An Adaptive Mini-Batching Strategy for Reliable Streaming Data Delivery in Real-Time

Han Wu, *Member, IEEE*, Zhihao Shang, *Member, IEEE*, Huaming Wu, *Senior Member, IEEE*, and Katinka Wolter, *Member, IEEE*

***Abstract*—Modern applications show an increasing demand for continuously processing massive data streams in real time. Mini-batching technique is commonly used for transporting streaming data across these applications. However, although a larger mini-batch size increases throughput, it also raises the end-to-end latency and easily violate the latency constraint required by real-time applications. While existing work mostly studies this problem at the computation stage, this work explores it in the streaming data transportation stage. We show that selecting a proper mini-batch size is essential for efficient streaming data delivery. We further identify the shortcomings of existing latency measurements and introduce new quality-of-service (QoS) metrics: latency violation rate, timely throughput, message loss, and duplicate rate. To address the challenge of mini-batching under varying network conditions, we first develop prediction models for the proposed QoS metrics and then adaptively adjust the mini-batch size based on these predictions. In our experiments, the random forest regressor achieves an R^2 of 0.99 for performance metrics, and the multilayer perceptron achieves mean absolute error below 0.02 for reliability metrics. Using these predictions, the proposed adaptive strategy continuously updates the mini-batch size according to the observed network state. In a Kafka testbed with network packet loss rate approaching 25%, our strategy improves timely throughput by up to 30% compared with empirical mini-batch size selection. The results confirm the effectiveness of the adaptive mini-batching approach.**

***Index Terms*—Adaptive batching, machine learning, performance, quality-of-service (QoS), reliability, streaming data.**

NOMENCLATURE

Main Notations

$M\{m_1, m_2, \dots\}$	Set of messages in streaming data.
$B\{b_1, b_2, \dots\}$	Set of mini-batches. Each b_j contains $\lvert b_j\rvert$ number of messages.
β_{ij}	Batching latency of message i in batch j.
L_p	Batching latency.
L_r	Transmission latency.
L	End-to-end latency.
ζ_L	User-defined latency constraint.
η_v	Latency violation rate.

Received 30 May 2025; revised 16 November 2025; accepted 19 April 2026. Date of publication 23 April 2026; date of current version 12 May 2026. This work was supported in part by the University of Southampton, and the Emerging Frontiers Cultivation Program of Tianjin University Interdisciplinary Center, in part by Tianjin Natural Science Foundation Project under Grant 25JCYBJC01540, in part by Tianjin Municipal Science and Technology Major Program under Grant 25ZXZSSS00390, and in part by Tianjin Future Industry Science and Technology Major Project under Grant 25ZXWCSY00290. Associate Editor: S.-Y. Hsieh. *(Corresponding authors: Zhihao Shang; Huaming Wu.)*

Han Wu is with the School of Electronics and Computer Science, University of Southampton, SO17 1BJ Southampton, U.K. (e-mail: h.wu@soton.ac.uk).

Zhihao Shang is with the School of Information Engineering, Zhengzhou University, Zhengzhou 450000, China (e-mail: zhihao.shang@zzuli.edu.cn).

Huaming Wu is with the Center for Applied Mathematics, Tianjin University, Tianjin 300072, China (e-mail: whming@tju.edu.cn).

Katinka Wolter is with the Department of Computer Science, Freie Universität Berlin, 14195 Berlin, Germany (e-mail: katinka.wolter@fu-berlin.de).

Digital Object Identifier 10.1109/TR.2026.3687124

I. INTRODUCTION

DRIVEN by the social media phenomenon, the massive deployment of Internet of Things (IoT) devices and the expansion of 5G networks, modern Big Data applications are now evolving from batch-oriented to stream-oriented [1], [2]. In batch-oriented systems (e.g., Hadoop), data is collected and stored in databases to be periodically processed in large chunks [3]. In contrast, stream-oriented systems provide insights from data as soon as it is generated. Common examples include sensor readings from weather monitoring devices, user activities collected on websites, and payment requests from mobile devices. Such data streams form an unbounded and continuously growing dataset, commonly referred to as *streaming data*. A key challenge is, the value of streaming data decays rapidly, which requires processing in real-time [4], [5].

To handle streaming data with strict timeliness requirements, many stream processing platforms have been developed in the past decade [3], [6], [7], [8]. A common feature across these systems is the usage of *mini-batches*, also referred to as *microbatches* in some studies [9], [10], [11], [12]. Mini-batching aggregates multiple messages and sends them together, which reduces request overhead and improves efficiency compared with transmitting each message individually. However, these studies mostly examine mini-batching in the computation stage, whereas the impact on streaming data transportation, which concerns delivering data from its source to the processing servers, has received far less attention.

In this work, we examine how mini-batching affects data transportation under varying network conditions, such as high network latency and packet loss. Increasing the mini-batch size can improve throughput, but it also causes end to end latency to rise rapidly [13], [14], [15], which means a portion of messages no longer arrives within the required timeliness bound. Therefore selecting an appropriate mini-batch size is essential for reliable and timely streaming data delivery. Our analysis

shows that commonly used performance metrics, such as mean end-to-end latency, fail to capture the waiting time accumulated while constructing a mini-batch and are thus insufficient for evaluating transportation behavior. These limitations become more pronounced when the network connection is unstable, making such metrics inadequate for optimizing mini-batching during streaming data transportation.

To address the above challenges, we propose an adaptive mini-batching strategy. The main idea is to reactively adjust the size of the mini-batch when the streaming data or network condition changes. Aiming at building a versatile strategy, we consider both spatial and temporal batch sizes, which have not been studied concurrently in other works. In combination with the concept of service-level objective (SLO), new quality-of-service (QoS) metrics are proposed to evaluate the timeliness and reliability of streaming data delivery. Traditional metrics, such as mean end-to-end latency and raw throughput, are inadequate for mini batching because they ignore two fundamental behaviors observed in streaming data transportation. First, constructing mini-batches introduces significant variability in message waiting time, and our measurements show that the latency distribution becomes highly skewed as the batch size increases; in such cases, a low mean latency can coincide with a large fraction of messages arriving too late to be useful. Second, unstable network conditions cause message loss, retransmissions, and occasional duplication, none of which are reflected in traditional latency centric metrics. To overcome these limitations, we introduce four QoS metrics tailored to streaming data transportation: 1) *latency violation rate*, which measures the proportion of messages that exceed the user defined latency constraint and therefore quantifies timeliness at distribution level rather than through averages; 2) *timely throughput*, which captures the effective throughput contributed only by messages that meet the latency constraint; 3) *message loss rate*, which characterizes the likelihood of message drops under unstable network connection; 4) *duplicate rate*, which captures the effect of repeated message delivery when acknowledgments are delayed or lost. These metrics collectively provide a more complete representation of timeliness and reliability.

The key to achieving adaptive mini-batching is to predict the proposed QoS metrics given the relevant mini-batch configurations and network condition parameters. In our work, we use machine learning-based techniques, random forest regressor (RFR) and multilayer perceptron (MLP) to construct a prediction model. Given a mini-batch configuration and network conditions, our model predicts the QoS metrics with high accuracy and facilitates the study of the impact of mini-batch size. Using these predictions, we design an adaptive mini-batching strategy that periodically monitors network status and selects the batch size that maximizes timely throughput while respecting user-defined constraints. The strategy selects mini-batch sizes by combining the predicted QoS metrics with the current network condition. It evaluates feasible spatial and temporal batch-size configurations using the prediction models and chooses the setting that maximizes timely throughput while respecting user-defined latency constraints. The strategy is then evaluated on a Kafka based streaming testbed, where we integrate a QoS monitor to observe message latency distributions and conduct experiments under both stable and faulty network conditions. Experimental results show that the adaptive strategy achieves higher timely throughput, with improvements of up to 30% compared with empirical batch size selection methods.

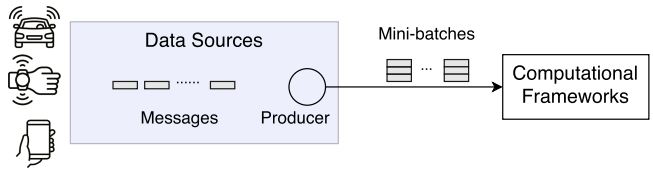

Fig. 1. Pipeline of streaming data transportation.

A preliminary version of this article appeared as [16]. In this significantly extended version, we make the following new contributions.

1) We extended the streaming testbed by integrating a new QoS monitor module, which approximates the distribution of end-to-end latency using kernel density estimation (KDE). The new module thoroughly monitors the impact of changing the mini-batch size on the latency. It is expected to be utilized in other performance-monitoring dashboards.
2) We have replaced the previous performance prediction approach with a new machine learning-based model. The previous model based on distribution fitting is clumsy in comparison. The new model adopts an RFR. It is lightweight and achieves higher precision.
3) New experiments and analyses are presented with the extended approach. We validate the effectiveness of our mini-batching method by demonstrating the experimental results under different SLOs.

The rest of this article is organized as follows. Section II highlights the common design principles of mini-batching for transporting streaming data, as well as the performance and reliability problems. In Section III we describe the proposed QoS metrics and input features for prediction. The Kafka testbed and QoS monitor are introduced in Section IV. In Section V, we describe the prediction models. Section VI gives the adaptive mini-batching strategy and it is experimentally evaluated in Section VII. Section VIII summarizes related work. Finally, Section IX concludes this article.

II. Background and Problems

A. Mini-Batching for Transporting Streaming Data

In general, the pipeline of streaming data transportation follows the example in Fig. 1. Sources, such as autonomous vehicles, wearable devices, and mobile apps, continuously generate messages, each capturing the latest event (e.g., a transaction request, a location update, or a web server access record). These messages are typically small and arrive at high frequency; for instance, vehicle location updates are about 300 bytes and can be produced thousands of times per second in large-scale systems [17]. A producer process collects these messages and

TABLE I
COMPARISON OF MESSAGE LOSS RATE AND TIMELY THROUGHPUT

	Adaptive mini-batching	Empirical method	Bs=1	Bs=10
R_l	18.96%	12.63%	71.0%	3.9%
χ'_P	42.7MB/s	33.0MB/s	5.14MB/s	27.3MB/s

transmits them to downstream computational frameworks. Sending each message individually incurs substantial overhead from system calls and hardware interruptions. To reduce this cost, the producer aggregates multiple messages into a mini-batch and transmits them together. This mini-batching mechanism greatly improves throughput but increases mean end-to-end latency [10], [13], where end-to-end latency refers to the time taken for a message to travel from the source to the computational framework.

Although the same batching rule is often referred to as a "micro batch" in studies that focus on the computation stage [13], [14], [15], [18], we use the term "mini batch" to distinguish our focus on the transportation stage. In this stage, mini-batching introduces additional tradeoffs: 1) a larger batch improves throughput but increases waiting time inside the producer; and 2) under unstable networks, such as mobile devices switching between base stations, larger batches risk missing timeliness constraints because their transmission is more sensitive to retransmissions and congestion. These tradeoffs motivate our focus on the performance and reliability challenges that arise specifically during streaming data transportation.

We observed that the size of the mini-batch has a significant impact both on the throughput and end-to-end latency [15]. Therefore choosing the proper batch size is the key to achieving the optimal tradeoff between throughput and latency. Previous works either define the batch size in terms of the batching interval [13], [14], [19], or in terms of data volume [10], [20]. In this article, we consider both by introducing two parameters for controlling the size of the mini-batch, as described below.

Spatial batch size controls the maximum number of messages that can be buffered in each mini-batch. Some work [10] controls the maximum volume of mini-batch in bytes, but this complicates the system model. This is because disparate applications generate different sizes of messages and the hardware performance also differs a lot. When a mini-batch size is the optimal choice in one system, the absolute number in bytes can not be generalized across other heterogeneous systems. Thus, for simplicity, we use the maximum number of messages per mini-batch as the unified *spatial batch size* parameter.

Temporal batch size limits the maximum time interval to construct a mini-batch. The starting time is when the first message arrives at the mini-batch, and the batching is closed when its waiting time reaches this upper limit. This mini-batching parameter is equivalent to the concept of batch interval in other works [13].

In practice, the mini-batch is closed when either condition is met. Algorithm 1 provides a formal description of this process. To the best of the authors' knowledge, no existing work jointly considers both spatial and temporal batching mechanisms, and Table II summarizes the absence of a joint investigation.

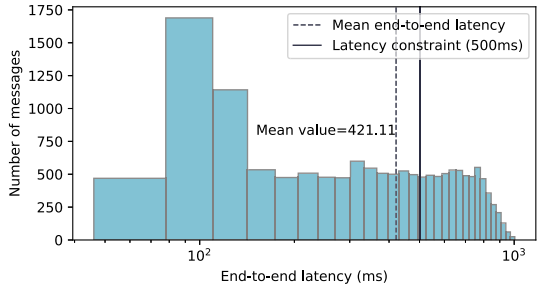

Fig. 2. End-to-end latency histogram with spatial batch Size = 6 and temporal batch size = 1000. Although the mean latency satisfies the constraint, a large fraction of messages violate it.

B. Performance and Reliability Problems

The goal of this work is to efficiently and reliably transport streaming data in real time by adaptively selecting the mini-batch size. A first challenge lies in modeling the combined spatial–temporal batching mechanism. Queueing theory is not applicable because it assumes service times are independent of batch size [21], which contradicts our observations. Existing analytical models consider only one batching parameter [10], [13], [14], [19], and therefore cannot fully capture batching behavior. Additional challenges arise from the diversity of streaming workloads and the variability of network conditions.

Because streaming data originates from heterogeneous sources, real-time requirements differ significantly across applications. Fraud detection may require message latencies of a few hundred milliseconds, whereas weather monitoring can tolerate delays of several seconds. Developers therefore specify application-level SLOs [22], such as minimum throughput and a maximum acceptable latency. Performance metrics (e.g., producer throughput and end-to-end latency) must reflect these SLOs.

However, prior work typically evaluates only the mean end-to-end latency [10], [13], [23], which is unreliable when batching is used. As the batch size increases, latency variance grows rapidly. Fig. 2 illustrates this problem: although the mean latency (421.11 ms) is below the 500 ms SLO, 39.7% of messages exceed the constraint. Relying solely on the mean therefore obscures the prevalence of stale messages.

A further challenge is the complexity of network environments in which streaming data are transported. Mobile and IoT devices often transmit data over wireless links, resulting in unstable connections [24], [25]. High packet loss rates directly lead to message loss [26]. Although modern stream processing systems provide exactly-once semantics for server-side reliability [3], [6], [8], their robustness to network failures remains less understood. Our experiments show that mini-batch size strongly influences the probability of successful delivery, making batch-size selection crucial for reliability under adverse network conditions.

TABLE II
COMPARISON WITH OTHER ADAPTIVE MINI-BATCHING STRATEGIES FOR STREAMING DATA PROCESSING

Methods	Focus	Spatial batch size	Temporal batch size	Comprehensive latency metric	Poor network study	Fault injection	Focus on message delivery
Remark: "p" and "r" denote performance and reliability respectively.							
Das et al. [13] (2014)	p & r	✗	✓	✗	✗	✗	✗
Zhang et al. [19] (2016)	p & r	✗	✓	✗	✗	✗	✗
Cheng et al. [14] (2018)	p	✗	✓	✗	✗	✗	✗
Mai et al. [45] (2018)	p	✓	✗	✗	✗	✗	✗
Stein et al. [10] (2020)	p	✓	✗	✗	✗	✗	✓
Abdelhamid et al. [11] (2020)	p & r	✗	✓	✗	✗	✗	✗
Garcia et al. [12] (2022)	p	✓	✓	✗	✗	✗	✗
Garcia et al. [55] (2023)	p	✓	✗	✗	✗	✗	✗
Deepthi et al. [18] (2024)	p	✗	✓	✗	✓	✗	✓
Leonarczyk et al. [56] (2025)	p	✗	✓	✗	✗	✗	✗
Our approach	p & r	✓	✓	✓	✓	✓	✓

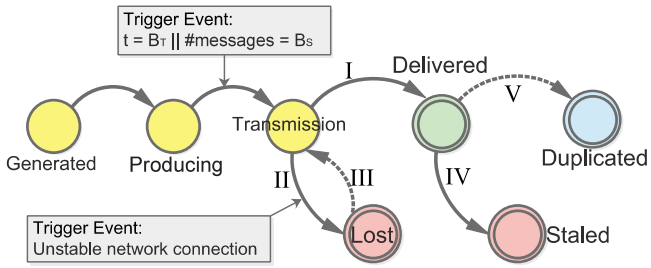

Fig. 3. Message state transition during streaming data transportation.

Algorithm 1: Mini-Batch Formation.

```
while messages arrive do
    receive next message m_i and append it to b_j;
    if |b_j| = B_S or max{β_ij} = B_T then
        close b_j and send it for transmission;
        increment j ← j + 1 and create new batch b_j;
    end
end
```

III. PROBLEM FORMULATION

A. Message State Machine Model

For a better understanding of the problems, we use the state machine model depicted in Fig. 3 to describe all possible cases while delivering a message. Let $M\{m_1, m_2, \dots\}$ be the set of messages in the incoming streaming data, and $B\{b_1, b_2, \dots\}$ denote the set of mini-batches that hold these messages. Once a message m_i is generated by the upstream applications, it is ingested by the producer and arrives in the *producing* state. In practice, the producer actively pulls messages from upstream applications, and the ingesting rate has its upper limit due to hardware limitations (e.g., network I/O or memory). When the producer reaches the maximum ingestion rate, we call it *fully loaded*, meaning that it ingests the next message as soon as the previous one has been processed. We note that this assumption does not cover all real-world workloads, since some applications exhibit bursty or low-frequency arrivals. Our goal here is narrower: to study mini-batching under sustained demand, where batching has the clearest impact on throughput, latency, and reliability. This regime is widely used for performance evaluation of stream processing systems because it removes source idleness as a confounding factor and reveals the system's effective processing capacity. It is also representative of many high-volume streaming applications, such as telemetry, monitoring, and online fraud detection, where platforms process continuous event streams at large scale.

We use the set $L_p = \{\beta_{ij}\}$ to represent the time these messages M stay in the producer for batching, namely *batching latency*. In practice, the producer normally ingests each message via a local network, or runs it as an integrated module of the application. Thus the ingestion latency can be neglected compared to the batching latency. β_{ij} is the batching latency of message m_i meanwhile it is in batch b_j.

We use B_S and B_T to denote the spatial batch size and temporal batch size, respectively. A batch b_j is closed when either its number of messages reaches B_S or the waiting time of the oldest message reaches B_T. The batching process is summarised in Algorithm 1, which formalises how messages are grouped under these two constraints. After a batch is closed, messages enter the transmission stage with latency L_r, and may eventually fall into one of the four possible final states shown in Fig. 3, depending on network conditions and delivery semantics.

When the network condition is normal without any congestion or packet loss, message m_i enters the *delivered* state (see transition I) as it is received by the downstream application in the computational framework. In our work, we define the message's end-to-end latency L_{ij} as the time it takes to transit from the generated to the delivered state. Thus, the end-to-end latency L of all messages M is denoted by

$$L = L_p + L_r. \tag{1}$$

Based on the introduction in Section II-B, a message m_i is considered *stale* (see transition IV) if $L \geq \zeta_L$, where ζ_L is the user-defined latency constraint. Other messages that end in the delivered state are those bringing timely content to downstream applications.

In case of poor network conditions, we observe some messages are *lost* during transmission. While some stream processing systems have implemented exactly-once delivery semantics [3], [6], these typically necessitate additional coordinators for managing transactions, leading to significant time and resource consumption. We can characterize "exactly-once" semantics as fundamentally an idempotent mechanism on the client side, enhancing the "at-least-once" delivery framework. Given that our focus is on the producer side, an in-depth exploration of "exactly-once" semantics falls outside the purview of this article. Consequently, we concentrate on two other prevalent semantic models in our study. Under *at-most-once* semantics, each message is sent only once and the producer does not need any response from downstream applications. In this case, when the producer fails to deliver a message, the message ends in a lost state (see transition II). When *at-least-once* semantics is enabled the producer retries to send the lost message until it receives a response from the downstream applications (transition II→III→I). It is worth mentioning that this does not guarantee every lost message to be delivered due to the limitations of retry times and the timeout mechanism. Furthermore, it is possible that the producer successfully resends the lost message but fails to receive a response. In this case, the message may be *duplicated* due to multiple retries (transition II→III→I→V).

B. Proposed QoS Metrics

We propose new QoS metrics to provide comprehensive insights into the performance and reliability of streaming data transportation. These QoS metrics are the optimization targets in our batching strategy, and can also be adopted in the QoS monitoring modules (e.g., dashboard) in stream processing systems.

For the evaluation of message timeliness, we introduce a new performance metric *latency violation rate* η_v to represent the proportion of messages that are stale. Assuming that a total of N_0 messages are transported within a certain time, and the end-to-end latency of N_v messages exceeds the latency constraint ζ_L. Then, the latency violation rate under this constraint is

$$\eta_v = N_v/N_0. \tag{2}$$

Other messages that are delivered within ζ_L are considered *timely messages*. We use χ_P to represent the producer's output throughput, which can be measured in bytes or the number of messages per second. The performance metric *timely throughput* is proposed in this work to evaluate the throughput of timely messages, obtained from the equation below

$$\chi'_P = (1 - \eta_v)\chi_P. \tag{3}$$

We present two reliability QoS metrics with regard to streaming data transportation under unstable network conditions. Let N_l and N_d denote the total number of lost messages and duplicated messages. We study the impact of network failure by measuring *message loss rate* and *message duplicate rate*

$$\eta_l = N_l/N_0; \eta_d = N_d/N_0. \tag{4}$$

As we discussed in Section II-B, the performance metrics measured in other mini-batching strategies [10], [13], [14], [19], including the average latency and raw throughput, are unable to reflect the real status of streaming data delivery. Therefore the suggested batch sizes in those strategies are unreliable under some circumstances. In contrast, our proposed QoS metrics provide a more systematic view of whether the applied batch size meets the real-time SLOs. Consequently, our proposed mini-batching strategy by measuring those QoS metrics turns out to be more reliable, which is demonstrated in Section VII.

C. Feature Selection

We address the challenges of choosing the appropriate batch size by predicting the proposed QoS metrics. Based on the analysis of the end-to-end latency L components in Section III-A, we screened out the key features strongly correlated to L. To achieve considerable precision and more intuitive understanding, we build two machine learning-based models to estimate the performance metrics and reliability metrics, respectively.

In addition to the spatial and temporal batch size parameters B_S and B_T the input features of the performance prediction model also include the SLO ζ_L and network condition. We use $\varepsilon_\star(d_\star, l_\star)$ to denote the network condition between the producer and downstream applications. $d_\star$ represents the round-trip network delay and $l_\star$ is the packet loss rate. The predictive model is denoted as follows:

$$\eta_v, \chi'_P = f(\varepsilon_\star, B_S, B_T, \zeta_L). \tag{5}$$

Similarly, the input features of the reliability prediction model include the network condition feature, as lost and duplicated messages are induced by unstable network conditions. Moreover, from our observation, configuring the delivery semantics and mini-batch size parameters can significantly affect the message loss and duplicate rate. We use the equation below to denote the reliability prediction model

$$\eta_l, \eta_d = f(\varepsilon_\star, B_S, B_T, \text{acks}) \tag{6}$$

where acks $= 0$ represents that at-most-once delivery semantics is enabled while acks $= 1$ stands for at-least-once.

IV. QoS Metric Monitoring

In order to build the machine learning-based prediction models and validate their accuracy, we create a testbed using the popular open-source streaming platform Apache Kafka. The testbed is used for the training dataset collection, the analysis of QoS metrics and validation of the prediction models. In Section VI, we also implement the proposed mini-batching strategy on the testbed and evaluate the effectiveness of our approach in Section VII.

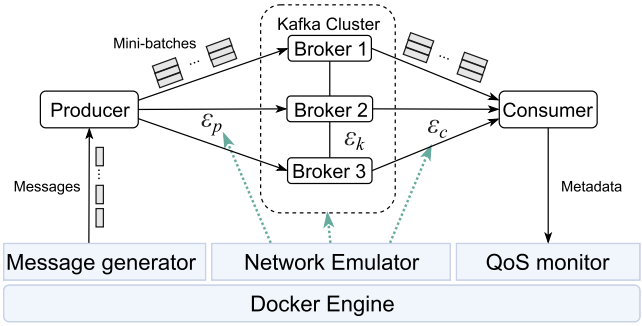

Fig. 4. Architecture of the Docker container based Kafka testbed.

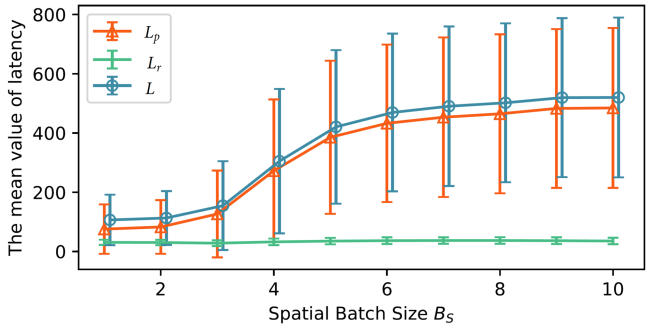

Fig. 5. Mean value and standard deviation of the latency, given B_S from 1 to 10 and B_T=1000 ms.

A. Testbed Design

Unlike stream processing frameworks, such as Samza, Storm, and Flink, which focus on computation, Apache Kafka [27] is designed for transporting streaming data. Kafka is widely used as the messaging backbone for other stream processing systems [28], [29], and its central role in linking upstream and downstream applications makes it an appropriate platform for this study.

We build our testbed with Docker, a lightweight virtualization technology [30]. As shown in Fig. 4, the testbed contains four components: a Kafka system deployed in containers, a message generator, a network emulator and a QoS monitor. Kafka follows the publish and subscribe model with producers, topics, brokers and consumers. The message generator produces continuous streams of data that the producer groups into mini batches and sends to topics. Consumers subscribe to topics and read mini batches from the brokers. The Kafka cluster consists of several brokers that receive mini batches, store them in ordered write ahead logs and replicate data for fault tolerance.

Message generator: It specifies message metadata, such as format and size. Each message carries an incremental key for tracing and embedded timestamps for computing end to end latency.

Network emulator: Using the Linux Traffic Control tool, we emulate diverse network conditions following [31]. For each test, the emulator configures and records the network characteristics between producer and cluster $\varepsilon_p(d_p, l_p)$, among brokers $\varepsilon_k(d_k, l_k)$ and between cluster and consumer $\varepsilon_c(d_c, l_c)$.

QoS monitor: It computes message latency using recorded timestamps. Further details and results are presented in later sections.

The advantages of our testbed are as follows: 1) Each Apache Kafka broker, producer, or consumer is instantiated as a Docker container embedding all required dependencies and code. Containerisation ensures consistent configuration and provides a lightweight, standalone, reproducible execution environment [30], [32], [33]. 2) Using *Docker Compose*, a Kafka cluster with a configurable number of brokers can be created or shut down quickly. To eliminate legacy impacts from prior runs, we destroy existing containers along with their associated volumes that store broker logs, topic data, and caches, and then deploy a freshly constructed Kafka cluster before each individual test. We allow a warm-up period of 60 s after cluster start-up to ensure that internal state transitions, leader election, topic partition assignments and caches stabilize before measurement begins, and we perform three independent runs, averaging the results to reduce run-to-run variability. 3) All containers operate on the same host machine, and because Linux containers share the host kernel's clock (time is not namespaced in standard container setups), clock-skew between containers is effectively negligible for millisecond-level latency measurements. To validate this assumption we measured the maximum intercontainer timestamp difference under test conditions and found it below 0.5 ms.

The resources for reproducing the testbed configuration are available in our public GitHub repository [34]. The repository provides the Docker Compose specifications needed to instantiate the Kafka cluster used in our experiments. The Compose file references the Kafka broker image hosted on DockerHub [35], so users do not need to build images locally. After starting the cluster using the provided Compose configuration, the containers already include the Python scripts required to run the performance and reliability tests.

B. New QoS Monitor

Although Kafka provides latency monitoring via Java Management Extensions (JMX), we choose to build our own QoS monitor. The reason is that Kafka only exposes internal conclusive metrics for enterprise-level monitoring. For instance, the latency metric is measured as an average over the past 1 or 5 min. The 75th and 99th percentiles of the latency are also available, but the latency violations of any user-defined ζ_L remain unclear. As shown in Fig. 4, Our QoS monitor collects the metadata in the messages received by the consumer. Three timestamps are recorded in these metadata: 1) when the message is generated, 2) when it finishes batching, and 3) when it is received by the consumer. Thus, the QoS monitor calculates the end-to-end latency L and its two parts, the batching latency L_p and the transmission latency L_r, as described in (1). In our testbed, L_p denotes the time that messages spend in the producer for mini-batches to be sent. L_r consists of the network delays d_p and d_c, and the time the messages stay on broker disks.

The results illustrated in Fig. 5 are obtained under stable network conditions where the network delays d_p, d_k, and d_c are less than 100 ms. The curves represent the mean values of batching latency $\bar{L}_p$, transmission latency $\bar{L}_r$ and their sum,

end-to-end latency $\bar{L}$ under different spatial batch sizes. The vertical bars stand for the standard deviations of the latency. The transmission latency hardly changes when the size of the mini-batch increases because messages in the same mini-batch are transmitted as a whole. Based on our assumption in Section III-A, this streaming data pipeline works in a stable state and mini-batches are fetched by the consumer as soon as they are written to the brokers. Therefore, the transmission time can be estimated by summing the network delay

$$L_r = (d_p + d_c)/2 + d_k \tag{7}$$

where d_p and d_c are the round-trip delays, and we approximate the one-way delay as half of the round-trip time. The term d_k represents the delay among Kafka brokers, which becomes relevant when synchronous replication is enabled.

In this work, we aim to study the characteristics of batching latency L_p, as it dominates the end-to-end latency L. Regarding the transmission latency L_r, it plays a comparatively smaller role in the end-to-end delay, and in our adaptive mini-batching strategy it can be obtained directly from lightweight network probing. Such probing is commonly implemented using periodic round-trip measurements (for example, sending small probe packets to observe current network delay). These probes incur negligible overhead because they are infrequent and operate independently of the streaming data path, allowing us to incorporate network-delay information without affecting message processing. As we observe from the results in Fig 5, $\bar{L_p}$ increases rapidly as the spatial batch size B_S rises from one to six. The standard deviation of L_p also grows but tends to be steady after B_S reaches seven, and so does $\bar{L_p}$. For a better understanding of this regular pattern, we regard the batching latency $L_p = \{\beta_{ij}\}$ as a stochastic process. As introduced in Section III-A, the random variable β_{ij} is the batching latency of message m_i in batch b_j.

Our QoS monitor estimates the probability density function (PDF) of β_{ij} using KDE [36]

$$\hat{f}(\beta) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{\beta - \beta_{i\star}}{h}\right) \tag{8}$$

where K is the kernel function, h denotes the bandwidth, and n represents the total number of samples. In our testbed, we set $n = 10\,000$ and the samples are extracted from the last n messages received by the consumer. The bandwidth $h = 0.215$ is computed based on Scott's Rule [36]. We use Scott's Rule for bandwidth selection because it provides a theoretically grounded tradeoff between bias and variance in KDE and thereby minimises manual tuning of the parameter [36], [37]. While the choice of n and h does affect the accuracy and responsiveness of the estimate (e.g., too small n increases variance; too large h oversmooths the distribution) prior work shows that rule-of-thumb selectors like Scott's produce near-optimal performance under a wide range of conditions. Because we run the monitor at modest sampling rates and all timestamp collection executes off the critical data path, the overhead of monitoring is minimal and does not degrade system performance. For K we select the Gaussian kernel function. The QoS monitor reports the estimated PDF once per measurement interval (e.g., once per 10 s). Through the report, we can observe the estimated distribution of β_{ij} as well as its fluctuation range and trend in real-time. We argue this QoS monitor module can be integrated into the real-time performance monitoring dashboard of the Kafka system.

Through the analysis of the results in our QoS monitor, a strong correlation is observed between B_S, B_T and the shape of the batching latency distribution, including uniformity, tendency to skew and tail. In Fig. 6 we present several estimated PDF results obtained from a series of independent tests. All tests are performed under a stable network connection and the size of the mini-batch within each test is configured with fixed B_S and B_T. Here are some insights: 1) It is worthwhile to mention that when B_S is configured as one, it indicates no batching is applied as every mini-batch contains only one message. In this case, we find that β_{ij} is nearly Gaussian distributed within a narrow range. 2) As we start using mini-batches by increasing B_S, the distribution of β_{ij} becomes positively skewed and shows a tail on its right. This indicates that the time that a message needs to wait for the mini-batch to be closed is random. 3) The length of the tail is bounded by B_T as we observe from Fig. 6. Given any fixed B_T, configuring higher B_S leads to a heavier tail, or in other words, more messages are delivered with high end-to-end latency. (iv) Another obvious fact is that after reaching a certain value, increasing B_S has no effect on the PDF of β_{ij}. The reason is that the mini-batch is unable to accumulate more messages due to the time limitation of B_T.

In addition, we clarify that the QoS Monitor is used solely as an auxiliary tool for observing system behavior. During each test, an independent process records the three timestamps associated with every message and buffers them in memory. These records are written to a log file only after the test finishes, so the runtime overhead is limited to the memory used to store the timestamp entries. In all experiments, this footprint remains below 10 MB per test, which is negligible compared with the system capacity. Therefore, the QoS Monitor does not interfere with message processing latency. Finally, the monitor is required only for collecting training data. In practical deployment it can be disabled entirely, as the proposed mini-adaptive batching strategy does not use monitored QoS metrics as input.

V. QoS Metric Prediction

A. Training Data Collection

Our QoS prediction model is built upon machine learning techniques, therefore the collection of training data is a crucial step. The training dataset consists of the features selected In Section III-C and the QoS metrics introduced in Section III-B. Fig. 7 depicts the training data collection process using our testbed.

Before each individual test, a new Kafka system is built, thus avoiding the legacy impacts from the previous test. In each test the Kafka configuration parameters B_S, B_T, and acks are fixed, as well as the network environment parameters $\varepsilon_\star$. Then, the producer ingests a certain number of messages (e.g., 10 000) from the message generator and publishes them to a new topic in Kafka. Finally, those messages are received by the consumer

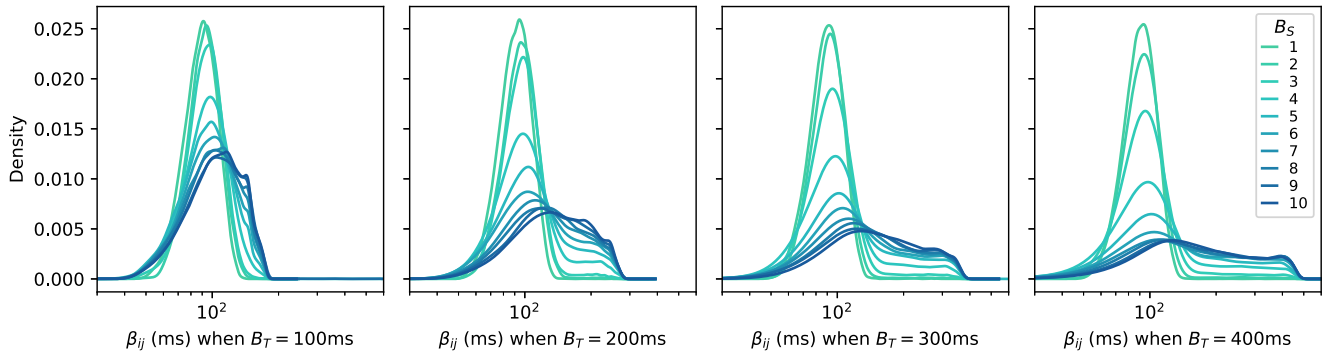

Fig. 6. Monitored result of the batching latency PDF (B_S ranging from 1 to 10 and B_T configured to 100, 200, 300, and 400 ms).

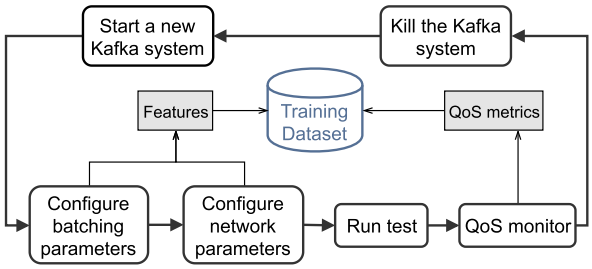

Fig. 7. Training data collection process.

and the QoS monitor collects all the metadata and computes the required QoS metrics according to (2)–(4). The number of lost messages N_l and duplicated messages N_d is counted using the unique keys assigned to the messages. Another essential feature, the latency constraint ζ_L is a user-defined SLO with uncertain value. To obtain sufficient training data, first, we sort the collected end-to-end latency data in descending order. We use $\theta_1, \theta_2, \ldots, \theta_m$ to denote the set of sorted end-to-end latency, and take every θ_i as the given latency constraint. Then, the latency violation rate under this constraint can be counted according to the position of θ_i

$$\eta_v(\zeta_L = \theta_i) = (i-1)/(m-1). \tag{9}$$

Thus in each test, we can obtain m rows of training data for predicting η_v. After recording the features and corresponding QoS metrics to the training dataset, we finish the test by killing all the existing Docker containers in the testbed. The features are configured with new values in the next test.

B. Performance Prediction

We use an ensemble learning method known as RFR to build the performance prediction model. RFR consists of multiple decision trees, while each tree plays the role of nonlinear mapping from complex input spaces into continuous output spaces. We use RFR for predicting the latency violation rate η_v as it is not computationally expensive and can avoid overfitting. Our RFR model is based on 100 decision trees with a maximum depth of 20. To generate sufficient training data, we perform tests configured with the spatial batch size B_S ranging from 1 to 25 and the temporal batch size B_T between 100 and 1000 ms. In our work, over three million samples are fed to the RFR model. 60% of the samples are used for training, 20% for the model validation and the other 20% are used for validation. The prediction model achieves a high precision with the coefficient of determination $R^2 = 0.99$.

In Fig. 8, we present several prediction results to demonstrate the correlation between mini-batch configuration parameters and the latency violation rate η_v. Each figure depicts the predicted η_v given any user-defined latency constraint ζ_L. The curves are grouped by the configured B_S while the figures are grouped by different B_T. For the developers of downstream applications, defining a latency constraint ζ_L higher than 200 ms is more reasonable as we can observe an extremely high latency violation rate (nearly 1.0) with ζ_L below 200 ms. Apparently, given the same user-defined ζ_L, configuring a larger mini-batch sees a higher latency violation rate. For instance, assuming the latency constraint for a certain application is $\zeta_L = 200$ ms, choosing the spatial batch size $B_S = 4$ and the temporal batch size $B_T = 200$ ms gains the latency violation rate at $\eta_v = 0.0202$, while increasing B_S to 10 leads to $\eta_v = 0.2378$, which is over 10 times higher. In the case with B_T configured to 400 ms, increasing B_S from 4 to 10 results in η_v burst from 0.1038 to 0.5702, which indicates that over half of the delivered messages are stale.

For the prediction of timely throughput χ'_P, we use the prediction model proposed in our previous work [15] to predict the producer's output throughput χ_P, which also reaches coefficient of determination $R^2 = 0.99$. Thus, we combine the predicted η_v with this output throughput to produce the timely throughput according to (3).

C. Reliability Prediction

For the prediction of the reliability QoS metrics, we construct the model using MLP due to the complex dimension of features. To reproduce the failure scenarios in streaming data delivery, we inject faults to the network condition $\varepsilon_p(d_p, l_p)$ introduced in Fig. 4. Both message loss and duplication are observed under the unstable network connection. In each test, we send one million messages under poor network conditions, and count the number of received messages using the unique key introduced

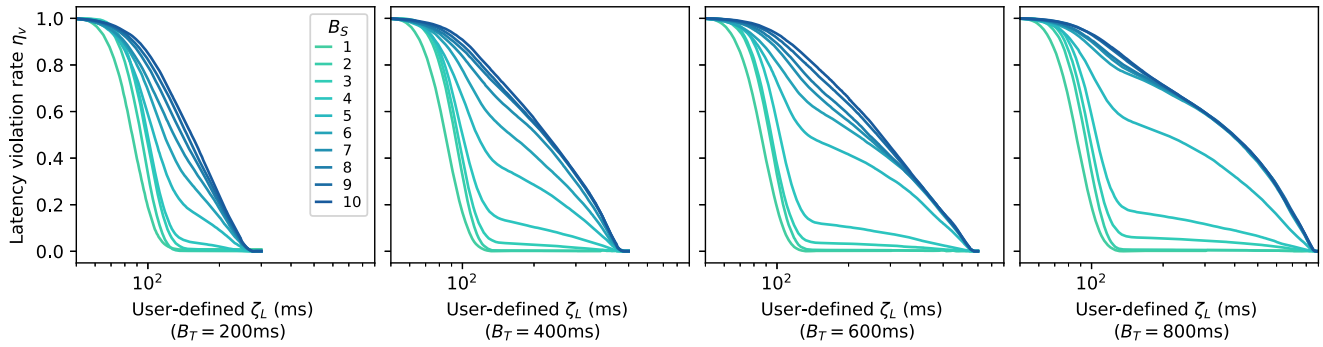

Fig. 8. Prediction result of latency violation rate η_v with various user-defined latency constraints ζ_L (B_S ranging from 1 to 10 and B_T configured to 200, 400, 600, and 800 ms).

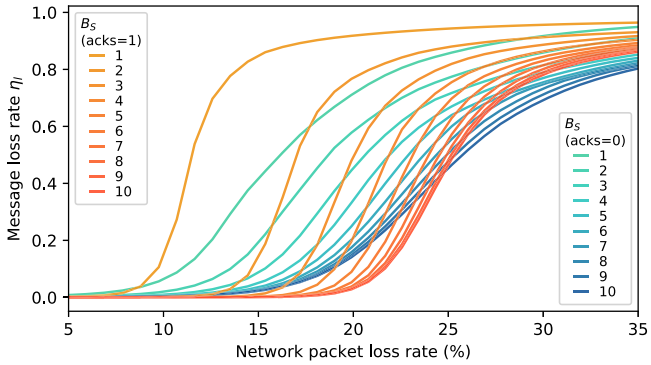

Fig. 9. Predicted impact of delivery semantics (acks=0 denotes at-most-once and acks=1 denotes at-least-once), B_S and l_p on message loss rate η_l.

in Section IV-A. Thus, the QoS metric message loss rate η_l and duplicate rate η_d can be calculated according to (4). By observing the results, we choose to ignore message duplication as its impact is minor. As a matter of fact, the message duplicate rate η_d is generally below 0.5% due to the harsh preconditions introduced in Section III-A. Moreover, most downstream applications have an idempotence mechanism to automatically deduplicate messages (e.g., using event identifiers (IDs)). From the experimental results, we observe that injecting a high packet loss rate l_p leads to a high message loss rate η_l, and the value of η_l is impacted by B_S under different semantics.

The features for the prediction model are listed in (6). Since the input and output experiment data are strongly correlated, we can use a fairly standard MLP model to achieve considerable prediction accuracy. We apply the stochastic gradient descent optimizer in the MLP model, and build four hidden layers with 200, 200, 200 and 64 neurons, respectively. We set the learning rate to 0.5 and the number of epochs to 1000. The mean absolute error, defined as the average absolute difference between the predicted values and the corresponding ground-truth measurements, is below 0.02. This is accurate enough to help us compare the impact of different batch sizes B_S, as illustrated in Fig. 9.

The horizontal axis is the network packet loss rate ranging from 5% to 35%. The curves in the acks $= 0$ group represent the predicted message loss rate η_l under at-most-once semantics while those in the acks $= 1$ group are configured with at-least-once. Within the group, each curve denotes the results obtained with fixed spatial batch size B_S. We can observe that as B_S increases, the predicted curve moves from left to right. The results also indicate that Kafka tolerates a packet loss rate $l_p \leq 8\%$ due to the TCP retransmission mechanism. The retransmissions on the network interface of the docker container can be observed by Wireshark, an open-source packet analysis tool. With a higher packet loss rate, lost messages emerge due to the timeout of retransmission. It is worth noting that when mini-batching is not applied ($B_S = 1$), transporting streaming data under at-least-once delivery semantics tends to lose more messages than at-most-once. We speculate the reason for this counter-intuitive result is that under at-least-once delivery without mini-batching, the producer requires an acknowledgment for each message, which will preempt the network bandwidth with more TCP retransmissions.

The prediction model is more meaningful for the unstable network cases with $10\% \leq l_p \leq 25\%$ because changing the spatial batch size B_S could have a significant impact on the message loss rate. For instance, when $l_p = 15\%$, increasing B_S from 1 to 3 under at-least-once delivery reduces the loss rate η_l from over 80% to less than 5%. For applications that do not consider throughput or latency and prioritize the completeness of streaming data (e.g., trading report system that runs daily), we suggest transporting the streaming data with a mini-batch size of 10 and at-least-once delivery semantics, if the network packet loss rate fluctuates below 25%. With a higher packet loss rate, at-most-once performs better but the effect is minor at such a high message loss rate (over 0.5).

Using the prediction model of η_l, we are able to formulate the concept of *timely throughput* when encountering poor network conditions, which is an extension of (3)

$$\chi'_P = (1 - \eta_v)(1 - \eta_l)\chi_P. \tag{10}$$

VI. Adaptive Mini-Batch Sizing

The proposed QoS prediction model forms the basis of our adaptive mini-batching strategy, whose goal is to maximize the timely throughput $Max\{\chi'_P\}$ under current network conditions. The strategy assumes that the producer 1) ingests data continuously while monitoring the network condition $\varepsilon_\star$ once per

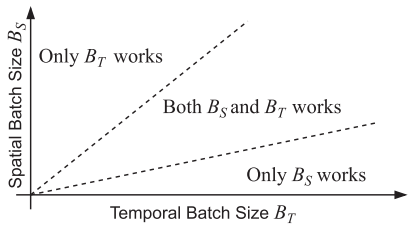

Fig. 10. Effective range of B_S and B_T to determine the actual number of messages per batch N_b.

second, 2) knows the available network bandwidth quota H, and 3) is given the SLO latency constraint ζ_L.

Mini batch configuration requires adjusting the spatial batch size B_S and the temporal batch size B_T. As discussed in Section IV-B, when B_T is fixed, the actual number of messages per batch N_b stops increasing once B_S exceeds a certain value. Fig. 10 illustrates this behavior. When either B_S or B_T is too small, the other parameter has no practical effect on N_b; for example, when $B_S = 1$, varying B_T does not change N_b. To enable adaptive sizing, both parameters must operate in their effective range.

Based on our earlier findings in [15], the producer throughput χ_P increases with N_b but must remain below the bandwidth quota H. Therefore, we first select B_T such that, even in cases where B_S is ineffective, B_T alone keeps χ_P within H. Using the prediction models in (5) and (6), the producer then estimates the timely throughput χ'_P under the current conditions. By iterating B_S within its effective range (for example, 1 to 20), we obtain a set of predicted values of χ'_P and choose the B_S that yields the maximum. This process is executed every ten seconds, following the approach in [38], and the updated configuration is applied to the producer. The adpative mini-batching process is illustrated in Algorithm 2.

In this article, we argue that our prediction model can be applied in other stream processing systems because the batching mechanisms are identical: whether batch size in terms of time or volume. We demonstrate this point of view with a detailed investigation in Section VIII. Moreover, these popular stream processing systems often directly adopt Kafka as their messaging systems [6], [39], which is also conducive to the versatility of our approach.

Algorithm 2: Adaptive Mini-Batching.

Input: Network environment metrics, $\varepsilon_\star$;
Configuration parameters, $acks, B_S, B_T$;
User-defined SLO ζ_L
Output: New configuration parameters, $acks', B'_S, B'_T$

```
B_S = 25; B_T = 0 //initial value;
while predicted χ_P ≤ H do
    B_T += 100;
    predict χ_P;
end
use current B'_T;
while true do
    monitor current ε_⋆ and obtain current SLO ζ_L;
    predict η_v and χ'_P using Eqs. (5),(3),(10);
    take χ'_P as current timely throughput;
    for B_S ∈ {1,2,3,...,25}, acks ∈ {0,1} do
        if predicted χ'_P >current χ'_P then
            update current χ'_P = predicted χ'_P;
            record current B'_S, acks';
        end
    end
end
Update mini-batching configuration acks', B'_S, B'_T;
Wait 10 seconds;
```

VII. Experimental Evaluation

We run the testbed on a computer with an Intel Core i7-8700 CPU (12 cores), 32 GB RAM and 2 TB HDD. We build the Kafka cluster from version 2.3.0, using the Docker version 19.03.6, running on Ubuntu 18.04.4 LTS. In order to evaluate the mini-batching strategy proposed in Section VI we assume the following experimental scenario. Connected cars are vehicles connected to wireless networks, and have become significantly important in the foreseeable IoT area [40]. The services of connected cars include traffic safety and cost efficiency and the demand for real-time processing is increasing [41]. In this experiment, we use the Kafka producer to publish the vehicle sensor data in Berlin, which is derived from the public website of the car-sharing business [17]. Each message is generated in real-time and contains the current location of the car. The average size of the message is around 350 bytes. We run a stream processor on the consumer side, which receives the streaming data, and calculates the geographical distance between the car and a specific charging or gas station. We conduct the experiment in both good and poor network conditions. The network delay follows a Pareto distribution to emulate real-world network status [42]. In the experiments with fault injection, we generate the network packet loss rate from the Gilbert–Elliot model, which is a two-state Markov model that has been widely applied to analyze measurements on wireless networks [43].

A. Latency and Throughput Tradeoff

We show the advantage of the proposed mini-batching strategy by comparing it with the approaches in other works. The methods to achieve the desired tradeoffs between latency and throughput in these works are similar: batching as much as possible as long as the latency constraint is guaranteed. By observing the experimental results of the vehicle sensor data processing in Fig. 11(a), we know that both the mean end-to-end latency $\bar{L_K}$ and the producer throughput χ_P rise as the mini-batch size increases. Therefore the empirical way to choose the best mini-batch size is that given the latency constraint ζ_L, choosing the largest mini-batch within the constraint can maximize the throughput [38].

To better elaborate this empirical idea, we plot the Pareto front of throughput and latency, as shown in Fig. 11(b). The horizontal axis denotes the mean end-to-end latency and the vertical axis is the reciprocal of producer throughput ($1/\chi_P$). Therefore on

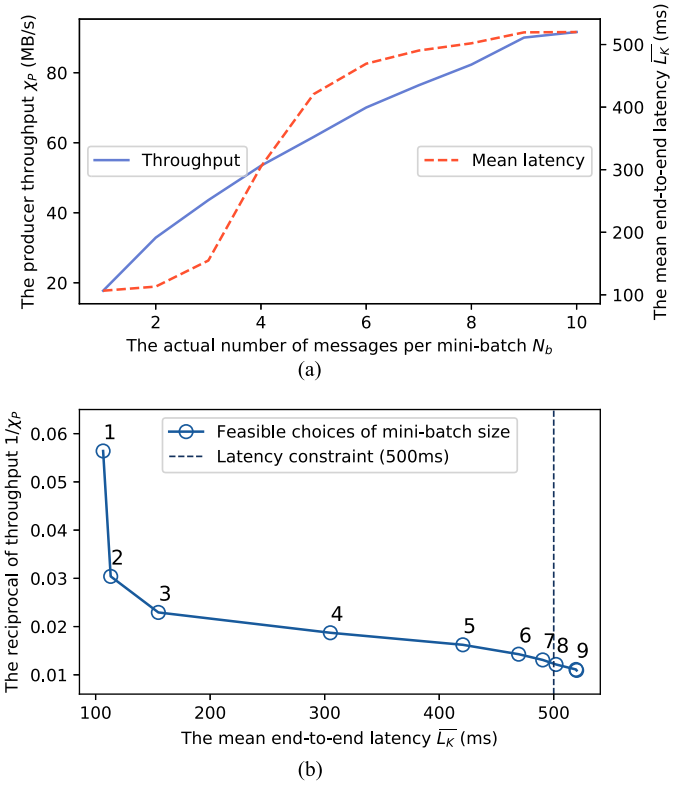

Fig. 11. Empirical tradeoff between throughput and mean end-to-end latency. (a) Impact of mini-batch size on throughput and mean latency. (b) Pareto front of throughput versus mean latency.

both axes, lower values are preferred to the higher ones. Each point represents the feasible choice of the mini-batch size, where the annotated numeral denotes the number of messages per mini-batch. In the case of processing the vehicle sensor data, we assume the user-defined latency constraint to be $\zeta_L = 500$ ms. Then, following the empirical tradeoff in other works, the best configuration is suggested as seven messages per mini-batch.

However, the above choice for mini-batching is not reliable as it measures only the mean end-to-end latency. We discuss the defects in Section II-B and here we present the choice using our QoS metrics and mini-batching strategy. We assume that in this vehicle sensor data processing case, the maximum available network bandwidth is $H = 100$ MB/s. According to the adaptive mini-batch sizing algorithm in Section VI, we first fix the temporal batch size at $B_T = 1000$ ms, which allows the producer throughput to saturate the network bandwidth. Then, the best spatial batch size B_S is selected based on the maximum timely throughput. Our mini-batching strategy is able to adapt the mini-batch size as the SLO necessitates. In Fig. 12, we illustrate the effect of different spatial batch sizes on the latency violation rate and the timely throughput, given various user-defined latency constraints.

It is observed that when the latency constraint of the downstream application is defined as $\zeta_L = 500$ ms, our adaptive mini-batching strategy configures the spatial batch size $B_S = 4$, which gains the maximum timely throughput at $\chi'_P = 50.4$ MB/s. Compared to the best configuration of seven messages per mini-batch suggested by the empirical methods, our strategy chooses a smaller mini-batch size. The advantage is depicted intuitively in Fig. 12. The observed timely throughput of $B_S = 7$ is 40.9 MB/s, which is over 18% lower than the results using our approach. Back to the definition of timely throughput in Section III-B, our mini-batch configuration leads to more messages delivered within the user-defined latency constraint.

The results indicate that although a larger mini-batch size can achieve higher producer throughput, it may also result in more messages not meeting the SLO. This is because the latency violation rate also rises rapidly as more messages wait longer to finish the mini-batch, which is observed from the dashed curves Fig. 12. We expect the proposed QoS metrics to remind developers when the system is taking up resources to deliver a large proportion of stale messages. It is worth noting that when the SLO is not strict, e.g. $\zeta_L = 700$ ms, our approach selects higher mini-batch size $B_S = 9$, as depicted in the far right of Fig. 12. This is close to the results of empirical methods ($B_S = 10$). We argue that our adaptive mini-batching strategy is more significant for time-sensitive applications.

In this experiment, our mini-batching algorithm consumes 7.9 MB of memory and the computational overhead is between 160 and 180 ms. The costs are acceptable considering the fact that the algorithm runs once per 10 s.

B. Message Success Rate

In this experiment with network fault injection, we assume that network status is known as depicted in Fig. 13. According to the mini-batching algorithm proposed in Section VI, the producer checks the network metrics $\varepsilon_p(d_p, l_p)$ every 10 s and reconfigures the mini-batch size when necessary. This is because changing configuration parameters too frequently will cause additional coordination overhead due to shuffling communication among producers and brokers [44].

The overall timely throughput is measured over the entire experiment process. Since the network features in this experiment are all some functions of the elapsed time t, we use $\eta_l(t)$ to denote the reliability QoS metric in this experiment. Given the ingestion rate of source data $\lambda(t)$, which also changes with t, we obtain the total number of messages published from the producer as $\int \lambda(t)dt$. Therefore the overall message loss rate R_l in this experiment is defined as follows:

$$R_l = \frac{\int \lambda(t)\eta_l(t)dt}{\int \lambda(t)dt}. \tag{11}$$

The overall message loss rate and timely throughput in this experiment are illustrated in Table I. Since a set of messages is accumulated and processed as a batch, it is not feasible to evaluate the throughput continuously [45]. We compare the results obtained via our adaptive mini-batching strategy with those using Pareto front-based empirical methods. We also provide two groups of experiment results using static mini-batching methods, where the spatial batch sizes are fixed to $B_S = 1$ and $B_S = 10$ throughout the entire experiment. We observe that our adaptive mini-batching strategy outperforms others in terms of timely throughput χ'_P. However, the observed overall message

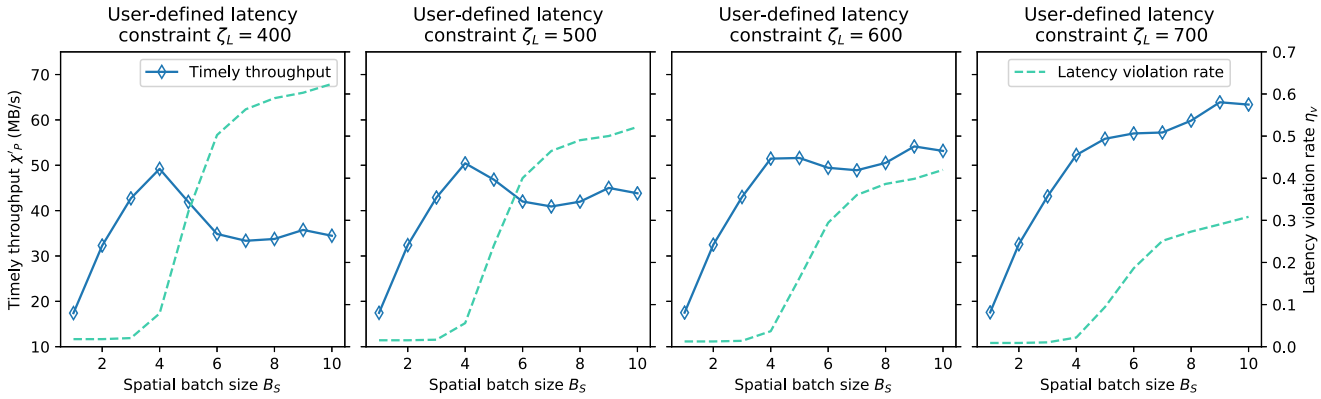

Fig. 12. Impact of spatial batch size B_S on the latency violation rate η_v and the timely throughput χ'_P ($B_T = 1000$ ms).

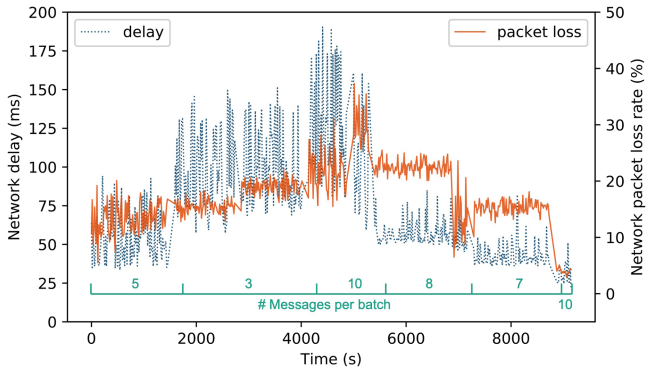

Fig. 13. Network status $\varepsilon_\star(d_\star, l_\star)$ and assigned batch size over time.

loss rate is higher than the empirical method. This indicates that more messages are successfully delivered with the empirical method but a larger proportion of them are stale messages.

It is important to note that these characteristics are specific to the Kafka testbed and the workload we use. Docker containers offer lower latency costs and lower variability than hardware virtualization [46]. Therefore the absolute number of those metrics is best ignored when other researchers try to apply our method to other streaming systems.

C. Discussions

1) Real-World Applicability: Our experiments utilize a distributed messaging system constructed with Docker containers, diverging from prior studies that often employed virtual machines (VMs) for building distributed stream processing systems [14], [45]. The container-based and VM-based systems are basically alike except for some key differences: 1) Containers are more lightweight since multiple Kafka nodes share a single operating system kernel, while VMs require one kernel for each node. 2) Because of the strong isolation among different nodes, a VM-based system is more secure when one of the nodes is under attack. However, such security risk is beyond the scope of this article. 3) It is easier to migrate the container-based Kafka testbed across various experiment environments since all dependencies are packaged into one Docker image. This flexibility allows other researchers to easily reproduce our Kafka testbed and perform follow-up work.

Currently, large-scale computing infrastructures are increasingly transitioning from VM-based to container-based systems due to their efficient software deployment capabilities [47], [48], [49]. Consequently, our experimental setup aligns closely with real-world environments.

One may ask about the potential generalization issues of our model, especially considering the hardware differences between real-world deployments and our experimental setup. To address this, we offer two perspectives. First, the use of machine learning models for predicting system performance is an emerging trend in optimizing large-scale computing infrastructures [50], [51], [52]. While generalization is a common challenge in this area, it can be effectively managed by generating extensive training data through benchmark testing, leading to significant performance benefits. Second, our model, trained in a specific and controlled environment, still holds valuable insights (e.g., the correlation between network status and message loss rate) for similar applications utilizing Apache Kafka. With the application of Transfer Learning [53], [54], these pretrained models can be adapted for related but new tasks. The transfer learning cost is expected to be substantially lower than full retraining because typically only a small subset of parameters is fine-tuned using a modest amount of new data (e.g., thousands of new instances). In practice, prior studies show that this can reduce training time by one to two orders of magnitude.

2) Comparison With Other Studies: Table II provides a qualitative comparison between our method and existing work. The column on the purpose of mini batching indicates whether each study is driven by performance or reliability aspects. We also distinguish spatial and temporal batching approaches and review how each work evaluates reliability, including end to end latency, poor network conditions, and the use of fault injection. The final column shows whether the method focuses on the message delivery stage, that is, the transfer of data from the source to the computational framework.

Most existing studies, such as [11], [12],[13], [14], [19], [45], and [56], examine mini batching after data has reached the computational framework. Their evaluations are conducted in controlled settings and center on processing behavior. Our work

TABLE III
MINI-BATCHING PARAMETERS IN STREAM PROCESSING SYSTEMS

Streaming Systems	Relevant Parameters for Mini-Batching	
	Spatial Batch Size	**Temporal Batch Size**
Flink	table.exec.mini-batch.size	table.exec.mini-batch.allow-latency
Flume	maxBatchSize	maxBatchDurationMillis
Heron	heron.instance.network.-read/write.batch.size.bytes	heron.instance.network.-read/write.batch.time.ms
Samza	bulk.flush.max.size.mb /bulk.flush.max.actions	bulk.flush.max.interval.ms
Storm	topology.producer.batch.size	topology.flush.tuple.freq.millis

instead focuses on the delivery stage and aims to ensure that streaming data reaches the framework efficiently and reliably. Stein et al. [10] also considered message delivery, it does not explore difficult network conditions in depth, which are common in settings such as mobile devices and autonomous systems. Our study examines how mini-batching interacts with network variation and includes comprehensive metrics and fault injection experiments. As many platforms, such as Apache Spark and Apache Flink, rely on Apache Kafka for data ingestion [57], our method complements existing batching strategies that operate after data reception.

VIII. RELATED WORK

The duality of spatial and temporal batching mechanisms is common among all stream processing systems. To reduce state access, Apache Flink [3] buffers input records in MiniBatch subjecting to the configuration of maximum buffer time and the maximum number of records per MiniBatch. When using Apache Flume [58] to collect tweets from Twitter, the batch size is determined by the maximum number of tweets in a single batch and the longest time to wait before closing the batch. Heron [8] also reads and writes data in batches by specifying the maximum batch size in bytes or in milliseconds. In Apache Samza [6] and Storm [59], sending a batch is called flushing and similarly, the batch size depends on the flushing interval and the maximum number of messages to be batched before flushing. In Table III we list the relevant configuration parameters for spatial and temporal batching in those popular stream processing systems.

Many studies have examined mini batching or micro batching in streaming data processing. Facebook engineers reported that combining streaming and batching can significantly accelerate long pipelines [60]. Lohrmann et al. [61] proposed adaptive batching with dynamic task changes to balance throughput and latency in large scale systems, using QoS reporters to track mean latency and QoS managers to adjust batch sizes [38]. The effect of batch size on throughput and end to end latency in Apache Spark was analyzed in [13], where the Das et al. [13] applied fixed point iteration to determine the batch interval. Cheng et al. [14] introduced a fuzzy control mechanism for dynamic batching in Spark Streaming. Stein et al. [10] developed latency aware adaptive micro batching for streaming compression on graphic process units. Garcia et al. [12] studied the interaction between micro batching and data stream frequency in multi core systems, highlighting throughput latency tradeoffs.

While these works mainly evaluate mean latency, other research considers full latency distributions. Latency aware scheduling for extended Hadoop examines both average and quantile latency [62], and Heinze et al. [63] used elastic scaling to reduce latency violations. These ideas motivate our introduction of latency violation rate as a QoS metric. Our work employs machine learning techniques to predict QoS metrics from configuration and network conditions, similar to predictive approaches in other systems [64], [65]. Venkataraman et al. [64] predicted performance of analytic applications using simple features and small training sets, while Didona et al. [65] combined analytical models with machine learning for performance prediction.

Recent work by Leonarczyk et al. [56] provided a detailed follow-up to the micro-batching algorithms originally proposed by Stein et al. [10]. Both studies use reactive controllers that adjust batch size based on recently observed end-to-end latency. These approaches rely on a moving-average of the last 5–10 latency samples to smooth outliers and avoid oscillatory batching decisions, since the controller reacts directly to instantaneous latency measurements. In contrast, our method uses predictive models that estimate QoS metrics from mini-batch configurations and network conditions. As decisions are based on predicted latency distributions rather than individual samples, our approach is less sensitive to transient spikes and does not require moving-average smoothing. This marks a key distinction: prior work stabilizes reactive control, whereas our approach predicts and optimizes batching behavior under dynamic network conditions during data transportation.

IX. CONCLUSION

In this work, we explore how to use mini-batches to transport streaming data efficiently in real-time. We discuss the limitations of existing batching methods, and use experimental data to illustrate their unreliability. In order to collect experimental data efficiently, we build a Docker testbed of Kafka with fault injection components. We explore the correlation between mini-batch size and the end-to-end latency. Using KDE, we create a new QoS monitor to detect the distribution of batching latency. We build a performance prediction model to estimate the latency violation rate with high accuracy. The message loss rate with poor network status is predicted by an MLP model. We use a newly defined metric, the timely throughput, as the optimization objective in our adaptive mini-batching strategy. In the end, we compare the proposed mini-batching strategy with methods in other works, and the experimental results confirm the superiority of our new method. For future work, we plan to evaluate more variable real-world workloads with multiple client scenarios.

REFERENCES

[1] I. Lee, "Big Data: Dimensions, evolution, impacts, and challenges," *Bus. Horiz.*, vol. 60, no. 3, pp. 293–303, 2017.
[2] P. Carbone, M. Fragkoulis, V. Kalavri, and A. Katsifodimos, "Beyond Analytics: The evolution of stream processing systems," in *Proc. 2020 ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 2651–2658.

[3] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State management in Apache Flink: Consistent stateful distributed stream processing," *Proc. VLDB Endowment*, vol. 10, no. 12, pp. 1718–1729, 2017.

[4] F. Van Wyk, Y. Wang, A. Khojandi, and N. Masoud, "Real-time sensor anomaly detection and identification in automated vehicles," *IEEE Trans. Intell. Transp. Syst.*, vol. 21, no. 3, pp. 1264–1276, Mar. 2020.

[5] A. Abdallah, M. A. Maarof, and A. Zainal, "Fraud detection system: A survey," *J. Netw. Comput. Appl.*, vol. 68, pp. 90–113, 2016.

[6] S. A. Noghabi et al., "Samza: Stateful scalable stream processing at LinkedIn," *Proc. VLDB Endowment*, vol. 10, no. 12, pp. 1634–1645, 2017.

[7] T. Akidau et al., "MillWheel: Fault-tolerant stream processing at internet scale," *Proc. VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.

[8] S. Kulkarni et al., "Twitter Heron: Stream processing at scale," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 239–250.

[9] H. Herodotou, L. Odysseos, Y. Chen, and J. Lu, "Automatic performance tuning for distributed data stream processing systems," in *Proc. IEEE 38th Int. Conf. Data Eng.*, 2022, pp. 3194–3197.

[10] C. M. Stein et al., "Latency-aware adaptive micro-batching techniques for streamed data compression on graphics processing units," *Concurrency Comput.: Pract. Experience*, vol. 10, 2020, Art. no. e5786.

[11] A. S. Abdelhamid, A. R. Mahmood, A. Daghistani, and W. G. Aref, "Prompt: Dynamic data-partitioning for distributed micro-batch stream processing systems," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 2455–2469.

[12] A. M. Garcia, D. Griebler, C. Schepke, and L. G. L. Fernandes, "Evaluating micro-batch and data frequency for stream processing applications on multi-cores," in *Proc. 30th Euromicro Int. Conf. Parallel Distrib. Netw.-Based Process.*, 2022, pp. 10–17.

[13] T. Das, Y. Zhong, I. Stoica, and S. Shenker, "Adaptive stream processing using dynamic batch sizing," in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 1–13.

[14] D. Cheng, X. Zhou, Y. Wang, and C. Jiang, "Adaptive scheduling parallel jobs with dynamic batching in spark streaming," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 12, pp. 2672–2685, Dec. 2018.

[15] H. Wu, Z. Shang, and K. Wolter, "Performance prediction for the Apache Kafka messaging system," in *Proc. IEEE 21st Int. Conf. High Perform. Comput. Commun.; IEEE 17th Int. Conf. Smart City; IEEE 5th Int. Conf. Data Sci. Syst.*, 2019, pp. 154–161.

[16] H. Wu, Z. Shang, G. Peng, and K. Wolter, "A reactive batching strategy of Apache Kafka for reliable stream processing in real-time," in *Proc. IEEE 31st Int. Symp. Softw. Rel. Eng.*, 2020, pp. 207–217.

[17] D. Happ, N. Karowski, T. Menzel, V. Handziski, and A. Wolisz, "Meeting IoT platform requirements with open pub/sub solutions," *Ann. Telecommun.*, vol. 72, no. 1–2, pp. 41–52, 2017.

[18] B. G. Deepthi, K. S. Rani, P. V. Krishna, and V. Saritha, "An efficient architecture for processing real-time traffic data streams using Apache Flink," *Multimedia Tools Appl.*, vol. 83, no. 13, pp. 37369–37385, 2024.

[19] Q. Zhang, Y. Song, R. R. Routray, and W. Shi, "Adaptive block and batch sizing for batched stream processing system," in *Proc. 2016 IEEE Int. Conf. Autonomic Comput.*, 2016, pp. 35–44.

[20] Z. Chen, J. Xu, J. Tang, K. A. Kwiat, C. A. Kamhoua, and C. Wang, "GPU-accelerated high-throughput online stream data processing," *IEEE Trans. Big Data*, vol. 4, no. 2, pp. 191–202, Jun. 2016.

[21] J. Medhi, "Waiting time distribution in a Poisson queue with a general bulk service rule," *Manage. Sci.*, vol. 21, no. 7, pp. 777–782, 1975.

[22] D. Griebler, A. Vogel, D. De Sensi, M. Danelutto, and L. G. Fernandes, "Simplifying and implementing service level objectives for stream parallelism," *J. Supercomput.*, vol. 76, pp. 1–26, 2019.

[23] P. Dobbelaere and K. S. Esmaili, "Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry Paper," in *Proc. 11th ACM Int. Conf. Distrib. Event-Based Syst.*, 2017, pp. 227–238.

[24] D. E. Quevedo, A. Ahlén, and J. Ostergaard, "Energy efficient state estimation with wireless sensors through the use of predictive power control and coding," *IEEE Trans. Signal Process.*, vol. 58, no. 9, pp. 4811–4823, Sep. 2010.

[25] J. Ding, S. Sun, J. Ma, and N. Li, "Fusion estimation for multi-sensor networked systems with packet loss compensation," *Inf. Fusion*, vol. 45, pp. 138–149, 2019.

[26] H. Wu, Z. Shang, and K. Wolter, "Learning to reliably deliver streaming data with Apache Kafka," in *Proc. 50th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2020, pp. 564–571.

[27] K. Goodhope et al., "Building LinkedIn's real-time activity data pipeline," *IEEE Data Eng. Bull.*, vol. 35, no. 2, pp. 33–45, Feb. 2012.

[28] Apache Kafka, "Powered by - Apache Kafka - the Apache software foundation," 2021. [Online]. Available: https://kafka.apache.org/powered-by

[29] G. Hesse, C. Matthies, M. Perscheid, M. Uflacker, and H. Plattner, "ESPBench: The enterprise stream processing benchmark," in *Proc. ACM/SPEC Int. Conf. Perform. Eng.*, 2021, pp. 201–212.

[30] C. Boettiger, "An introduction to Docker for reproducible research," *ACM SIGOPS Operating Syst. Rev.*, vol. 49, no. 1, pp. 71–79, 2015.

[31] A. Jurgelionis, J.-P. Laulajainen, M. Hirvonen, and A. I. Wang, "An empirical study of NetEm network emulation functionalities," in *2011 Proc. 20th Int. Conf. Comput. Commun. Netw.*, 2011, pp. 1–6.

[32] A. Havet, R. Pires, P. Felber, M. Pasin, R. Rouvoy, and V. Schiavoni, "SecureStreams: A reactive middleware framework for secure data stream processing," in *Proc. 11th ACM Int. Conf. Distrib. Event-Based Syst.*, 2017, pp. 124–133.

[33] J. Cito and H. C. Gall, "Using Docker containers to improve reproducibility in software engineering research," in *Proc. 38th Int. Conf. Softw. Eng. Companion*, 2016, pp. 906–907.

[34] "Kafka cluster startup repository." Accessed: Nov. 11, 2025. [Online]. Available: https://github.com/woohan/kafka_start_up

[35] "Dockerhub library." Accessed: Nov., 11, 2025. [Online]. Available: https://hub.docker.com/r/woohanx/kfk_node

[36] D. W. Scott, *Multivariate Density Estimation: Theory, Practice, and Visualization*. Hoboken, NJ, USA: Wiley, 2015.

[37] S. Chen, "Optimal bandwidth selection for kernel density functionals estimation," *J. Probab. Statist.*, vol. 2015, no. 1, 2015, Art. no. 242683.

[38] B. Lohrmann, P. Janacik, and O. Kao, "Elastic stream processing with latency guarantees," in *Proc. IEEE 35th Int. Conf. Distrib. Comput. Syst.*, 2015, pp. 399–410.

[39] Apache Flink, "Apache Flink-stateful computations over data streams," (n.d.). [Online]. Available: https://flink.apache.org/

[40] R. Coppola and M. Morisio, "Connected Car: Technologies, issues, future trends," *ACM Comput. Surv.*, vol. 49, no. 3, pp. 1–36, 2016.

[41] L. Zhu, F. R. Yu, Y. Wang, B. Ning, and T. Tang, "Big Data analytics in intelligent transportation systems: A survey," *IEEE Trans. Intell. Transp. Syst.*, vol. 20, no. 1, pp. 383–398, Jan. 2019.

[42] W. Zhang and J. He, "Modeling end-to-end delay using Pareto distribution," in *Proc. 2nd Int. Conf. Internet Monit. Protection*, 2007, pp. 21–21.

[43] A. Bildea, O. Alphand, F. Rousseau, and A. Duda, "Link quality estimation with the Gilbert–Elliot model for wireless sensor networks," in *Proc. IEEE 26th Annu. Int. Symp. Pers. Indoor, Mobile Radio Commun.*, 2015, pp. 2049–2054.

[44] A. Gounaris, G. Kougka, R. Tous, C. T. Montes, and J. Torres, "Dynamic configuration of partitioning in Spark applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 7, pp. 1891–1904, Jul. 2017.

[45] L. Mai et al., "Chi: A scalable and programmable control plane for distributed stream processing systems," *Proc. VLDB Endowment*, vol. 11, no. 10, pp. 1303–1316, 2018.

[46] J. Anderson, H. Hu, U. Agarwal, C. Lowery, H. Li, and A. Apon, "Performance considerations of network functions virtualization using containers," in *Proc. 2016 Int. Conf. Comput. Netw. Commun.*, 2016, pp. 1–7.

[47] A. Ahmed and G. Pierre, "Docker container deployment in fog computing infrastructures," in *Proc. 2018 IEEE Int. Conf. Edge Comput.*, 2018, pp. 1–8.

[48] R. Scolati, I. Fronza, N. El Ioini, A. Samir, H. R. Barzegar, and C. Pahl, "A containerized edge cloud architecture for data stream processing," in *Proc. 9th Int. Conf. Cloud Comput. Serv. Sci.*, Heraklion, Crete, Greece, 2020, pp. 150–176.

[49] T. Liu, Z. Yang, and Y. Sun, "Docker container networking based Apache storm and Flink benchmark test," in *Proc. 22nd Asia-Pacific Netw. Operations Manage. Symp.*, 2021, pp. 49–52.

[50] D. Ghoshal, K. Wu, E. Pouyoul, and E. Strohmaier, "Analysis and prediction of data transfer throughput for data-intensive workloads," in *Proc. 2019 IEEE Int. Conf. Big Data*, 2019, pp. 3648–3657.

[51] A. J. S. Tipu, P. Ó. Conbhuı, and E. Howley, "Artificial neural networks based predictions towards the auto-tuning and optimization of parallel IO bandwidth in HPC system," *Cluster Comput.*, vol. 27, pp. 1–20, 2022.

[52] H. AlQuwaiee and C. Wu, "On performance modeling and prediction for Spark-HBase applications in Big Data systems," in *Proc. IEEE Int. Conf. Commun.*, 2022, pp. 3685–3690.

[53] F. Zhuang et al., "A comprehensive survey on transfer learning," *Proc. IEEE*, vol. 109, no. 1, pp. 43–76, Jan. 2021.

[54] D. B. Rawat, "Deep transfer learning for physical layer security in wireless communication systems," in *Proc. 3rd IEEE Int. Conf. Trust Privacy Secur. Intell. Syst. Appl.*, 2021, pp. 289–296.

[55] A. M. Garcia, D. Griebler, C. Schepke, and L. G. Fernandes, "Micro-batch and data frequency for stream processing on multi-cores," *J. Supercomput.*, vol. 79, no. 8, pp. 9206–9244, 2023.
[56] R. Leonarczyk, G. Mencagli, and D. Griebler, "Self-adaptive micro-batching for low-latency GPU-accelerated stream processing," *Int. J. Parallel Program.*, vol. 53, no. 2, 2025, Art. no. 14.
[57] G. Van Dongen and D. Van den Poel, "Evaluation of stream processing frameworks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 8, pp. 1845–1858, Aug. 2020.
[58] S. Hoffman, *Apache Flume: Distributed Log Collection for Hadoop*, Birmingham, U. K.: Packt Publishing Ltd., 2015.
[59] M. Yang and R. T. Ma, "Smooth task migration in Apache Storm," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 2067–2068.
[60] G. J. Chen et al., "Realtime data processing at Facebook," in *Proc. 2016 Int. Conf. Manage. Data*, 2016, pp. 1087–1098.
[61] B. Lohrmann, D. Warneke, and O. Kao, "Nephele streaming: Stream processing under QoS constraints at scale," *Cluster Comput.*, vol. 17, no. 1, pp. 61–78, 2014.
[62] B. Li, Y. Diao, and P. Shenoy, "Supporting scalable analytics with latency constraints," *Proc. VLDB Endowment*, vol. 8, no. 11, pp. 1166–1177, 2015.
[63] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, "Latency-aware elastic scaling for distributed data stream processing systems," in *Proc. 8th ACM Int. Conf. Distrib. Event-Based Syst.*, 2014, pp. 13–22.
[64] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *Proc. 13th {USENIX} Symp. Networked Syst. Des. Implementation*, 2016, pp. 363–378.
[65] D. Didona, F. Quaglia, P. Romano, and E. Torre, "Enhancing performance prediction robustness by combining analytical modeling and machine learning," in *Proc. 6th ACM/SPEC Int. Conf. Perform. Eng.*, 2015, pp. 145–156.

Han Wu (Member, IEEE) received the B.S. degree in civil engineering from the Huazhong University of Science and Technology, Wuhan, China, in 2013, and the M.S degree in software engineering from Tongji University, Shanghai, China, in 2016, and the Ph.D. degree in computer science from Freie Universität Berlin, Berlin, Germany, in 2020.

He is currently an Assistant Professor with the University of Southampton, Southampton, U.K., working on federated learning, privacy attacks, and system dependability.

Zhihao Shang (Member, IEEE) was born in Henan Province, China, in 1988. He received the M.S degree in computer science from Lanzhou University, Lanzhou, China, in 2015, and the Ph.D. degree in computer science from the Free University of Berlin, Berlin, Germany, in 2019.

He is currently with the Zhengzhou University of Light Industry, Zhengzhou, China. His research interests include deep learning and queueing theory.

Huaming Wu (Senior Member, IEEE) received the B.E. and M.S. degrees in electrical engineering from the Harbin Institute of Technology, Harbin, China, in 2009 and 2011, respectively, and the Ph.D. degree with the highest honor in computer science from Freie Universität Berlin, Berlin, Germany, in 2015.

He is currently a Professor with the Center for Applied Mathematics, Tianjin University, Tianjin, China. His research interests include mobile cloud computing, edge computing, Internet of Things, deep learning, complex networks, and DNA storage.

Katinka Wolter (Member, IEEE) received the Ph.D. degree in computer science from Technische Universität Berlin, Berlin, Germany, in 1999.

She has been an Assistant Professor with Humboldt-University Berlin, Berlin, and a Lecturer with Newcastle University, Newcastle upon Tyne, U.K., before joining Freie Universität Berlin, Berlin, as a Professor for dependable systems in 2012. Her research interests include model-based evaluation and improvement of dependability, security and performance of distributed systems, and networks.