Jyothi SA, Curino C, Menache I, Narayanamurthy SM, Tumanov A, Yaniv J, Mavlyutov R, Goiri I, Krishnan S, Kulkarni J, Rao S (2016) Morpheus: Towards automated slos for enterprise clusters. In: 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, 2–4 Nov 2016, pp 117–134
16. Kang U, Tsourakakis CE, Faloutsos C (2009) Pegasus: A peta-scale graph mining system implementation and observations. In: 2009 Ninth IEEE International Conference on Data Mining, pp 229–238. https://doi.org/10.1109/icdm.2009.14
17. Li P, Guo S, Miyazaki T, Liao X, Jin H, Zomaya A, Wang K (2016) Traffic-aware geo-distributed big data analytics with predictable job completion time. IEEE Trans Parall Distribut Syst 28(6):1785–1796. https://doi.org/10.1109/TPDS.2016.2626285
18. Low Y, Gonzalez J, Kyrola A, Bickson D, Guestrin C, Hellerstein JM (2012) Distributed graphlab: a framework for machine learning in the cloud. Proc VLDB Endow 5(8):716–727. https://doi.org/10.14778/2212351.2212354
19. Malewicz G, Austern MH, Bik AJC, Dehnert JC, Horn I, Leiser N, Czajkowski G (2010) Pregel: a system for large-scale graph processing. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, 6–10 June 2010, pp 135–146. https://doi.org/10.1145/1807167.1807184
20. Metric Reporters of Apache Flink. https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/deployment/metric_reporters/
21. Pham T et al (2020) Predicting workflow task execution time in the cloud using a two-stage machine learning approach. IEEE Trans Cloud Comput 8(1):256–268. https://doi.org/10.1109/TCC.2017.2732344
22. Popescu AD, Balmin A, Ercegovac V, Ailamaki A (2013) Predict: towards predicting the runtime of large scale iterative analytics. Proc VLDB Endow 6(14):1678–1689. https://doi.org/10.14778/2556549.2556553
23. Reiss C, Tumanov A, Ganger GR, Katz RH, Kozuch MA (2012) Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In: Proceedings of the Third ACM Symposium on Cloud Computing, pp 1–13
24. Stanford Large Network Dataset Collection. http://snap.stanford.edu/data/
25. The KONECT Project. http://konect.cc/
26. Tumanov A, Zhu T, Park JW, Kozuch MA, Harchol-Balter M, Ganger GR (2016) Tetrisched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In: Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18–21, 2016, pp 1–16. https://doi.org/10.1145/2901318.2901355
27. Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S, Saha B, Curino C, O'Malley O, Radia S, Reed BC, Baldeschwieler E (2013) Apache hadoop YARN: yet another resource negotiator. In: ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, 1–3 Oct 2013, pp 5–1516. https://doi.org/10.1145/2523616.2523633

28. Venkataraman S, Yang Z, Franklin MJ, Recht B, Stoica I (2016) Ernest: efficient performance prediction for large-scale advanced analytics. In: 13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, 16–18 Mar 2016, pp 363–378
29. Verenich I, Dumas M, Rosa ML, Maggi FM, Teinemaa I (2019) Survey and cross-benchmark comparison of remaining time prediction methods in business process monitoring. ACM Trans Intell Syst Technol (TIST) 10(4):1–34
30. Wang K, Khan MMH (2015) Performance prediction for apache spark platform. In: 17th IEEE International Conference on High Performance Computing and Communications, HPCC 2015, 7th IEEE International Symposium on Cyberspace Safety and Security, CSS 2015, and 12th IEEE International Conference on Embedded Software and Systems, ICESS 2015, New York, NY, USA, 24–26 Aug 2015, pp 166–173. https://doi.org/10.1109/HPCC-CSS-ICESS.2015.246
31. Wen Z, Wang Y, Liu F (2022) Stepconf: Slo-aware dynamic resource configuration for serverless function workflows. In: IEEE INFOCOM 2022—IEEE Conference on Computer Communications, London, United Kingdom, 2–5 May 2022, pp 1868–1877. https://doi.org/10.1109/INFOCOM48880.2022.9796962
32. Xiao Y, Yao Y, Zhu F, Chen K (2021) Simulation runtime prediction approach based on stacking ensemble learning. In: Proceedings of the 11th International Conference on Simulation and Modeling Methodologies, Technologies and Applications, SIMULTECH 2021, Online Streaming, 7–9 July 2021, pp 42–49. https://doi.org/10.5220/0010517600420049
33. Yadwadkar NJ, Hariharan B, Gonzalez JE, Smith B, Katz RH (2017) Selecting the *best* VM across multiple public clouds: a data-driven performance modeling approach. In: Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, 24–27 Sept 2017, pp 452–465. https://doi.org/10.1145/3127479.3131614
34. Yue X, Shi L, Zhao Y, Ji H, Wang G (2021) Online runtime prediction method for distributed iterative jobs. In: Web Information Systems and Applications: 18th International Conference, WISA 2021, Kaifeng, China, 24–26 Sept 2021, Proceedings 18. Springer, pp 156–168

Authors and Affiliations

Xiaofei Yue[1] · Qingyang Ding[2] · Jianming Zhu[3] · Yanbing Ding[4]

✉ Qingyang Ding
gltqingyang@buu.edu.cn

Xiaofei Yue
xfyue@bit.edu.cn

Jianming Zhu
zjm@cufe.edu.cn

Yanbing Ding
dingyanbing@cebbank.com

[1] School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China

[2] School of Management, Beijing Union University, Beijing 100101, China

[3] School of Information, Central University of Finance and Economics, Beijing 102206, China

[4] Department of Finance and Technology, China Everbright Bank, Beijing 100045, China