Computers, Materials & Continua

Tech Science Press

Doi:10.32604/cmc.2025.062007

ARTICLE

Modeling and Performance Evaluation of Streaming Data Processing System in IoT Architecture

Feng Zhu*, Kailin Wu and Jie Ding

School of Computer, Jiangsu University of Science and Technology, Zhenjiang, 212100, China

*Corresponding Author: Feng Zhu. Email: zhufeng@just.edu.cn

Received: 08 December 2024; Accepted: 17 February 2025; Published: 16 April 2025

ABSTRACT: With the widespread application of Internet of Things (IoT) technology, the processing of massive real-time streaming data poses significant challenges to the computational and data-processing capabilities of systems. Although distributed streaming data processing frameworks such as Apache Flink and Apache Spark Streaming provide solutions, meeting stringent response time requirements while ensuring high throughput and resource utilization remains an urgent problem. To address this, the study proposes a formal modeling approach based on Performance Evaluation Process Algebra (PEPA), which abstracts the core components and interactions of cloud-based distributed streaming data processing systems. Additionally, a generic service flow generation algorithm is introduced, enabling the automatic extraction of service flows from the PEPA model and the computation of key performance metrics, including response time, throughput, and resource utilization. The novelty of this work lies in the integration of PEPA-based formal modeling with the service flow generation algorithm, bridging the gap between formal modeling and practical performance evaluation for IoT systems. Simulation experiments demonstrate that optimizing the execution efficiency of components can significantly improve system performance. For instance, increasing the task execution rate from 10 to 100 improves system performance by 9.53%, while further increasing it to 200 results in a 21.58% improvement. However, diminishing returns are observed when the execution rate reaches 500, with only a 0.42% gain. Similarly, increasing the number of TaskManagers from 10 to 20 improves response time by 18.49%, but the improvement slows to 6.06% when increasing from 20 to 50, highlighting the importance of co-optimizing component efficiency and resource management to achieve substantial performance gains. This study provides a systematic framework for analyzing and optimizing the performance of IoT systems for large-scale real-time streaming data processing. The proposed approach not only identifies performance bottlenecks but also offers insights into improving system efficiency under different configurations and workloads. The code and experimental results are accessible at https://github.com/giszhu/PESdps (accessed on 10 January 2025).

KEYWORDS: System modeling; performance evaluation; streaming data process; IoT system; PEPA

1 Introduction

The widespread application of Internet of Things (IoT) systems has generated massive amounts of real-time streaming data. These data are characterized by high frequency, real-time processing, and diversity, originating from various fields such as energy monitoring and control, smart manufacturing, smart cities, healthcare, and precision agriculture. This situation poses an unprecedented challenge to the computational and data-processing capabilities of IoT systems. To cope with this serious challenge, distributed streaming data processing frameworks have emerged, such as Apache Flink, Apache Spark Streaming, Apache Storm, etc. Their large-scale deployments in the cloud or on edge computing nodes provide powerful real-time data

processing capabilities for IoT systems, but they also raise new challenges. Ensuring high throughput and high utilization while meeting the stringent response time requirements of IoT applications has become a hot topic in current research, particularly concerning quality of service (QoS). Achieving optimal QoS and efficiency in IoT systems is a critical issue that researchers are actively addressing.

Researchers have compared the performance of different streaming data processing systems [1–3]. However, the performance of these systems is highly dependent on the cloud environment in which they are deployed. Existing research has also explored the scalability issues associated with deploying streaming data processing systems in the cloud [4]. On the other hand, to ensure the QoS performance of IoT systems, recent studies have proposed optimization strategies to address QoS challenges in Fog/Edge environments, such as latency and bandwidth limitations. These strategies include adaptive resource management, task scheduling, and processing data near data sources through edge devices to alleviate network and computation pressures [5–7]. Building on these efforts, researchers modeled a three-tier IoT architecture consisting of cloud, fog, and devices and conducted performance evaluations from both computing and communication dimensions [8]. However, these studies primarily focused on resource optimization strategies rather than formal modeling. Moreover, although formal modeling methods such as queuing theory, Petri nets, and timed automata have been applied to IoT systems [9–11], these methods often face scalability challenges and struggle to capture the dynamic and concurrent nature of large-scale streaming data processing. Therefore, formal modeling and comprehensive performance evaluation remain underexplored for IoT systems handling large-scale streaming data processing.

To address this gap, the study aims to optimize the performance of IoT systems handling large-scale real-time streaming data by applying the Performance Evaluation Process Algebra (PEPA). As a formal modeling language, PEPA uses algebra as a tool to decompose a concurrent system at different levels into a number of subsystems, which are combined through concurrent actions to form an extensive system [12]. Due to its compositional and scalable nature, PEPA is well-suited for modeling the performance of cloud-based streaming data processing systems, which involve concurrency and dynamic workloads. Therefore, this study investigates the performance modeling and analysis of a cloud-based streaming data processing system within the IoT architecture using PEPA. The goal is to provide a systematic framework for analyzing performance metrics such as resource utilization, throughput, and response time under different loads and configurations, ultimately offering insights into system optimization.

The main contributions of this paper are as follows:

(a) This study proposes a generic service flow generation algorithm that can automatically extract service flows from the PEPA model. By tracking these active service flows, the algorithm enables the computation and analysis of performance metrics such as response time, throughput, and utilization. This method enhances the ability to evaluate and optimize the performance of IoT systems under different configurations and workloads.

(b) For the first time, the PEPA language is applied to model cloud-based streaming data processing systems within the IoT architecture. This approach provides a new perspective for optimizing the performance of IoT systems oriented towards large-scale real-time streaming data processing.

The primary innovation of this paper lies in the development of the service flow generation algorithm, which bridges the gap between formal modeling and practical performance evaluation for IoT systems. Additionally, the application of PEPA to IoT streaming data processing systems demonstrates its potential as a scalable and compositional modeling tool for addressing the challenges of large-scale, real-time data processing in IoT environments.

The structure of the paper is as follows: Section 2 reviews existing work related to IoT performance evaluation, highlighting the strengths and weaknesses of various approaches. Section 3 introduces PEPA and uses it to model the streaming data processing system in IoT architecture. Section 4 describes methods for solving performance metrics and proposes a service flow extraction algorithm for better response time analysis. Section 5 conducts performance evaluation. Finally, Section 6 summarizes the results and outlines future research directions.

2 Related Work

This section explores several common modeling approaches and analyzes their applicability in this scenario, comparing them to the chosen PEPA approach.

Queuing theory, in particular queuing networks, is widely used to analyze system performance metrics such as latency, throughput, and resource utilization. However, queuing network models often rely on simplifying assumptions like infinite buffers and simplified service models, which deviate from the realities of stream processing systems. For example, Chen et al. [13] demonstrated that finite buffer size significantly impacts QoS in stream processing topologies, a factor not adequately captured by standard queuing network models with their infinite buffer assumption. Meanwhile, Moreno-Vozmediano et al. [9] highlighted the need for dynamic resource allocation in complex stream processing environments, advocating for the integration of machine learning techniques. While their work emphasizes the importance of dynamic resource allocation, it doesn't provide a formal modeling approach like PEPA to analyze the performance implications of different allocation strategies. Furthermore, queuing theory struggles to formally represent the complex interactions and combinations of system components and often fails to capture the concurrency and synchronous behavior inherent in stream processing frameworks.

Stochastic Petri Nets (SPNs) and Stochastic Reward Networks (SRNs) offer the capability to model concurrency and synchronization but suffer from the state space explosion problem when applied to large-scale IoT systems. Liu et al. [14] encountered this issue when modeling a blockchain-based agricultural supply chain using Petri nets, where the system's heterogeneity and complex operations led to a rapidly expanding state space, making model validation challenging. While extensions like Mobile SRNs (MSRNs), proposed by Kabashkin et al. [15] for modeling intra-IoT mobility, attempt to mitigate the state space explosion problem, they introduce additional complexity in model construction and analysis. Similarly, Liu et al. [16] illustrated the state space explosion challenges faced by SRNs in analyzing complex timing systems. Moreover, the syntax of Petri nets and SRNs limits their expressiveness in representing complex system architectures, particularly the intricate interactions between components and the data flow processing logic [17]. For instance, Song et al. [10] used Petri nets to model the traceability of food supply chains, but Petri nets lack the capability to integrate big data stream processing techniques, highlighting their limitations in handling complex stream processing logic. Our work utilizes PEPA to effectively model the data flow and processing logic within stream processing frameworks, addressing this limitation of Petri nets.

Timed automata (TA) are able to accurately describe the behavior and temporal constraints of real-time systems. However, its state space explosion problem limits its application in complex IoT systems. For example, Chen et al. [18] pointed out the difficulties TA faces in effectively modeling the heterogeneous, large-scale, and dynamic nature of IoT architectures. Additionally, TA struggles to capture the probabilistic and stochastic behavior common in IoT systems. As highlighted in [19], converting distributed systems characterized by asynchronous communication and node autonomy into TA models presents inherent challenges, hindering the accurate derivation of performance metrics like throughput and response time. Our approach using PEPA can model probabilistic behavior and asynchronous communication, enabling more accurate performance analysis.

In contrast, PEPA offers a more suitable modeling approach for this study. Its process-based approach effectively captures the concurrency and interaction within stream processing frameworks in IoT systems. The combinatorial and modular modeling capabilities of PEPA enhance model comprehensibility and extensibility. Ding et al. [11] demonstrated PEPA's effectiveness in supply chain modeling and response time calculation. However, their work focuses on supply chains and doesn't address the specific challenges of stream processing in IoT systems, which is the focus of our research. Chen et al. [20] utilized PEPA to model a hybrid cloud-fog system and propose an intelligent scheduling scheme. While their work utilizes PEPA, it focuses on scheduling in cloud-fog environments, whereas our work leverages PEPA to model and analyze the performance of stream processing frameworks within IoT systems. As summarized in Table 1, PEPA addresses key limitations of existing IoT data processing frameworks by capturing data flow characteristics, such as data dependencies and dynamic processing logic, while effectively avoiding the state space explosion problem.

Table 1: Comparative analysis of iot data processing frameworks

Reference	Objective	Key findings	Dataset	Limitations	Year
[13]	Evaluate the performance of streaming data processing systems in IoT using Queuing Theory.	Demonstrated scalability issues in cloud-based streaming systems under high workloads.	Synthetic workload generator.	Lacks formal modeling; limited to empirical evaluation.	2023
[14]	Propose a framework for real-time streaming data processing in IoT using SPNs.	Achieved improved response time using distributed processing frameworks.	Public IoT datasets (e.g., smart grid data).	No formal evaluation of system scalability or bottleneck identification.	2023
[16]	Propose a stochastic modeling approach using SRNs to optimize transportation processes in IoT-enabled intermodal systems.	Demonstrated reduced delays in operations using SPNs and Markov chains. Shows enhanced process scalability via real-time IoT-enabled data flows for transportation.	IoT sensor data from smart logistics systems.	Limited evaluation of stochastic IoT workloads or dynamic conditions in supply chain networks.	2024

(Continued)

Table 1 (continued)

Reference	Objective	Key findings	Dataset	Limitations	Year
[18]	Analyze the scalability of cloud-based streaming systems using TA.	Highlighted the impact of resource allocation strategies on system performance.	Simulated workloads for cloud environments.	Limited to cloud-only systems; lacks consideration of dynamic workloads or real-time constraints.	2024
[20]	Model and analyze a three-tier IoT architecture (cloud, fog, edge) using PEPA.	Identified bottlenecks in communication and computation in fog-based IoT systems.	IoT sensor data from smart city applications.	Focused only on communication delays; no detailed analysis of resource utilization or throughput.	2020
This Work	Develop a formal modeling approach for IoT real-time streaming systems using PEPA.	Proposed a PEPA-based framework for modeling and evaluating throughput, response time, and utilization. Identified bottlenecks under varying workloads.	Synthetic workloads and public IoT datasets (e.g., smart manufacturing).	Focuses on modeling and simulation; experimental validation in real-world IoT deployments is ongoing.	NA

3 System Modeling

This section delves into the key aspects of system modeling. Section 3.1 presents the cloud-based streaming data processing framework, which discusses the architecture and methodologies employed for handling large volumes of streaming data in cloud environments. Following this, Section 3.2 introduces a system model based on PEPA language, providing a formal approach to modeling and analyzing system performance. This sub-section further breaks down into Section 3.2.1, which explains the PEPA language, detailing its syntax and semantics that enable the precise specification of system behaviors. Lastly, Section 3.2.2 presents the PEPA model, illustrating how this algebraic framework can be utilized to represent and evaluate complex system scenarios, thereby contributing to a deeper understanding of system performance characteristics.

3.1 Cloud-Based Streaming Data Processing Framework

In large-scale IoT-oriented applications, the cloud is required to respond to and process large amounts of streaming data quickly. Based on this goal, the study embeds the Flink architecture into the cloud, and the main framework is shown in Fig. 1. Logically, it contains three main components: the Client, the JobManager,

and the TaskManager. The Client serves as the exit point for users to submit jobs and is responsible for submitting user programs and data flow definitions to the JobManager. The JobManager is the central coordinating component of the system and is responsible for resource scheduling, task allocation, and fault recovery. The TaskManager is the work node of the system and is responsible for performing specific data processing tasks. These components interact with each other through messaging. For example, the JobManager sends task deployment instructions to the TaskManager and receives feedback on the execution status of tasks.

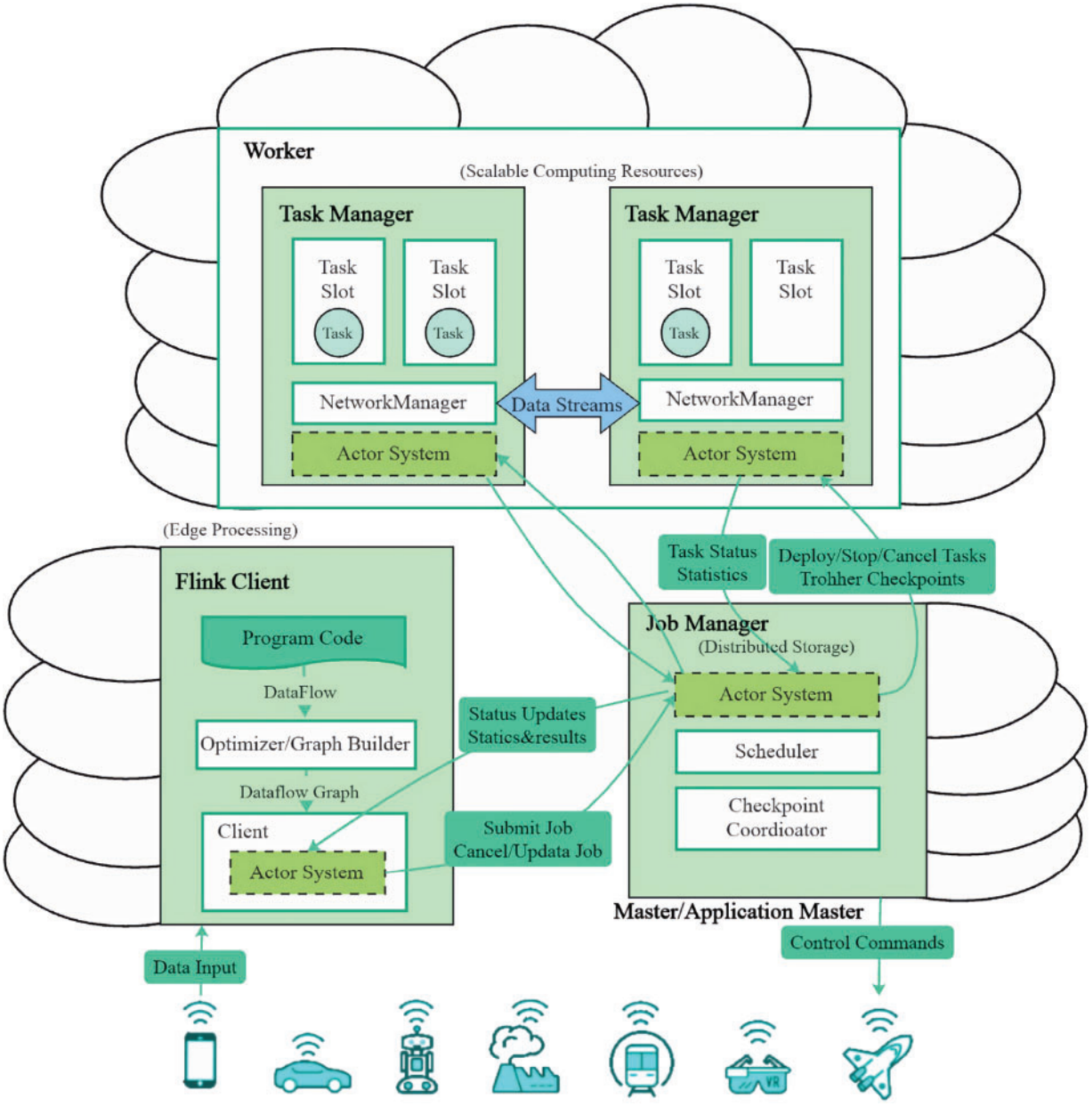

Figure 1: Cloud-based streaming data processing system architecture

The above framework structure can be naturally mapped to a common cloud-based streaming data processing system. The Client can be regarded as the management module of the system, through which users

submit jobs and monitor the status of the system. The JobManager can be considered as the control module of the system and is responsible for coordinating the various computing resources. A TaskManager can be viewed as a computing module deployed in a pool of computing resources in the cloud that is responsible for executing computing tasks submitted by users. The data input and data interface in the cloud can be interfaced with a wider range of IoT data collection and storage layers. Through this mapping relationship, the data processing flow in real scenarios can be effectively simulated, providing a basis for performance evaluation and optimization.

3.2 System Model Based on PEPA

3.2.1 PEPA Language

As a formal modeling language, PEPA is particularly suitable for describing the concurrent behaviors of a system due to its composability, accuracy, and dynamic nature. It has been widely used in communication networks [21,22], supply chain and manufacturing [11], as well as in the fields of system reliability [23] and transportation [24,25].

The basic syntax and rules of the PEPA language are as follows:

(a) **Prefix Operation:** $(\alpha, \lambda).E$. This illustrates the fundamental relationship between components and actions. α denotes the action type, with execution duration following an exponential distribution of rate λ. Post-execution, the component behaves as E.

(b) **Constant Definition:** $C \triangleq E$. This equation indicates that constant C exhibits behavior akin to component E.

(c) **Choice Operation:** $E_1 + E_2$. The system may manifest as either E_1 or E_2. Competition exists between them, with both activities enabled. Upon completion of one activity, others are discarded.

(d) **Parallel Operation:** $E_1 \parallel E_2$. This represents two concurrent yet entirely independent components.

(e) **Cooperation Operation:** $E_1 \bowtie_L E_2$. Components E_1 and E_2 must collaborate to complete the activity. Set L, termed the cooperation set, signifies actions of type L requiring interaction between E_1 and E_2. For actions outside L, components can execute independently. When $L = \varnothing$, components operate autonomously.

To precisely describe action types, we define action pre-sets and post-sets, as shown in Eqs. (1)–(3).

$$\mathrm{Pre}(\alpha) = \{E \mid E \xrightarrow{\alpha} E'\} \tag{1}$$

$$\mathrm{Post}(\alpha) = \{E' \mid E \xrightarrow{\alpha} E'\} \tag{2}$$

$$\mathrm{Post}(E, \alpha) = \{E' \mid E \xrightarrow{\alpha} E'\} \tag{3}$$

$\mathrm{Pre}(\alpha)$ denotes the set of components capable of executing α. $\mathrm{Post}(\alpha)$ represents the set of components reachable after α's execution. $\mathrm{Post}(E, \alpha)$ indicates the set of components attainable after component E performs action α.

For independent action α, it can be denoted as $\alpha^E \to E'$, where $E \in \mathrm{Pre}(\alpha)$ and $E' \in \mathrm{Post}(E, \alpha)$. For cooperative action α, executable by a single component of the potential execution process, it may be represented as $\alpha(E_1 \to E_1', E_2 \to E_2', \ldots, E_n \to E_n')$, where $E_i \in \mathrm{Pre}(\alpha_i)$ and $E_i' \in \mathrm{Post}(E_i, \alpha)$.

In PEPA, transitions between actions can alter the quantity of components in a given state. Assuming $x_1(E)$ represents the number of components E in state x_1, when component$E$ in state x_1 executes action α, the number of components E' reaching state x_2 changes: $x_1(E) = x_1(E) - 1$, $x_2(E') = x_2(E') + 1$. This relationship is described using an action transition matrix, represented by −1, 0, and 1.

3.2.2 PEPA Model

As shown in Fig. 2, the sequence diagram demonstrates the interaction process of each component in the cloud-based streaming data processing system. The PEPA definition is as follows:

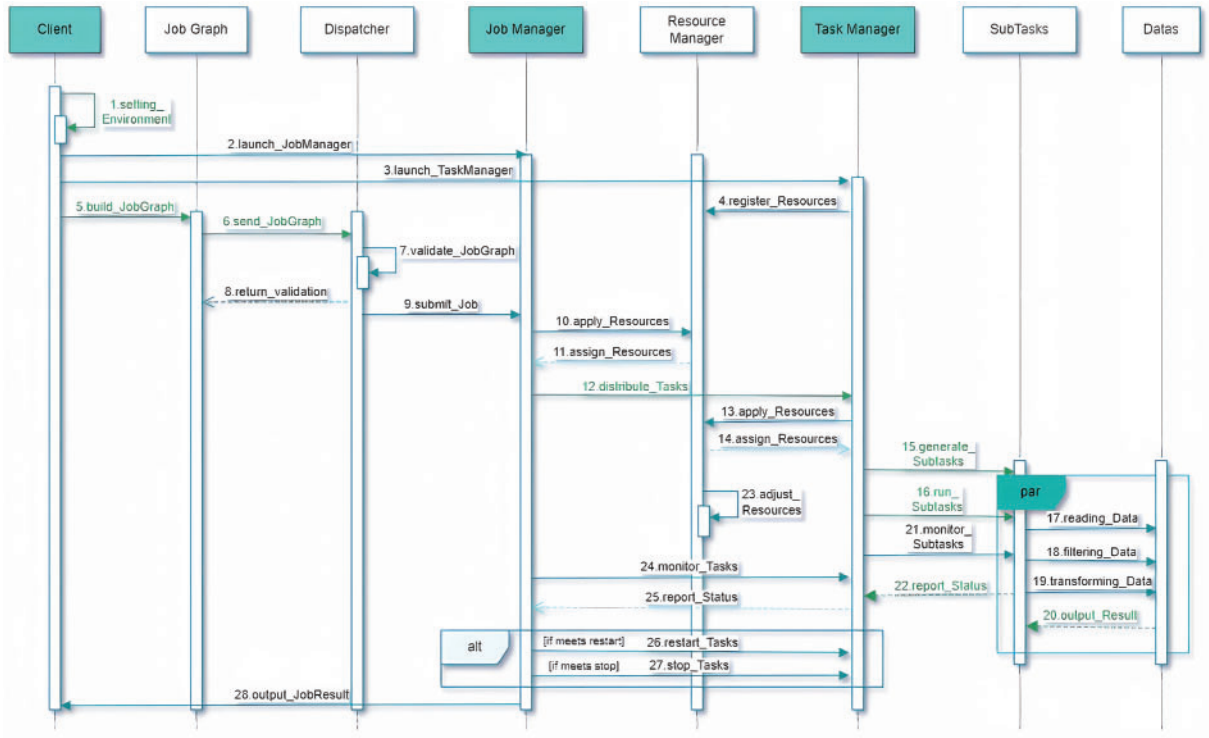

Figure 2: The sequence diagram of a cloud-based streaming data processing

Client (Ct): The Ct component is located at the front end of the system and is responsible for initializing and starting the job processing flow. First, the Ct component sets up the runtime environment, as defined in Eq. (4) to ensure that all the necessary configurations and resources are in place. Next, the Ct component launches both the JobManager and the TaskManager components according to Eq. (5), which are responsible for job management and task execution, respectively. In addition, the Ct is responsible for building the job graph as shown in Eq. (6), which is a structured description of the job execution process. After completing these steps, the JobManager outputs the results back to the Ct component as specified in Eq. (7), thus completing the entire job lifecycle.

$$\mathrm{Ct}_1 \overset{\mathrm{def}}{=} (\mathrm{setting\_Environment}, r_{\mathrm{setting\_Environment}}).\mathrm{Ct}_2 \tag{4}$$

$$\mathrm{Ct}_2 \overset{\mathrm{def}}{=} (\mathrm{launch\_JobManager}, r_{\mathrm{launch\_JobManager}}).\mathrm{Ct}_3 + (\mathrm{launch\_TaskManager}, r_{\mathrm{launch\_TaskManager}}).\mathrm{Ct}_3 \tag{5}$$

$$\mathrm{Ct}_3 \overset{\mathrm{def}}{=} (\mathrm{build\_JobGraph}, r_{\mathrm{build\_JobGraph}}).\mathrm{Ct}_4 \tag{6}$$

$$\mathrm{Ct}_4 \overset{\mathrm{def}}{=} (\mathrm{output\_JobGraph}, r_{\mathrm{output\_JobGraph}}).\mathrm{Ct}_1 \tag{7}$$

JobGraph (JG): The JG component manages the construction and validation of the job graph. Firstly, a request to build a job graph is received from the Ct component as defined in Eq. (8). The JG component is responsible for organizing the various parts of a job into a structured graph that details the execution flow of the job and the dependencies between tasks. Once built, the JG component sends the job graph to

the Dispatcher component according to Eq. (9). After validation, the JG component receives the validation results through return_Validation action as shown in Eq. (10).

$$JG_1 \stackrel{def}{=} (\text{build\_JobGraph}, r_{\text{build\_JobGraph}}).JG_2 \tag{8}$$

$$JG_2 \stackrel{def}{=} (\text{send\_JobGraph}, r_{\text{send\_JobGraph}}).JG_3 \tag{9}$$

$$JG_3 \stackrel{def}{=} (\text{return\_Validation}, r_{\text{return\_Validation}}).JG_1 \tag{10}$$

Dispatcher (Dp): The Dp component is responsible for job graph validation and job scheduling. Firstly, the job graph is received from the JG component through send_JobGraph action as specified in Eq. (11). The Dp component is then responsible for performing a detailed validation of the job graph according to Eq. (12) to ensure that its structure and logic are correct. When validation is complete, the Dp component returns the validation results to the JG component through return_Validation action as shown in Eq. (13). Finally, once the job graph is validated, the Dp component submits the job to the JobManager component based on Eq. (14). This process ensures smooth job execution and efficient system operation, as well as flexibility and accuracy in job scheduling.

$$Dp_1 \stackrel{def}{=} (\text{send\_JobGraph}, r_{\text{send\_JobGraph}}).Dp_2 \tag{11}$$

$$Dp_2 \stackrel{def}{=} (\text{validate\_JobGraph}, r_{\text{validate\_JobGraph}}).Dp_3 \tag{12}$$

$$Dp_3 \stackrel{def}{=} (\text{return\_Validation}, r_{\text{return\_Validation}}).Dp_4 \tag{13}$$

$$Dp_4 \stackrel{def}{=} (\text{submit\_Job}, r_{\text{submit\_Job}}).Dp_1 \tag{14}$$

JobManager (JM): The JM component is mainly responsible for the management and execution of the entire job. First, it receives the request to launch the job management from the Ct component as defined in Eq. (15), and then it receives the submitted job from the Dp component according to Eq. (16). The JM component then requests the required resources from the ResourceManager component as shown in Eq. (17). After obtaining the resources based on Eq. (18), it assigns the task to the TaskManager component for execution as specified in Eq. (19). During task execution, the JM component monitors task execution according to Eq. (20) and receives status reports from the TaskManager component as defined in Eq. (21). If necessary, the JM component can instruct the TaskManager component to restart the task based on Eq. (22) or stop the task as shown in Eq. (23). Finally, the JM component outputs the results of the job execution back to the Ct component according to Eq. (24), completing the job lifecycle. The process ensures effective management and smooth execution of jobs while also supporting flexibility and accuracy in job execution.

$$JM_1 \stackrel{def}{=} (\text{launch\_JobManager}, r_{\text{launch\_JobManager}}).JM_2 \tag{15}$$

$$JM_2 \stackrel{def}{=} (\text{submit\_Job}, r_{\text{submit\_Job}}).JM_3 \tag{16}$$

$$JM_3 \stackrel{def}{=} (\text{apply\_Resources}, r_{\text{apply\_Resources}}).JM_4 \tag{17}$$

$$JM_4 \stackrel{def}{=} (\text{assign\_Resources}, r_{\text{assign\_Resources}}).JM_5 \tag{18}$$

$$JM_5 \stackrel{def}{=} (\text{distribute\_Resources}, r_{\text{distribute\_Tasks}}).JM_6 \tag{19}$$

$$JM_6 \stackrel{def}{=} (\text{monitor\_Tasks}, r_{\text{monitor\_Tasks}}).JM_7 \tag{20}$$

$$JM_7 \stackrel{def}{=} (\text{report\_Status}, r_{\text{report\_Status}}).JM_8 \tag{21}$$

$$JM_8 \stackrel{def}{=} (\text{restart\_Tasks}, r_{\text{restart\_Tasks}}).JM_9 \tag{22}$$

$$JM_9 \overset{def}{=} (\text{stop\_Tasks}, r_{\text{stop\_Tasks}}).JM_{10} \tag{23}$$

$$JM_{10} \overset{def}{=} (\text{output\_JobResult}, r_{\text{output\_JobResult}}).JM_1 \tag{24}$$

ResourceManager (RM): Located in the system, it is responsible for the management and allocation of resources. First, it receives the resource registration request from the TaskManager component as defined in Eq. (25). Then, it processes the resource request from the JM component according to Eq. (26) and assigns the resources to the JM component as shown in Eq. (27). In addition, the RM component handles resource requests from the TaskManager component based on Eq. (28) and assigns resources to the TaskManager component as specified in Eq. (29). In the resource management process, the RM component can also adjust its own resource allocation according to Eq. (30) to ensure the optimal utilization of resources. This process ensures effective management and rational allocation of system resources and supports the smooth execution of work and tasks.

$$RM_1 \overset{def}{=} (\text{register\_Resources}, r_{\text{register\_Resources}}).RM_2 \tag{25}$$

$$RM_2 \overset{def}{=} (\text{apply\_Resources}, r_{\text{apply\_Resources}}).RM_3 \tag{26}$$

$$RM_3 \overset{def}{=} (\text{assign\_Resources}, r_{\text{assign\_Resources}}).RM_4 \tag{27}$$

$$RM_4 \overset{def}{=} (\text{apply\_Resources}, r_{\text{apply\_Resources}}).RM_5 \tag{28}$$

$$RM_5 \overset{def}{=} (\text{assign\_Resources}, r_{\text{assign\_Resources}}).RM_6 \tag{29}$$

$$RM_6 \overset{def}{=} (\text{adjust\_Resources}, r_{\text{adjust\_Resources}}).RM_1 \tag{30}$$

TaskManager (TM): The TM component acts as the central control unit in the system, coordinating the entire lifecycle of task management from initialization to termination. The process starts with the activation of the TM itself as defined in Eq. (31), followed by resource registration to prepare the environment according to Eq. (32). Once the resources are registered, the TM distributes the tasks to their respective components as shown in Eq. (33) and applies the necessary resources based on Eq. (34), ensuring that all the prerequisites for the task execution are in place. After the resource allocation is complete as specified in Eq. (35), the TM proceeds to generate subtasks according to Eq. (36) and start execution as defined in Eq. (37). Throughout task execution, the TM actively monitors subtasks based on Eq. (38) and reports the execution status back to the appropriate component according to Eq. (39). It also performs continuous checking of the entire task process as shown in Eq. (40) and sends periodic status updates based on Eq. (41), which is essential for maintaining system awareness. If an interruption or error occurs, the TM can restart the task as needed according to Eq. (42), ensuring resilience and continuity of operations. Finally, on task completion or termination, the TM stops all active tasks as specified in Eq. (43), and returns to its initial state, ready to start a new cycle.

$$TM_1 \overset{def}{=} (\text{launch\_TaskManager}, r_{\text{launch\_TaskManager}}).TM_2 \tag{31}$$

$$TM_2 \overset{def}{=} (\text{register\_Resources}, r_{\text{register\_Resources}}).TM_3 \tag{32}$$

$$TM_3 \overset{def}{=} (\text{distribute\_Tasks}, r_{\text{distribute\_Tasks}}).TM_4 \tag{33}$$

$$TM_4 \overset{def}{=} (\text{apply\_Resources}, r_{\text{apply\_Resources}}).TM_5 \tag{34}$$

$$TM_5 \overset{def}{=} (\text{assign\_Resources}, r_{\text{assign\_Resources}}).TM_6 \tag{35}$$

$$TM_6 \overset{def}{=} (\text{generate\_Subtasks}, r_{\text{generate\_Subtasks}}).TM_7 \tag{36}$$

$$\mathrm{TM}_7 \overset{\mathrm{def}}{=} (\mathrm{run\_Subtasks}, r_{\mathrm{run\_Subtasks}}).\mathrm{TaskManager}_8 \tag{37}$$

$$\mathrm{TM}_8 \overset{\mathrm{def}}{=} (\mathrm{monitor\_Subtasks}, r_{\mathrm{monitor\_Subtasks}}).\mathrm{TM}_9 \tag{38}$$

$$\mathrm{TM}_9 \overset{\mathrm{def}}{=} (\mathrm{report\_Status}, r_{\mathrm{report\_Status}}).\mathrm{TM}_{10} \tag{39}$$

$$\mathrm{TM}_{10} \overset{\mathrm{def}}{=} (\mathrm{monitor\_Tasks}, r_{\mathrm{monitor\_Tasks}}).\mathrm{TM}_{11} \tag{40}$$

$$\mathrm{TM}_{11} \overset{\mathrm{def}}{=} (\mathrm{report\_Status}, r_{\mathrm{report\_Status}}).\mathrm{TM}_{12} \tag{41}$$

$$\mathrm{TM}_{12} \overset{\mathrm{def}}{=} (\mathrm{restart\_Tasks}, r_{\mathrm{restart\_Tasks}}).\mathrm{TM}_{13} \tag{42}$$

$$\mathrm{TM}_{13} \overset{\mathrm{def}}{=} (\mathrm{stop\_Tasks}, r_{\mathrm{stop\_Tasks}}).\mathrm{TM}_1 \tag{43}$$

SubTasks (ST): The ST component is responsible for specific data processing tasks. First, it receives subtask generation instructions as defined in Eq. (44) and execution instructions according to Eq. (45) from the TM component. During execution, the ST component reads as shown in Eq. (46), filters based on Eq. (47), and transforms data as specified in Eq. (48), which are data operations that get the required data from the Datas component. At the same time, the ST component outputs the processed results according to Eq. (49). It also continuously monitors the execution of subtasks during the execution of each task as defined in Eq. (50). Finally, it periodically reports the status of task execution to the TM component based on Eq. (51). This process ensures accurate execution and efficient management of data processing tasks while also supporting flexibility and responsiveness in task execution.

$$\mathrm{ST}_1 \overset{\mathrm{def}}{=} (\mathrm{generate\_Subtasks}, r_{\mathrm{generate\_Subtasks}}).\mathrm{ST}_2 \tag{44}$$

$$\mathrm{ST}_2 \overset{\mathrm{def}}{=} (\mathrm{run\_Subtasks}, r_{\mathrm{run\_Subtasks}}).\mathrm{ST}_3 \tag{45}$$

$$\mathrm{ST}_3 \overset{\mathrm{def}}{=} (\mathrm{reading\_Data}, r_{\mathrm{reading\_Data}}).\mathrm{ST}_4 \tag{46}$$

$$\mathrm{ST}_4 \overset{\mathrm{def}}{=} (\mathrm{filtering\_Data}, r_{\mathrm{filtering\_Data}}).\mathrm{ST}_5 \tag{47}$$

$$\mathrm{ST}_5 \overset{\mathrm{def}}{=} (\mathrm{transforming\_Data}, r_{\mathrm{transforming\_Data}}).\mathrm{ST}_6 \tag{48}$$

$$\mathrm{ST}_6 \overset{\mathrm{def}}{=} (\mathrm{output\_Result}, r_{\mathrm{output\_Result}}).\mathrm{ST}_7 \tag{49}$$

$$\mathrm{ST}_7 \overset{\mathrm{def}}{=} (\mathrm{monitor\_Subtasks}, r_{\mathrm{monitor\_Subtasks}}).\mathrm{ST}_8 \tag{50}$$

$$\mathrm{ST}_8 \overset{\mathrm{def}}{=} (\mathrm{report\_Status}, r_{\mathrm{report\_Status}}).\mathrm{ST}_1 \tag{51}$$

Datas (Dt): The Dt component is responsible for providing data and outputting results. First, it responds to requests for data reading as defined in Eq. (52), data filtering according to Eq. (53), and data transforming as shown in Eq. (54) from the ST component, providing the raw data needed for these data processing tasks. After completing the data processing, the Dt component receives the results and performs the output based on Eq. (55) to return the processed data to the ST component. This process ensures the availability and correctness of data and supports the smooth execution of data processing tasks in the system.

$$\mathrm{Dt}_1 \overset{\mathrm{def}}{=} (\mathrm{reading\_Data}, r_{\mathrm{reading\_Data}}).\mathrm{Dt}_2 \tag{52}$$

$$\mathrm{Dt}_2 \overset{\mathrm{def}}{=} (\mathrm{filtering\_Data}, r_{\mathrm{filtering\_Data}}).\mathrm{Dt}_3 \tag{53}$$

$$\mathrm{Dt}_3 \overset{\mathrm{def}}{=} (\mathrm{transforming\_Data}, r_{\mathrm{transforming\_Data}}).\mathrm{Dt}_4 \tag{54}$$

$$\mathrm{Dt}_4 \overset{\mathrm{def}}{=} (\mathrm{output\_Result}, r_{\mathrm{output\_Result}}).\mathrm{Dt}_1 \tag{55}$$

Through the various components of the model described above, we derive the system equation, which outlines the interactions between these components. We assume that $C\,[M]$ is shorthand for the same type of activity, represented as $C\|\cdots\|C$. Thus, we can express the system formula as follows:

$$\begin{aligned}&\text{Client}[M] \underset{L_1}{\bowtie} \text{JobGraph}[N] \underset{L_2}{\bowtie} \text{Dispatcher}[O] \underset{L_3}{\bowtie} \text{JobManager}[P] \underset{L_4}{\bowtie} \text{ResourceManager}[Q]\\ &\underset{L_5}{\bowtie} \text{TaskManager}[R] \underset{L_6}{\bowtie} \text{SubTasks}[S] \underset{L_7}{\bowtie} \text{Datas}[T]\end{aligned} \tag{56}$$

$L_1 = \{\text{setting\_Environment}, \text{launch\_JobManager}, \text{launch\_TaskManager}, \text{build\_JobGraph}\},$

$L_2 = \{\text{send\_JobGraph}\},$

$L_3 = \{\text{validate\_JobGraph}, \text{return\_Validation}, \text{submit\_Job}\},$

$L_4 = \{\text{apply\_Resources}, \text{distribute\_Tasks}, \text{monitor\_Tasks}, \text{restart\_Tasks}, \text{stop\_Tasks}, \text{output\_JobResult}\},$

$L_5 = \{\text{assign\_Resources}, \text{assign\_Resources}, \text{adjust\_Resources}\},$

$L_6 = \{\text{register\_Resources}, \text{apply\_Resources}, \text{generate\_Subtasks}, \text{run\_Subtasks}, \text{monitor\_Subtasks}, \text{report\_Status}\},$

$L_7 = \{\text{reading\_Data}, \text{filtering\_Data}, \text{transforming\_Data}, \text{report\_Status}, \text{output\_Result}\}$

4 Performance Evaluation Method

This section details the performance evaluation methodology based on the PEPA model and develops it step-by-step through three stages. First, this study extracts the action transition matrix of the system from the constructed PEPA model to lay the foundation for the subsequent analysis. Next, specific algorithms are designed in this study to generate service flows from the action transition matrix. These service flows enable the simulation of system operations, allowing the calculation of system response time to assess its real-time performance. Finally, based on the acquired response time data, this study calculates the throughput and resource utilization of the system to comprehensively evaluate its performance.

4.1 Action Transition Matrix

From the PEPA semantics, it is evident that a component can transition to another component by executing an action. Executing an independent or cooperative action inevitably results in a decrease in the number of certain components in the state and an increase in others. The effect of an action on a component can be described using a column vector w consisting of 0, −1, and 1. w is the labeled action, and the non-zero element indicates the rate of transition from the current state to the next state. This subsection uses the TM component in the PEPA model, specifically discussed in Section 3.2.2, as an example. Table 2 presents the action transition matrix of the TM, where each row represents a state and each column represents an action.

Table 2: Action transition matrix for TaskManager (TM) component

	w_3	w_4	w_{12}	w_{13}	w_{14}	w_{15}	w_{16}	w_{21}	w_{22}	w_{24}	w_{25}	w_{26}	w_{27}
TM_1	−1	0	0	0	0	0	0	0	0	0	0	0	1
TM_2	1	−1	0	0	0	0	0	0	0	0	0	0	0
TM_3	0	1	−1	0	0	0	0	0	0	0	0	0	0
TM_4	0	0	1	−1	0	0	0	0	0	0	0	0	0
TM_5	0	0	0	1	−1	0	0	0	0	0	0	0	0
TM_6	0	0	0	0	1	−1	0	0	0	0	0	0	0

(Continued)

Table 2 (continued)

	w_3	w_4	w_{12}	w_{13}	w_{14}	w_{15}	w_{16}	w_{21}	w_{22}	w_{24}	w_{25}	w_{26}	w_{27}
TM_7	0	0	0	0	0	1	−1	0	0	0	0	0	0
TM_8	0	0	0	0	0	0	1	−1	0	0	0	0	0
TM_9	0	0	0	0	0	0	0	1	−1	0	0	0	0
TM_{10}	0	0	0	0	0	0	0	0	1	−1	0	0	0
TM_{11}	0	0	0	0	0	0	0	0	0	1	−1	0	0
TM_{12}	0	0	0	0	0	0	0	0	0	0	1	−1	0
TM_{13}	0	0	0	0	0	0	0	0	0	0	0	1	−1

Note: See the "Abbreviations for Table 1" section for definitions of TM and *w* values.

Abbreviations for Table 2:

TM_1: Task Manager 1, TM_2: Task Manager 2, TM_3: Task Manager 3, TM_4: Task Manager 4, TM_5: Task Manager 5, TM_6: Task Manager 6, TM_7: Task Manager 7, TM_8: Task Manager 8, TM_9: Task Manager 9, TM_{10}: Task Manager 10, TM_{11}: Task Manager 11, TM_{12}: Task Manager 12, TM_{13}: Task Manager 13.

w_3: launch_TaskManager, w_4: register_Resource, w_{12}: distribute_Tasks, w_{13}: apply_Resources, w_{14}: assign_Resources, w_{15}: generate_Subtasks, w_{16}: run_Subtasks, w_{21}: monitor_Subtasks, w_{22}: report_Status, w_{24}: monitor_Tasks, w_{25}: report_Status, w_{26}: restart_Task, w_{27}: stop_Tasks.

As you can see from this action transition matrix for the TM component, each action moves the TM from its current state to the next predefined state. For example, the w_3 action transfers from the TM_1 state to the TM_2 state. The behavior of the component is executed sequentially, with each state allowing only one specific action to be performed concurrently or selectively. The sequence of state transitions from the TM_1 state to the TM_{13} state defines the complete workflow of the TM component, and finally, the w_{27} action resets the state of the TM back to the initial state TM_1, forming a loop. This indicates that the TM component performs tasks according to a predefined process and repeats this process periodically.

The action transition matrix extracted from the PEPA model serves as a crucial foundation for analyzing IoT system performance and identifying bottlenecks. It aids in understanding system behaviors, such as the TM's task processing flow, and provides essential inputs for the subsequent section on service flow analysis.

4.2 Service Flow Generation Algorithm

Throughput, utilization, and response time are important measures of system performance, and their solution depends on the steady-state probability distribution of the model's potential CTMC. While throughput and utilization can be computed directly from the steady-state distribution, the computation of response time requires the introduction of the concept of service flow. In this paper, we define a service flow as the necessary execution process required to complete a service.We propose an automatic generation algorithm for service flows and utilize service flow simulation to compute response times, enabling a comprehensive evaluation of the performance of the streaming data processing system embedded in Flink.

As shown in Algorithm 1, the generation of this service flow consists of the following steps:

1. **Initialization.** In this section, the algorithm prepares the required data structures. First, it determines the total number of actions n and then prepares a list F to store all complete action transition paths. Next, it uses queue T to manage the expansion of the action transition path, initially containing only the start action b. In addition, the algorithm creates a selection matrix C to record the selection relations between actions. Finally, a list E is prepared to store the final executable sequence of service flows.
2. **Breadth-first search action transition path exploration.** The goal of this section is to generate all possible action transition paths from start to finish. By using a queue T, the algorithm gradually expands these paths. For the last action of each path, it checks all possible next actions. Extend the path if the next action can be reached directly. If the path reaches the end action e, it is stored in F. To get the shortest response time, the path should remove loops and contain only necessary actions. So when the path search meets when the new action has already appeared in the path, or the found action and the existing action are selected actions, you need to end the search for the next search. At the same time, the algorithm records the selection of action pairs for subsequent processing. The core of this section is divided into two steps:
    a) **Find the successor to the current action.** In the action transition matrix, if there is another action with a value of −1 in the same row where the current action has a value of 1, this indicates that the found action is a subsequent action that can be expanded into a path.
    b) **Determine whether these two actions form a selection action pair in the action transition matrix.** If the same row where the current action has a value of 1 also contains another action with a value of −1, this indicates that the found action is a selection action. The selection action pair is then recorded in the selection matrix C.
3. **Action transition path merging.** The purpose of this phase is to record the relationships of all complete action transition paths in the matrix M. The algorithm traverses each complete path and records the connectivity of the neighboring actions, which are stored as a neighboring linked list. Transform all action transition paths into a directed graph where each edge represents an action transition path. This step provides the basis for subsequent refinement of the service flow.
4. **Service flow refinement using depth-first search.** In this step, the algorithm generates the final executable service flow by eliminating selected and invalid paths. A depth-first search using stack S explores every possible service flow. The algorithm checks for and removes conflicts in the service flow and ensures the legitimacy of the service flow. The feasibility of the service flow is dynamically updated through matrix operations, and finally, the legal service flow is stored in E. The specific steps are as follows:
    a) Find the set of vertices M in the graph with incidence 0 and traverse it.
    b) Take a vertex element w from the set M, where w is a vertex in the graph and also an action in PEPA.
    c) Determine whether the other elements in the set M are right in relation to the selection of w. If so, mark the selected transformation as we and perform step (d). If not, skip to step (f).
    d) Delete vertex w' and its out-degree.
    e) After deleting vertex w' and its out-degree, if the in-degree of the vertex pointing to it becomes 0, mark the vertex pointing to it as w' and skip back to step (d); otherwise skip to step (f).
    f) Determine whether vertex w is a terminating transition, and if so, end this search; otherwise, remove vertex v and its out-degree, add vertex w to the service flow, and jump back to step (b).
    g) Steps (d) through (e) are recursive deletion processes that aim to exclude corresponding selection transitions and other transitions that are dependent on their activation.
5. **Completion.** The purpose of this last step is to return all executable service flow sequences. After the service flow refinement, the service flow list E is output.

Algorithm 1: Service flow generation algorithm

Input: A: Action matrix, b: Initial action, e: Final action
Output: E: Set of executable action sequences
Initialization:
$n \leftarrow$ Total number of actions $F \leftarrow [\ ]$; // List of complete paths
$T \leftarrow [[b]]$; // Exploration queue
$C \leftarrow M_{zero}(n, n)$; // Choice relationship matrix
$E \leftarrow [\ ]$; // Final sequence set
Breadth-First Search (BFS) to Generate Complete Paths:
while *T is not empty* **do**
    $flow \leftarrow T$.Dequeue() $last \leftarrow$ the last action in *flow*
    for *each possible next action next* **do**
        if *A [last][next]* is reachable and $next \notin flow$ **then**
            $new\_flow \leftarrow flow + [next]$
            if $next = e$ **then**
                F.Append(new_flow)
            else
                T.Enqueue(new_flow)
        if *A[last][next] represents a choice relationship* **then**
            the choice relationship matrix C
Path Merging:
$M \leftarrow M_{zero}$ (n, n) **foreach** $flow \in F$ **do**
    for *each consecutive action pair* $(flow[i], flow[i+1])$ **do**
        $M[flow[i]][flow[i+1]] \leftarrow 1$
Depth-First Search (DFS) to Refine Service Flows:
the stack $S \leftarrow [([b], M)]$
while *S is not empty* **do**
    $(sequence, current\_matrix) \leftarrow S$.Pop()
    if *the last action in* $sequence = e$ **then**
        E.Append$(sequence)$
    $M_{zero} \leftarrow$ Findverticeswithzeroin – degree$(current_matrix, e)$
    foreach $w \in M_{zero}$ **do**
        if *w has a choice conflict* **then**
            Select a conflicting vertex *w'*
            Update the matrix $new\_matrix \leftarrow$ Remove$(current_matrix, w')$
            S.Push$(sequence, new_matrix)$
        else
            $new\_sequence \leftarrow sequence + [w]$
            $new\_matrix \leftarrow$ Remove$(current_matrix, w)$
            S.Push$(new_sequence, new_matrix)$
return E

The start action w_1, the end action w_{23} or w_{28}, and the action transition matrix extracted from the PEPA model are used as inputs to the service flow generation algorithm. In the service flow path exploration and generation step, after path expansion and loop detection, we obtain 56 action transition paths from the start action w_1 to the end action w_{23} or w_{28}. In the path recording and transformation relationship construction

phase, the algorithm traverses the complete 56 action transition paths, recording all neighboring connectivity relationships and storing them in the form of an adjacency chain table. Transform all action transition paths into a directed graph where each line represents an action transition path. Fig. 3 shows the directed graph transformed from 56 action transition paths. In the service flow refinement phase, the algorithm will explore each executable path starting from w_1 through a depth-first search to generate the final executable service flow by eliminating selected and invalid paths. Finally, 28 service flows are obtained.

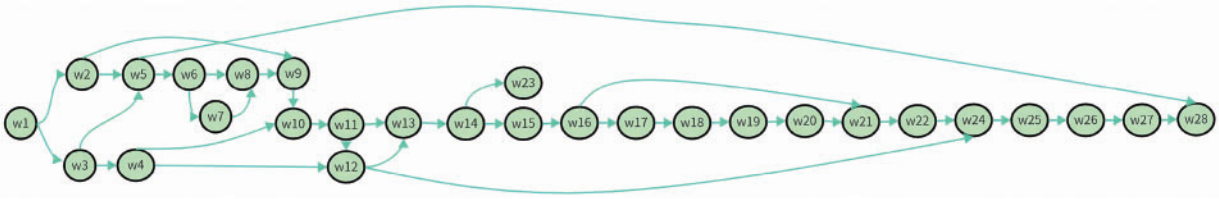

Figure 3: The directed graph transformed from 56 action transition paths

5 Simulation and Results Analysis

5.1 Experimental Setup

To ensure the reproducibility and transparency of our study, we carefully designed a simulation environment that closely reflects real-world conditions of cloud-based streaming data processing for large-scale IoT scenarios. The experimental setup includes modeling and implementation, hardware and software environment, parameter determination, workload scenarios, and data collection and analysis.

For modeling and implementation, we constructed a PEPA model that abstracts the components and interactions within the Flink-based streaming data processing framework. The model captures key elements of the system, including task submission, coordination and scheduling, as well as data processing, to reflect the behavior of the framework under various conditions. A custom solver, implemented in MATLAB, was employed to analyze the steady-state probabilities of each component state. By referencing Table 3, we mapped the action rates (e.g., job submission rate, task execution rate) directly into the PEPA model. This solver was used to compute key performance measures, including response time, throughput, and utilization.

Table 3: Action rate parameters

	Action	Description	Duration	Rate
w1	setting_Environment	Client sets the environment	0.0013	800
w2	launch_JobManager	Client launches the JobManager	0.002	500
w3	launch_TaskManager	Client launches the TaskManager	0.0013	800
w4	register_Resource	TaskManager registers the resources with ResourceManager	0.001	1000
w5	build_JobGraph	Client builds the JobGraph	0.001	1000
w6	send_JobGraph	Client sends the JobGraph to Dispatcher	0.002	500
w7	validate_JobGraph	Dispatcher validates the JobGraph	0.002	500
w8	return_validation	Dispatcher returns the validation results to Client	0.0033	300
w9	submit_Job	Client submits the job to JobManager	0.002	500

(Continued)

Table 3 (continued)

	Action	Description	Duration	Rate
w10	apply_Resources	JobManager applies for resources from ResourceManager	0.002	500
w11	assign_Resources	ResourceManager assigns the resources to JobManager	0.002	500
w12	distribute_Tasks	JobManager distributes the tasks to TaskManager	0.001	1000
w13	apply_Resources	TaskManager applies for resources from ResourceManager	0.002	500
w14	assign_Resources	ResourceManager assigns the resources to TaskManager	0.002	500
w15	generate_Subtasks	TaskManager generates the subtasks	0.0025	400
w16	run_Subtasks	TaskManager runs the subtasks	0.0033	300
w17	reading_Data	TaskManager reads the data	0.0033	300
w18	filtering_Data	TaskManager filters the data	0.0033	300
w19	transforming_Data	TaskManager transforms the data	0.001	1000
w20	output_Result	TaskManager outputs the result	0.002	500
w21	monitor_Subtasks	JobManager monitors the subtasks	0.001	1000
w22	report_Status	TaskManager reports the status to JobManager	0.001	1000
w23	adjust_Resources	ResourceManager adjusts the resources	0.002	500
w24	monitor_Tasks	JobManager monitors the tasks	0.001	1000
w25	report_Status	TaskManager reports the status to JobManager	0.0033	300
w26	restart_Task	JobManager restarts the task	0.0033	300
w27	stop_Tasks	JobManager stops the tasks	0.005	200
w28	output_JobResult	JobManager outputs the job result	0.0033	300

The experiments were conducted on a system equipped with 16 CPU cores and 32 GB of RAM, providing sufficient computational resources to run the MATLAB solver multiple times with varying parameters, such as different numbers of TaskManagers and job submission rates. The operating environment consisted of Ubuntu 20.04 with MATLAB R2023a for running the solver and additional MATLAB scripts for log parsing and data aggregation.

Most parameters, such as action durations and rates, were determined through trial runs in a genuine Internet-based environment, supplemented by official documentation of Apache Flink and consultations with experienced big data engineers. Due to equipment limitations, the quantities of certain components were set based on typical cluster configurations documented by cloud providers and open-source community best practices, as shown in Table 4. The performance metrics evaluated in this study included response time, throughput, and utilization. Response time was defined as the time elapsed from task submission to the return of results, reflecting the system's real-time capabilities. Throughput was measured as the number of tasks processed per second, calculated using the steady-state probability distribution and action rates in the PEPA model. Utilization was defined as the ratio of active components to total components for key actions, providing insights into bottlenecks where resource usage approached saturation.

Table 4: Number of components

Component	Number
Client	1
JobGraph	100
Dispatcher	1
JobManager	50
ResourceManager	1
TaskManager	50
SubTask	150
Data	150

The workload scenarios were designed to evaluate system performance under varying conditions. We began with a baseline job submission rate of 10 jobs per second and incrementally increased it to 200 jobs per second to observe system behavior under higher loads. The number of TaskManagers was varied from 10 to 500 to study scaling behavior and identify resource allocation bottlenecks. Each configuration was run for 10,000 iterations to ensure the system reached a steady state, and the average and variance of key metrics were recorded to improve result reliability.

Finally, data collection and analysis were conducted systematically. Logs generated during each simulation run were collected and processed to compute performance metrics. Simulations were repeated multiple times to ensure consistency, and the results were analyzed using MATLAB scripts to generate both summary statistics (e.g., mean, standard deviation) and visualizations such as probability distribution curves for response time. These detailed steps, combined with references to Tables 3 and 4, ensure that readers can replicate our experiments under similar conditions.

5.2 Response Time Analysis

Response time is the time elapsed from the initiation of a request to the return of the result when the system is running.

Fig. 4 shows the effect of the number of TaskManagers on response time. In the 10–50 TaskManager range, the response time decreases as the number of TaskManagers increases due to enhanced parallel computing power. More TaskManagers provide additional slots for task execution, thereby accelerating data processing. Specifically, as the number of TaskManagers increases from 10 to 20, the overall performance improves by 18.49%, while the improvement slows to 6.06% when increasing from 20 to 50. However, this performance improvement does not scale linearly.

In the 50–80 TaskManager range, the decline in response time slows further, with the curves essentially overlapping, indicating that the benefits of adding TaskManagers gradually diminish. For example, the improvement is negligible at −0.1% when increasing from 50 to 80 TaskManagers. After that, in the 100–500 TaskManager interval, the response time shows a significant upward trend. As the number of TaskManagers continues to increase, the response time instead rises dramatically. This is mainly due to the fact that system resources, especially the capabilities of the ResourceManager, become a bottleneck. A limited number of ResourceManagers cannot effectively manage too many TaskManagers, leading to increased competition for resources. For instance, the performance deteriorates by −2.6% when increasing from 80 to 100 TaskManagers, and the degradation becomes more pronounced as the number of TaskManagers grows

further, with changes of −27.86% (100 to 200), −14.88% (200 to 300), −12.56% (300 to 400), and −15.18% (400 to 500).

A large number of TaskManagers will spend more time requesting (apply_Resources) and waiting for the ResourceManager to allocate resources (assign_Resources). This time spent requesting and waiting offsets or even slows down the performance gains from parallel processing, ultimately leading to an increase in overall response time. To achieve optimal performance in the stream processing system, it is crucial to select an appropriate number of TaskManagers based on system resources and the capabilities of the ResourceManager. This careful selection ensures the full benefits of parallel processing while avoiding the negative impacts of resource competition.

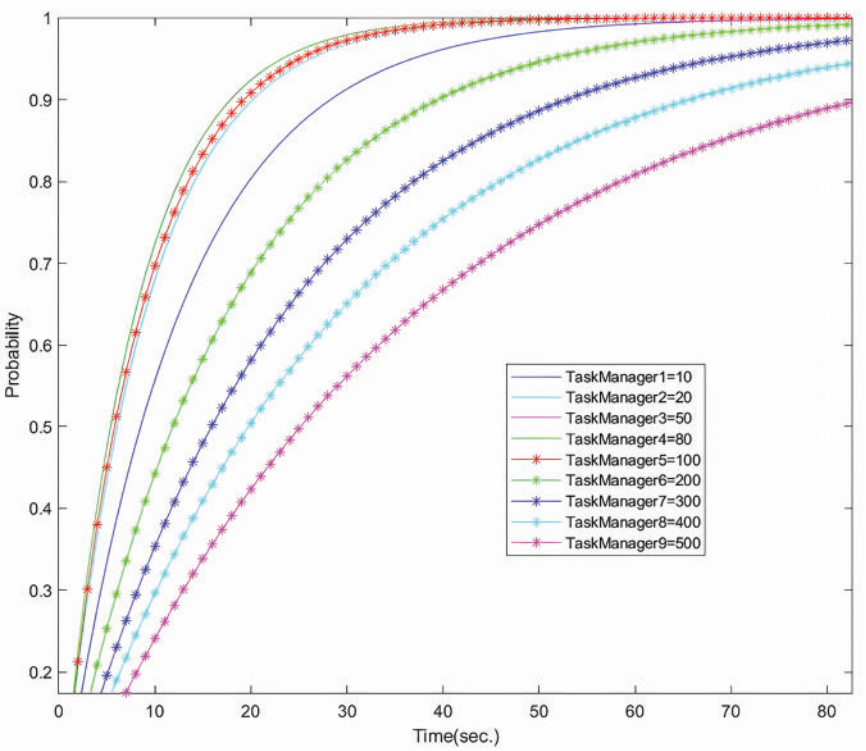

Figure 4: Number of TaskManager vs. response time

Fig. 5 illustrates the effect of the JobGraph build rate of the Client node on the response time. It is clear that the response time decreases as the rate increases. Specifically, increasing the JobGraph build rate from 10 to 100 yields an overall improvement in response time of approximately 3.19%. Further increasing the rate from 100 to 200 results in a similar improvement of 3.21%. However, increasing the rate from 200 to 500 leads to a negligible change, with the improvement being only −0.16%. This indicates that while increasing the efficiency of this operation reduces the response time of the system, beyond a certain rate, the performance gains become negligible, as reflected in the almost identical probability distribution curves for 200 and 500.

This phenomenon clearly illustrates that there is a critical point of diminishing returns even for optimization of critical components. Beyond this point, continuing to improve the performance of this component will no longer result in significant overall system performance improvements. This bottleneck occurs mainly because the execution efficiency of other components within the system becomes a new constraint, limiting further improvements in overall performance. Similarly, as shown in Fig. 6, the rate at

which TaskManager processes tasks has a comparable impact on response time. Specifically, increasing the number of subtasks from 10 to 100 yields an improvement of 12.49%, but further increases from 100 to 200 and 200 to 500 result in diminishing returns, with improvements of only 2.80% and 0.12%, respectively. This reinforces the observation that beyond a critical point, performance gains become negligible due to new bottlenecks elsewhere in the system.

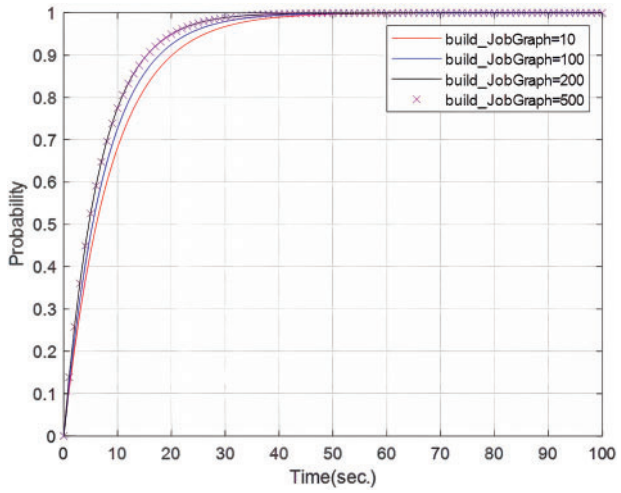

Figure 5: Rate of building JobGraph vs. response time

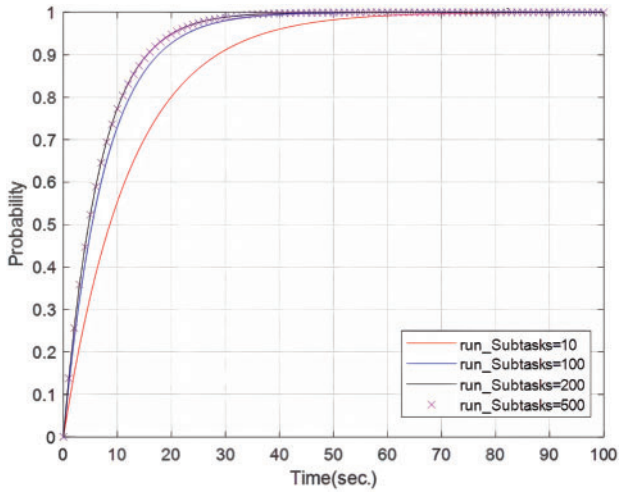

Figure 6: Rate of running Subtasks vs. response time

Fig. 7 illustrates the impact of the task submission rate of the Client component and the task execution rate of the TaskManager component on the response time. Initially, the response time decreases slightly as the Client component's task submission rate increases. However, since task execution can only begin after the Client's task scheduling is complete and the TaskManager component is fully initialized, an increase in the Client's commit rate does not translate into an increase in the TaskManager's execution speed.

This leads to a bottleneck, as evidenced by the fact that the probability distribution curve for a submission rate of 100 and a task execution rate of 10 is very close to the probability distribution curve for a

submission rate of 200 and a task execution rate of 10, with an overall improvement of only −0.11%. Similarly, when the task execution rate increases from 10 to 100 while keeping the submission rate at 200, the overall performance of the system improves significantly, with an improvement of 9.53%. Further increasing the task execution rate from 100 to 200 results in an even greater improvement of 21.58%, highlighting the importance of optimizing the TaskManager's execution efficiency. However, when the task execution rate increases from 200 to 500, the improvement becomes marginal, with an increase of only 0.42%, indicating diminishing returns as the bottleneck shifts to other parts of the system. Therefore, in cloud-based computing subsystems, performance advantages can be gained by improving task processing efficiency alone. However, substantial overall system performance gains can only be obtained by addressing a limited number of bottlenecks.

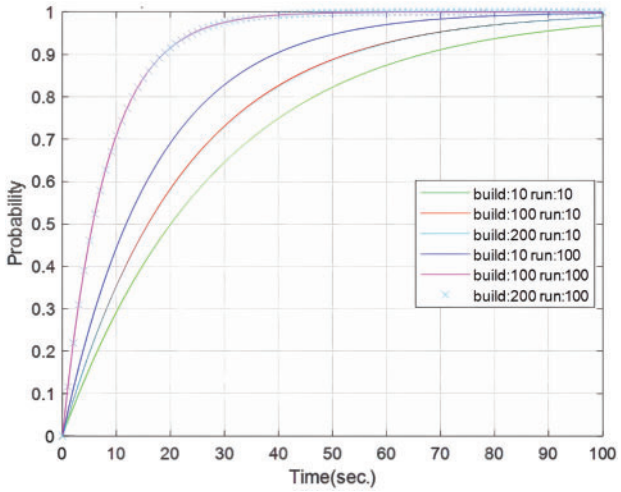

Figure 7: Rate of TaskManager running Subtasks and rate of Client building JobGraph vs. response time

5.3 Throughput Analysis

Throughput indicates the number of tasks that the system can handle per unit of time. In the PEPA model, throughput can be calculated by analyzing the steady-state probability distribution and action rates of the model. Specifically, the throughput of component X performing action a can be expressed as $Throughput(X, a) = P(X) \cdot ra$, where $P(X)$ is the steady-state probability that component X is in an active state, and ra is the rate of action a. Fig. 8 shows the throughput of the streaming data processing system components embedded in Flink under different workloads. These data were obtained through simulation experiments. The number of training rounds was 10,000 and the parameters listed in Tables 3 and 4 were used.

In cloud-based computing environments where streaming processing systems are deployed, the throughput of each component rises with the workload until it reaches a saturation point. This stabilization indicates that the component has reached its maximum processing capacity and that continuing to increase the number of jobs will not improve throughput. For example, in Fig. 8, the throughput of the JobManager initially rises as the rate of submitting jobs (rsubmit_Job) increases. However, throughput saturates when the number of applications exceeds 10. This indicates that the JobManager has reached a bottleneck in processing capacity, most likely due to rate-limiting processing in later stages such as monitor_Tasks or report_Status. Depending on the PEPA model, increasing the value of rmonitor_Tasks or rreport_Status may improve the throughput of the JobManager. Analyzing the throughput trends of different components can provide insight into the system's performance bottlenecks and guide optimization efforts. By identifying the components

that reach capacity limits first, targeted improvements can be made to increase the throughput and efficiency of the overall system.

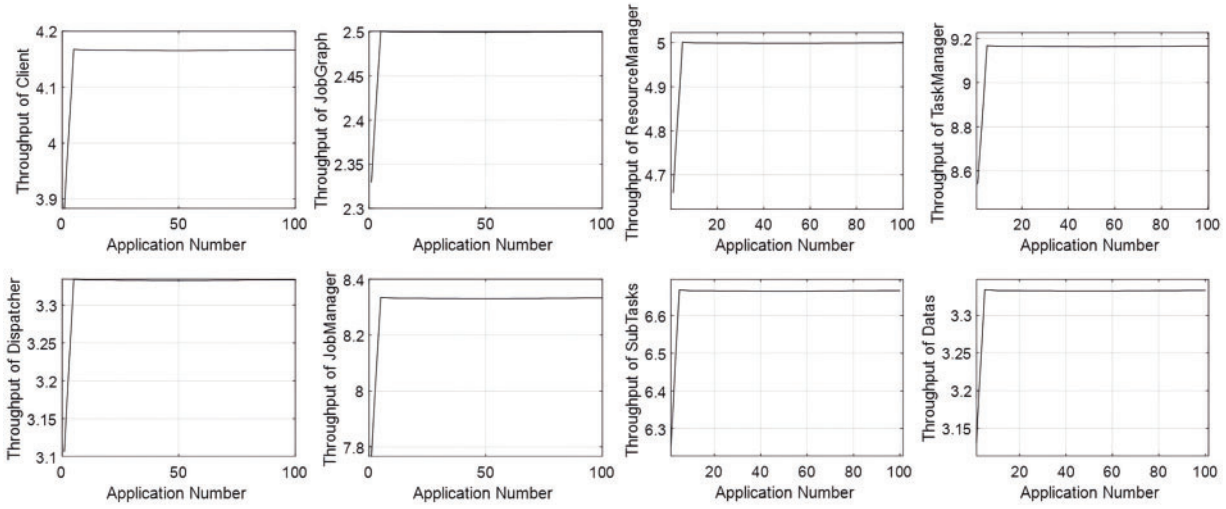

Figure 8: Application number vs. throughput

Similarly, this throughput profile is a direct reflection of the cloud platform's data processing capabilities. As throughput approaches its peak, the cloud's data processing speed is maximized to provide optimal service performance for IoT applications. For IoT systems, the high throughput of cloud platforms means the ability to process massive amounts of streaming data faster, providing real-time analytics. The real-time performance and scalability of IoT systems can be significantly improved by accurately controlling the load on each component of the cloud platform to keep it in the optimal throughput zone.

5.4 Utilization Analysis

The utilization rate represents the ratio of the number of a given component to the number of all the components involved in the execution of the sequential process of this component during the operation of the system. Specifically, the utilization of component P is given by the formula $Utilization\_P = x[P]/s(P)$. Here, $Utilization\_P$ represents the utilization of component P, $x[P]$ denotes the number of component P in the steady state, and $s(P)$ is the total number of sequential components in the process where component P is located.

Similar to throughput, utilization can be calculated by analyzing the steady-state probability distribution of the PEPA model. $x[P]$ can be determined from the steady-state probability distribution and the number of components P in each state. For example, if the number of components$P$ under state S_i is $n_i(P)$ and the steady-state probability of state S_i is π_i, then $x[P]$ can be expressed as: $x[P] = \sum_i (n_i(P) \cdot \pi_i)$, where the summation is performed over all possible states. Combined with $s(P)$, the utilization rate u_P of component P can be calculated. Based on the performance evaluation method in Section 4, it can solve the steady-state probability distribution and then calculate the utilization of each component of the system. In the previous section, the PEPA model of the streaming data processing system based on the Flink platform has been established, which describes the interaction and resource competition among components. To gain a deeper understanding of the system performance bottleneck, this section will analyze the utilization of the model.

Fig. 9 shows the trend of the utilization of each operation in the streaming data processing system embedded in Flink. For example, as can be observed in Fig. 9, the utilization of the JobManager approaches saturation (e.g., 90%) when the number of submitted applications reaches about 10. Similarly, the utilization of the TaskManager reaches a similar saturation level when the number of applications reaches about 20. This suggests that under the current configuration, JobManager is more likely to become a bottleneck, limiting the overall scalability of the system. Combined with the PEPA model, we can analyze the reasons for the rapid increase in JobManager utilization. For example, if the rate of the monitor_Tasks action in the PEPA model is low, it may cause the JobManager to consume too much time in monitoring tasks, which will push up its utilization. Therefore, by analyzing the utilization trends of various components in cloud-based stream processing systems, this study identifies potential performance bottlenecks and suggests directions for system optimization.

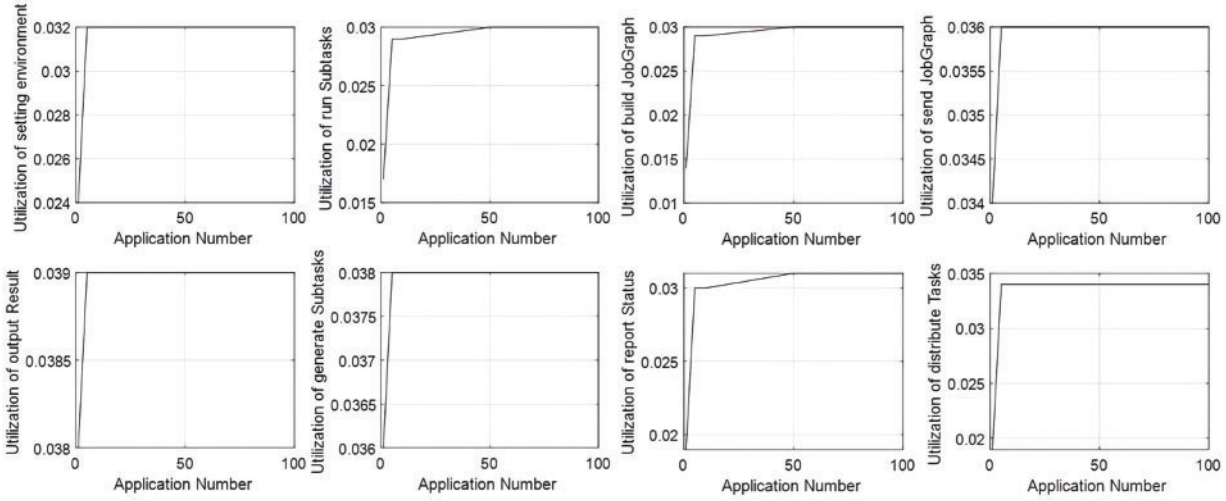

Figure 9: Application number vs. utilization

In a cloud-based computing environment, the utilization of various actions likewise rises as the workload increases until it reaches a saturation point. This utilization curve is a direct reflection of the computing power and efficiency of the cloud platform. As utilization nears its peak, the cloud's data processing speed is maximized to provide optimal response performance for IoT applications. For IoT systems, this means that IoT devices get the fastest feedback for data analysis and decision-making when the utilization of the cloud platform's processing units is in the optimal range. Therefore, by monitoring and optimizing the utilization of each component in the cloud, the real-time performance and scalability of the entire IoT system can be effectively improved.

6 Conclusion

In this paper, we model and evaluate the performance of a cloud-based streaming data processing system to address the challenges of real-time streaming data processing in large-scale IoT applications. A system model is constructed containing core components such as Client, JobManager, and TaskManager. A service flow generation algorithm is proposed based on which key performance metrics such as response time, throughput, and resource utilization of the system are calculated. This provides a new perspective on performance optimization of IoT systems for large-scale real-time streaming data processing.

The main contributions of this work are as follows. First, we provide a novel system model that captures the key components and interactions of a cloud-based streaming data processing system, offering a structured framework for analyzing and optimizing similar systems. Second, we propose a service flow generation algorithm that enables the calculation of critical performance metrics, such as response time, throughput, and resource utilization, which serve as a foundation for performance evaluation and optimization. Third, we address the challenges of large-scale real-time streaming data processing in IoT applications by demonstrating how the proposed model and algorithm can be applied to identify bottlenecks and improve system efficiency. By providing these contributions, this work helps the research community better understand and optimize cloud-based IoT systems for real-time data processing.

However, there are still some limitations in the current work that need to be addressed. First, the proposed system model simplifies certain aspects of real IoT environments, such as the dynamic and stochastic nature of IoT data streams and the potential impact of network failures or delays. This may lead to differences between the modeled results and real-world performance. Second, the service flow generation algorithm resolves conflicts in action transitions using a deterministic approach, which may not fully capture the probabilistic or adaptive behaviors of real-world systems. Third, the focus of this study is primarily on computational performance, while the performance of communication systems, which plays a critical role in IoT scenarios, is not explicitly considered.

Future research will focus on extending the existing PEPA model to develop a more comprehensive system model. This model will integrate both computational and communication systems to more accurately simulate real IoT environments. Furthermore, in response to the special performance requirements in unmanned scenarios, such as high concurrency, low latency, and high reliability, the optimization of network latency and data transmission rate can be another important direction to achieve performance breakthroughs in practical applications. Additionally, considering that BFS and DFS are static algorithms, future work could explore the use of AI-based algorithms to find more efficient and adaptive solutions, addressing the challenges posed by dynamic and complex IoT environments. To further enhance the robustness and practicality of the service flow generation algorithm, future research could also investigate more sophisticated conflict-resolution strategies, such as incorporating resource utilization metrics or system performance indicators. Moreover, extending the algorithm to better handle the dynamic and stochastic nature of real IoT systems, for example, through periodic updates to the action matrix or probabilistic modeling, could improve its adaptability in real-world scenarios.

Acknowledgement: The authors appreciate it that this research is funded by the Joint Project of Industry-University-Research of Jiangsu Province.

Funding Statement: This research is funded by the Joint Project of Industry-University-Research of Jiangsu Province (Grant: BY20231146).

Author Contributions: Feng Zhu: Methodology, Software, Writing—Reviewing and Editing; Kailin Wu: Software, Visualization, Writing—Original Draft; Jie Ding: Conceptualization, Supervision. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: Data available on request from the authors.

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest to report regarding the present study.

References

1. Hanif M, Kim E, Helal S, Lee C. SLA-based adaptation schemes in distributed stream processing engines. Appl Sci. 2019;9(6):1045. doi:10.3390/app9061045.
2. Mao Y, Chen Z, Zhang Y, Wang M, Fang Y, Zhang G, et al. StreamOps: cloud–native runtime management for streaming services in bytedance. Proc VLDB Endow. 2023;16(12):3501–14. doi:10.14778/3611540.3611543.
3. Xhafa F, Kilic B, Krause P. Evaluation of IoT stream processing at edge computing layer for semantic data enrichment. Fut Gener Comput Syst. 2020;105(11):730–6. doi:10.1016/j.future.2019.12.031.
4. Henning S, Hasselbring W. Benchmarking scalability of stream processing frameworks deployed as microservices in the cloud. J Syst Softw. 2024;208(12):111879. doi:10.1016/j.jss.2023.111879.
5. Walia GK, Kumar M, Gill SS. AI-empowered fog/edge resource management for IoT applications: a comprehensive review, research challenges and future perspectives. IEEE Commun Surv Tutor. 2024;26(1):619–69. doi:10.1109/COMST.2023.3338015.
6. Kumar M, Walia GK, Shingare H, Singh S, Gill SS. AI-based sustainable and intelligent offloading framework for IIoT in collaborative cloud–fog environments. IEEE Transactions on Consumer Electronics. 2024;70(1):1414–22. doi:10.1109/TCE.2023.3320673.
7. Long C, Cao Y, Jiang T, Zhang Q. Edge computing framework for cooperative video processing in multimedia IoT systems. IEEE Trans Multimed. 2017;20(5):1126–39. doi:10.1109/TMM.2017.2764330.
8. Čolaković A. IoT systems modeling and performance evaluation. Comput Sci Rev. 2023;50(2):100598. doi:10.1016/j.cosrev.2023.100598.
9. Moreno-Vozmediano R, Montero RS, Huedo E, Llorente IM. Efficient resource provisioning for elastic cloud services based on machine learning techniques. J Cloud Comput. 2019;8(1):1–18.
10. Song J, Huo H, Li T, Chu L. A dynamic source tracing method for food supply chain quality and safety based on big data. Discrete Dyn Nat Soc. 2022;2022(1):6385201. doi:10.1155/2022/6385201.
11. Ding J, Sun H, Chen X, Fang H. Response time analysis of a manufacturing supply chain with performance evaluation process algebra. Comput Ind Eng. 2022;167(4):108043. doi:10.1016/j.cie.2022.108043.
12. Hillston J. Compositional approach to performance modelling. Cambridge, UK: Cambridge University Press; 1994.
13. Chen L, Zheng J, Okamura H, Dohi T. Performance evaluation of a cloud datacenter using CPU utilization data. Mathematics. 2023;11(3):513. doi:10.3390/math11030513.
14. Liu S, Yu Z. Modeling and efficiency analysis of blockchain agriculture products E-commerce cold chain traceability system based on Petri net. Heliyon. 2023;9(11):e21302. doi:10.1016/j.heliyon.2023.e21302.
15. Kabashkin I. Model of multi criteria decision-making for selection of transportation alternatives on the base of transport needs hierarchy framework and application of Petri net. Sustainability. 2023;15(16):12444. doi:10.3390/su151612444.
16. Lei Y, Mu H. Analysis and optimization of a Stochastic Petri net for air–rail intermodal transportation. PLoS One. 2024;19(7):e0307647. doi:10.1371/journal.pone.0307647.
17. de Souza Neto JB, Moreira AM, Vargas-Solar G, Musicante MA. A two-level formal model for Big Data processing programs. Sci Comput Program. 2022;215(3):102764. doi:10.1016/j.scico.2021.102764.
18. Chen G, Jiang T, Wang M, Tang X, Ji W. Design and model checking of timed automata oriented architecture for Internet of thing. Int J Distrib Sens Netw. 2020;16(5):1550147720911008. doi:10.1177/1550147720911008.
19. Daszczuk WB. Modeling and verification of asynchronous systems using timed integrated model of distributed systems. Sensors. 2022;22(3):1157. doi:10.3390/s22031157.
20. Chen X, Ding J, Lu Z, Zhan T. An efficient formal modeling framework for hybrid cloud–fog systems. IEEE Trans Netw Sci Eng. 2020;8(1):447–62. doi:10.1109/TNSE.2020.3040215.
21. Ding J, Wang R, Chen X, Ge YE. Exploring auto-generation of network models with performance evaluation process algebra. IEEE Access. 2018;6:42971–83. doi:10.1109/ACCESS.2018.2862390.
22. Wang H, Laurenson DI, Hillston J. A general performance evaluation framework for network selection strategies in 3G–WLAN interworking networks. IEEE Trans Mob Comput. 2012;12(5):868–84. doi:10.1109/TMC.2012.60.
23. Wu X, Hillston J, Feng C. Availability modeling of generalized *k*–Out–of–*n*: G warm standby systems with PEPA. IEEE Trans Syst Man Cybern: Syst. 2016;47(12):3177–88. doi:10.1109/TSMC.2016.2563407.

24. Chen X, Wang L. Exploring fog computing-based adaptive vehicular data scheduling policies through a compositional formal method–PEPA. IEEE Commun Lett. 2017;21(4):745–8. doi:10.1109/LCOMM.2016.2647595.
25. Liu P, Wang R, Ding J, Yin X. Performance modeling and evaluating workflow of ITS: real-time positioning and route planning. Multimed Tools Appl. 2018;77(9):10867–81. doi:10.1007/s11042-017-5364-8.