Cost-Effective Autoscaler under Event-Driven Architecture

Hsin-Ying Li and Cooper Cheng-Yuan Ku

Institute of Information Management, National Yang Ming Chiao Tung University, Hsin-Chu City, Taiwan, ROC

Abstract

Elasticity is one appealing feature of the cloud system. Service providers can quickly scale up or down cloud resources according to service demands or budgets. Moreover, the event-driven microservice architecture is one of the primary paradigms in the cloud environment. The paradigm is widely used for real-time event streams, such as fraud detection. It provides not only high maintainability but also good scalability. To achieve scalability without human interaction, the autoscaling problem comes into play. In this paper, based on queueing theory and buffer length metric, a cost-effective autoscaler is proposed to manage the event-driven microservice architecture. With service level agreement (SLA) imposed, the autoscaler aims to minimize the average time for requests buffered in the system and the SLA violation percentage with minimum resources. Comparisons of performance with other methods are presented. Furthermore, evaluations of the proposed autoscaler with eager rebalancing and incremental cooperative rebalancing protocols are also implemented. Simulation results show that the proposed autoscaler achieves almost the same level of the average staying time in the system and SLA violation percentage with nearly half of the resources provisioned.

***Keywords*:** Autoscaling, Microservices, Event-Driven Architecture,

Introduction

The recent trend of cloud computing has attracted the attention of both the industrial and academic fields. The main reasons more enterprises are willing to migrate their on-premises servers to the cloud are due to the elasticity and the pay-as-you-use pricing mechanism. By moving to the cloud, companies do not have to predict hardware requirements in the future and can save massive amounts of costs for infrastructure. Cloud service providers generally utilize virtualization technology to offer resources to enterprises. Virtualization allows a single server to run multiple operating systems and multiple services at the same time. The virtualization trend of shifting from traditional virtual machines (VM) to containers is also prevalent. The booming Docker container [1] and the well-known container orchestration tool, Kubernetes [2] can not be overlooked. The container is more suitable for autoscaling as it provides a shorter startup time than VM.

Containerization is also a massive enabler of two widely adopted technologies, i.e., microservices and event-driven architecture. Microservices is the paradigm of decoupling traditional monolithic architecture into several independent services. This may enhance the maintainability and scalability of the overall application. Event-driven architecture indicates asynchronous communications between services through a message broker. Combined with microservice, the event-driven architecture improves the scalability and fault tolerance of cloud applications. This paradigm can be widely observed in distributed data stream processing, such as fraud detection and big data processing.

To achieve the elasticity of the architecture mentioned above without human intervention, the autoscaling method comes into play. Autoscaling refers to dynamically adjusting resources in response to changes in workload and maintaining the quality of service

(QoS). Its main idea is to provision more resources when the workload is high and reduce the number of resources when it is low. The objective is to meet the service level agreement (SLA) with minimum resources to save cost. The decision of scaling can be either reactive or predictive. How to autoscale properly and cost-effectively is one of the essential topics in the cloud computing field. If overprovisioned, higher costs will be incurred. If under-provisioned, more percentage of SLA violations will be observed. However, most previous research focuses on the traditional request/response architecture. Less attention has been paid to the event-driven microservice paradigm.

To deal with the autoscaling problem, researchers have tried both reactive and predictive approaches. A robust reactive approach should always be included in an autoscaler, even if a high accuracy proactive method exists, as stated in [3]. This indicates the importance of reactive autoscaling approaches. However, most reactive strategies are based on threshold decisions under event-driven microservice architecture [4]–[6]. Some research results argued that simple threshold-based decisions provide only limited autoscaling capability. In addition, scaling results in an additional rebalance time under event-driven microservice architecture, which degrades the overall performance. It is essential to take the rebalance overhead into consideration when scaling.

This work attempts to build a reactive autoscaler based on the queueing model and the current buffer length. In addition, we also integrate Kafka's incremental cooperative rebalancing protocol to maintain throughput when needed.

Related Work

Microservice and Event-Driven Architecture

In recent years, more and more companies have been willing to migrate their on-premises servers to cloud environments. In the cloud environment, enterprises such as

Amazon, Apple, Twitter, and Netflix have adopted microservice architecture instead of the traditional monolithic architecture in their applications [7]. Microservice architecture provides high maintainability, scalability, and flexibility through decoupling a monolithic application into multiple loosely coupled microservices [8], [9]. When it comes to resource provisioning, the act of scaling can be configured more precisely. If we can identify the bottleneck microservice in our applications, we can only scale the relevant services accordingly instead of scaling the whole application.

Based on the concept of microservices, event-driven architecture (EDA) uses events to trigger a set of decoupled services asynchronously [10]. The EDA can be found in many modern applications and is widely adopted in distributed data stream processing and the Internet of Things. In this paradigm, an event will be forwarded to an event queue and distributed into the individual channel. As shown in Fig. 1, the event processor is responsible for processing the event and producing a new event into another event channel when necessary. Another event processor will be responsible for processing the newly generated event. Combined with the microservice design pattern, the entire architecture can provide high throughput, availability, and elasticity [11].

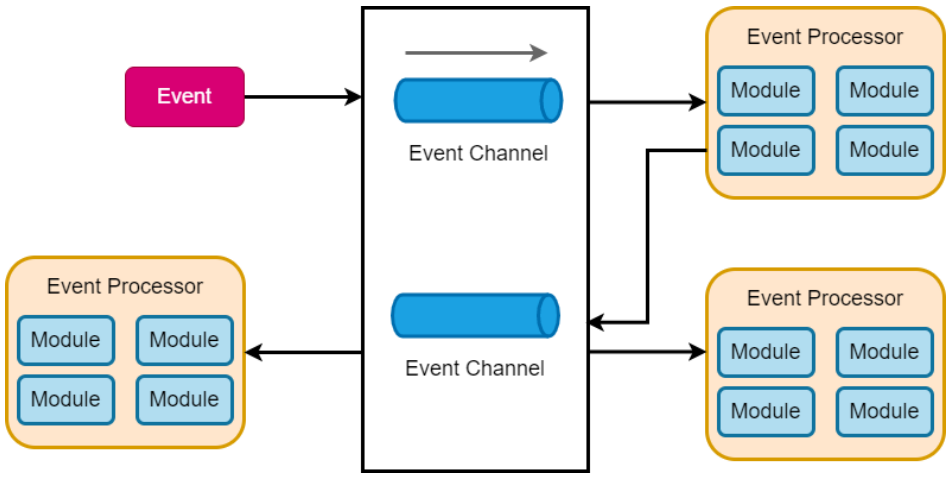

Fig. 1 Event-driven architecture [10]

The Autoscaling Problem

Autoscaling automates the tuning process of the required resources in the cloud. Users generally want to meet the quality of service (QoS) requirement with minimum resources. However, more service-level agreement (SLA) violations and performance degradation may occur if under-provisioned. If over-provisioned, the companies would need to pay extra money to the cloud provider. Therefore, the importance of autoscaling properly can not be overlooked.

Some researchers have conducted comprehensive surveys of diverse autoscaling strategies [3], [12]. The autoscaling strategy can be generally categorized into reactive and predictive methods. Reactive methods scale based on specific thresholds or rules, which can be either static or dynamic. Most cloud providers and Kubernetes, a widely adopted container administration tool, utilize this technique for scaling [13]. On the other hand, predictive approaches predict future workload and trigger scaling actions accordingly. Intuitively, one can anticipate that overall predictive methods will perform better than reactive ones. Nonetheless, time is usually required for predictive methods to train the prediction model. Before the model is performant enough to be put online for autoscaling decisions, a robust reactive approach is still required [3]. Thus, this work focuses on developing a reactive model for autoscaling. In addition, most of the previous research focuses on request/response microservices. While both request/response and event-driven microservices share many similar research characteristics, this section will mainly focus on request/response microservices. Related research concerning autoscaling in event-driven microservices architecture will be discussed in the next section. This thesis follows the footsteps of literature [3], [12] and discusses recent works of each autoscaling strategy.

Threshold: Threshold-based strategies can be commonly seen in industry autoscaler, such as Kubernetes horizontal pod auto-scaling (HPA) [14] and Amazon Autoscaling Service [15]. The thresholds are usually static and based on infrastructure metrics, like CPU or memory utilization. Some argued that these simple threshold-based autoscalers are not flexible enough and only provide limited autoscaling performance [16].

Queueing Model: Queueing Model is a mathematical analysis model constructed based on Queueing Theory [17]. Several autoscaler assume a memoryless arrival process and a flexible number of servers. This can be described as an M/M/c model. This queueing model is still mainstream in research and can provide closed-form solutions. Li et al. proposed a low-rate DDoS mitigation solution based on the M/M/c model [18]. The mechanism computes the minimum number of containers for trusted traffic and guarantees the QoS. The remaining resources are used to maximize the QoS of unknown requests as much as possible. Toka et al. utilized the MMPP/M/c model to approximate the Kubernetes horizontal pod auto-scaling (HPA) [16]. The model assumed a Markov-modulated Poisson process (MMPP) as the arrival distribution. They also further extended their work to include ML-based forecast methods. This work adopts multiple M/M/1 queueing models to abstract a single-tier microservice.

Time Series Analysis: Time series analysis is one of the major predictive autoscaling strategies. It refers to predicting future workload and resources needed using historical data. Abdullah et al. gather training datasets at profiling time to build a predictive autoscaler [19]. During this time, a reactive method based on CPU usage is used. The collected dataset is used to construct a predictive autoscaler based on decision tree regression. They forecast the workload to determine the optimal number of containers that satisfy the response time.

Reinforcement Learning: Reinforcement learning approaches aim to teach the autoscaler to scale for maximizing rewards adaptively. The autoscaler will take a scaling-up or scaling-

down action and observe the corresponding results. If the operation has a positive impact, the scaler will be more likely to take the same action next time under a similar situation and vice versa. Rossi et al. introduced a reinforcement learning autoscaler to scale container-based applications horizontally and vertically [20]. The solution exploits parameters that can be estimated about the system dynamics.

Apache Kafka and the Rebalance Problem

Autoscaling under event-driven architecture introduces challenges not faced in traditional request/response architecture [21]. One major issue is the synchronization between all the containers of one scaled service. The goal of synchronization is to distribute the events waiting in the queues among the scaled microservices. During synchronization, the rebalancing process is activated. Considering Apache Kafka, a specific topic is divided into several partitions and stored on different brokers. In Fig. 2, a group of consumers consuming from the same topic can be viewed as a consumer group. The consumer in Kafka can be considered a microservice and is the targeted service this thesis aims to scale.

When a consumer joins or leaves a consumer group, all the partitions will be first revoked from all consumers, and all the events in each partition will stop being processed. This is called the stop-the-world effect. All the consumers will then rejoin the consumer group, and partitions will be distributed evenly among them. This process is called eager rebalancing. It leads to a synchronization barrier and may increase buffer length. Therefore, in the context of autoscaling, the scaling-up and scaling-down actions should be made with careful consideration [22].

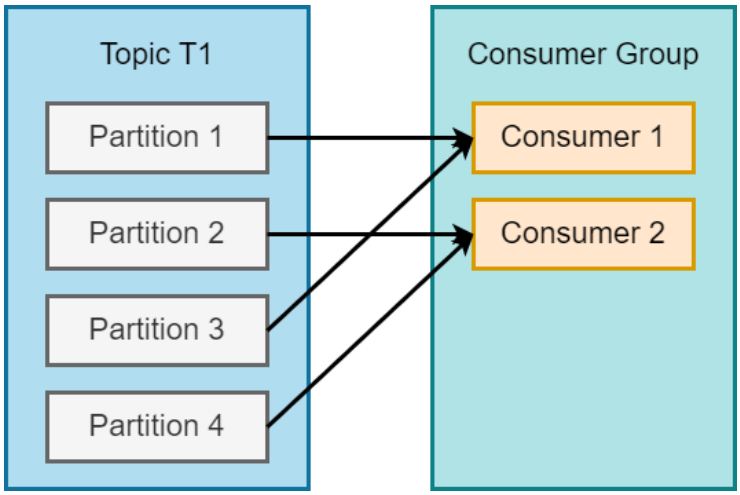

Fig. 2 Topic and consumer group [23]

To reduce the synchronization overhead and maintain throughput, Kafka version 2.4 has introduced the incremental cooperative rebalancing protocol [24]. Instead of halting all consumers of all partitions, the new cooperative partition assignor aims to revoke the least number of partitions to reach the new balance with two operations, as shown in Figure 3. We take adding a consumer as an example. Firstly, only the partitions that need to be modified are revoked. Secondly, the revoked partitions are then assigned to the newly joined consumer. During the two operations, all other consumers continue to consume from the rest of the unaffected partitions. When scaling up or down, the protocol can partly reduce the effect of rebalancing on overall throughput.

In this paper, the proposed autoscaler will be tested using both eager rebalancing protocol and incremental cooperative rebalancing protocol to see whether different rebalancing protocols affect the performance metrics regarding autoscaling.

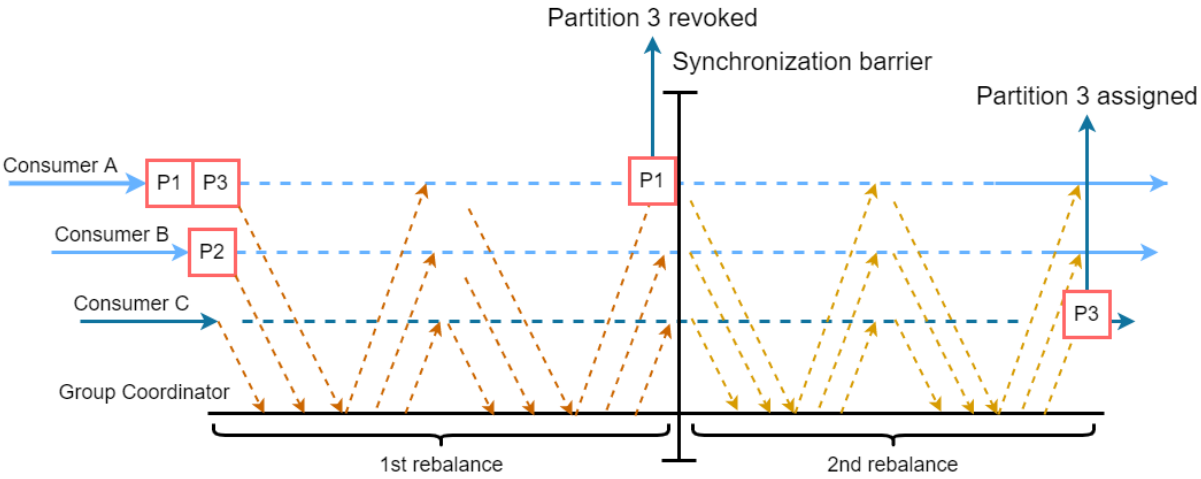

Fig. 3 Process of incremental cooperative rebalance [23]

Autoscaling Solutions under Event-Driven Microservice Architecture

In the context of autoscaling solutions under event-driven microservice architecture, limited research can be found compared to those under the traditional request/response paradigm. While some of the works considered the scaling overhead caused by rebalancing, some did not. Most of the mitigation mechanism is to fine-tune the assessment period of the autoscaler. One of the widely-used autoscaling tools in the industry is Kubernetes Event-driven Autoscaling, Keda [4]. It is currently the best choice for autoscaling under event-driven architecture in the industry [25]. Keda provides multiple built-in scalers based on different metrics. For Apache Kafka, the message broker used in the experiment, Keda offers to scale based on buffer length.

Ezzeddine et al. proposed a proactive and reactive combined horizontal autoscaler of consumer microservices [22]. The autoscaler aims to optimize the tail latency of event processing time. The mechanism considered the rebalance problem that may hurt the overall processing time. It mitigated the issue through careful scaling decisions to minimize the number of short-lived consumer services. For the reactive approach, the authors suggested scaling based on the ratio of arrival and consuming rates. The proactive approach is based on

workload prediction through an autoregressive model. Chindanonda et al. introduced a reactive autoscaler for IoT scenarios [21]. The authors proposed several metrics based on message producing rate, estimated processing rate, and queue length. These metrics are used to deal with SLO violations minimization for the dynamic workload. To prevent rebalance overhead, the mechanism applies a rule to constrain the rate at which consumers can scale down.

Rampérez et al. addressed the problem of autoscaling for distributed services through a reactive and predictive method [5]. The reactive way used response time as a threshold for making scaling decisions. The predictive methods utilized a regression model to predict the trend of relevant SLA parameters and determine how quickly a state of SLA violation will be reached. Lombardi et al. proposed a workload prediction autoscaler to estimate the minimum scaling action required to meet target performance [26]. It predicts the message input rate and estimates the performance of the application. The prediction is based on CPU usage and uses artificial neural networks for performance prediction. In addition, a maximum scaling frequency is defined by tuning the assessment period. A learning stage is included in research [5], [22], [26]. This phase is for the generation of predictive models. To address autoscaling during the learning period, a qualified reactive autoscaler is still required.

Khaleq and Ra introduced an agnostic approach by identifying the most resource the services consumed, such as CPU, memory, and traffic [6]. The resource with the highest demand is then used for scaling. The value of the scaling threshold will be obtained dynamically during the runtime of services. Dickel et al. proposed horizontal autoscaler on stateful IoT gateways [27]. The scaling decisions are based on CPU utilization, the number of concurrent active connections, and the throughput per gateway. Matic et al. introduced a monitoring tool for the RabbitMQ queue [28]. The mechanism monitored the message

producing rate and consuming rate. The scaling-up action will be triggered when the producing rate is larger than the consuming rate.

Proposed Method

Problem Statement

In the proposed framework, we focus on a specific scenario with the reactive approach based on queueing model. The buffer length metric (when), horizontal scaling (how), and a CPU-intensive consumer microservice are used for optimal decision-making, and then the consumer units are added or released. The overview of the proposed system architecture is presented in Figure 4. The goal of the proposed autoscaler is to extract useful metrics from the event-driven systems, use queueing model and the proposed buffer length metric to estimate the minimum consumers required, and scale the number of consumers accordingly. Details of each module are presented in the subsequent sections.

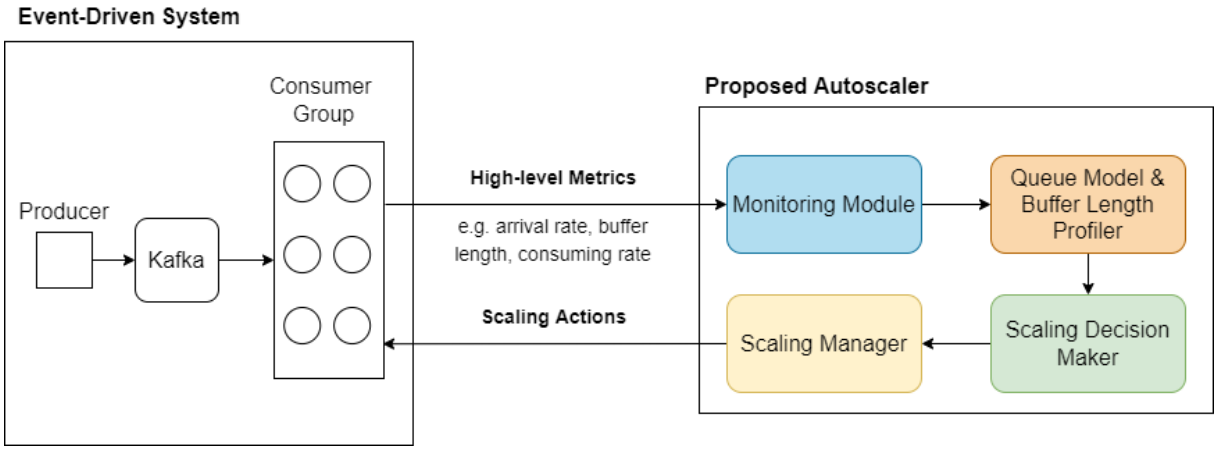

Fig. 4 Proposed Framework

System Modules

The detailed architecture of system modules is presented in Fig. 5.

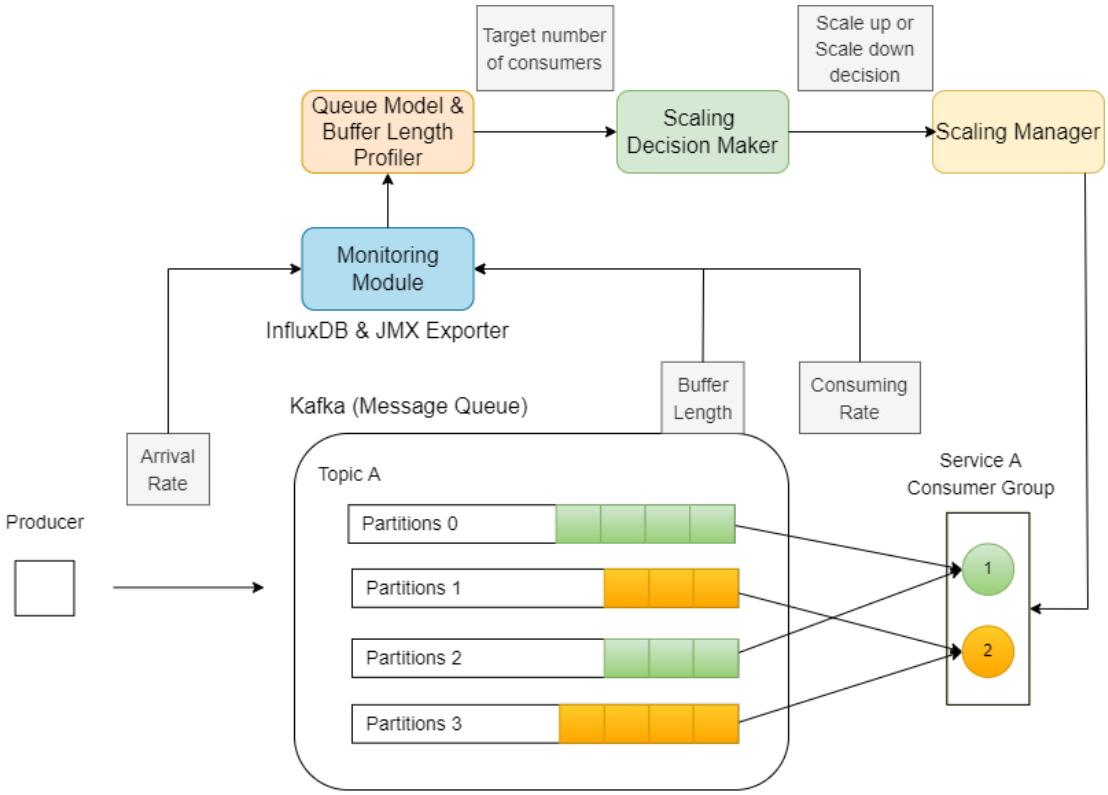

Fig. 5 System Modules

Monitoring Module

For the monitoring method, both active and passive monitoring approaches are adopted. The arrival rate, buffer length, consuming rate, and time in the system of each request is obtained from the target system as presented in Figure 5. Active monitoring means the targeted application sends monitoring metrics to a central location to be further processed in the queue model & buffer length profiler module. A time-series database, InfluxDB, is the central location for storing metrics. The consumer group microservice sends consumed events and event staying time in the system to InfluxDB. In addition, passive monitoring is also incorporated. Furthermore, JMX Exporter is also used to extract Kafka statistics, such as partition and topic status. These extra data are helpful when it comes to debugging the system.

Queue Model & Buffer Length Profiler

In the queue model & buffer length profiler module, estimations of the target number of

consumers are made using two methods, multiple M/M/1 queueing models and the proposed buffer length metric. Details of the two estimation methods are presented in the subsequent sections. The obtained two target numbers of consumers will be passed to the next decision-maker module to make scaling decisions accordingly. Related parameters used in this section can be found in Table 1.

Table 1 Related parameters in the experiment

Symbol	Definition
λ	The overall arrival rate of the system
μ	Consuming rate (service rate) of each consumer container
c_q	Minimum number of consumers required to satisfy QoS (calculated by queue model profiler)
ρ	The utilization rate of a single consumer
P_0	Probability of 0 requests in the system
P_n	Probability of n requests in the system
L	Expected number of requests in the system
T	Average staying time of the requests in the system
T_a	Target SLA (target average staying time of the requests in the system)
$Buffer$	Monitored buffer length in the queue
$Buffer_t$	Buffer length threshold
c	Current number of consumers
c_l	The target number of consumers (calculated by buffer length profiler)
w	Constant parameter

Queue Model Profiler

The formalization of the targeted event-driven system with the queueing model is presented. To estimate the system's performance, the average staying time (includes waiting time in the queue and processing time in the server) of each event is used to quantify the system performance. In addition, we assume the arrival rate of events follows the Poisson

distribution and the service rate of each consumer obeys an exponential distribution. These assumptions can be commonly seen in queueing analysis [18], [29] As shown in Fig. 5, Apache Kafka is used as the distributed message broker. In Kafka, a topic can be further divided into several partitions. Multiple consumers can consume from different partitions. In Fig. 5, consumer 1 is consuming from partitions 0 and 2, while consumer 2 is consuming from partitions 1 and 3. Events sent to the target system will be evenly distributed between partitions, and partitions will be evenly distributed between consumers. Therefore, the message queue is composed of multiple M/M/1 queues under Kafka architecture. According to this paradigm, in Fig. 5, the green part with consumer 1 is an M/M/1 queue, and the orange part is another M/M/1 queue. One can formalize the system with multiple M/M/1 queues and try to estimate the average event staying time in the system.

Based on the analyses above, it is reasonable to formalize the target system with multiple M/M/1 queueing models. The overall arrival rate is denoted as λ. The consuming rate of each consumer container is considered the same and denoted as μ. The number of consumers in the system is represented as c_q. The arrival rate of a single M/M/1 system is $\frac{\lambda}{c_q}$. Equations (1)-(5) can be obtained according to M/M/1 queueing model [17].

ρ is used to represent the utilization rate of a single consumer container. The relationship between arrival rate, consuming rate, and utilization rate is shown in Equation (1).

$$\boldsymbol{\rho = \frac{\lambda}{c_q \mu}} \qquad \textbf{(1)}$$

If the system has a stable state ($\rho < 1$), the probability of P_0 and P_n can be obtained. P_0 is the probability of no request in the system, and P_n is the probability of n requests in the system. Each can be calculated as indicated in Equations (2) and (3), respectively.

$$P_0 = 1 - \rho \quad (2)$$

$$P_n = (1 - \rho)\rho^n \quad (3)$$

Moreover, the expected number of requests in the system is denoted as L and calculated through Equation (4) according to the M/M/1 model.

$$L = \frac{\lambda / c_q}{\mu - \lambda / c_q} \quad (4)$$

Finally, according to Little's Formula [30], the average staying time T of the requests in the system is described in Equation (5).

$$T = \frac{1}{\mu - \lambda / c_q} \quad (5)$$

The objective of the autoscaler is to make the QoS acceptable, which means the average staying time T should be less than the target SLA time T_a. The system should meet Equation (6) to meet the target QoS.

$$T = \frac{1}{\mu - \lambda / c_q} \leq T_a \quad (6)$$

The task of the queue model profiler is to calculate the minimum number of consumers required c_q based on the arrival rate λ and consuming rate μ obtained from the monitoring module.

Buffer Length Profiler

As shown in Figure 5, the current buffer length can be obtained from the system. This work defines the monitored current buffer length as $Buffer$ and a buffer length threshold as $Buffer_t$. The current number of consumers can be denoted as c. Further, the target number of consumers c_l can be calculated with Equation (7). In the equation, w represents a constant

parameter and can be obtained optimally in the experiment. By calculating the target number of consumers using this proposed buffer length metric, the buffer length in the system can be maintained and avoid the side effects of consumer crashes.

$$c_l = c + \frac{Buffer - Buffer_t}{Buffer_t} \quad (7)$$

Scaling Decision Maker

In the scaling decision maker module, the decision of whether to scale up, scale down or do nothing is made. The proposed autoscaler obtains the two target numbers of consumers, c_q and c_l from the previous module as presented in Figure 5. In addition, the current number of consumers, c, is obtained through the Kubernetes' API, the container orchestration tool used in the experiment to manage the consumer microservice. This module is executed every second and a cool-down interval of 10 seconds will be imposed whenever a scaling action is made. Rampérez et al. introduced a reactW parameter in their work [5]. Before each scaling decision, the autoscaler checks whether the last N (reactW) estimations meet scaling criteria. This thesis follows suit and incorporates the concept of reactW into the proposed autoscaler.

In the proposed autoscaling approach, the decisions of scaling-up and scaling-down are separated. To be more precise, scaling up action is triggered if the situation meets the scaling up conditions for N ReactW. The number of N is set to 2 as in [5] and is denoted as upReactW. The scaling-up condition is when the minimum target number of consumers calculated by the queue model profiler c_q or the target number calculated by the buffer length profiler c_l is larger than the current number of consumers c. Then, the maximum of c_q and c_l is recorded as the target number, as shown in Equation (8). If the condition is met for N upReactW, the number of consumer containers is scaled up to the average of the recorded

target numbers.

$$\textbf{Target number} = \max(c_q,\ c_l) \qquad (8)$$

On the other hand, the scaling-down action is triggered only when the scaling-down conditions are met for 10 consecutive reactW. The scaling down reactW, DownReactW, is set to 10. This is to avoid oscillations caused by frequent scaling actions and maintain the stability of the autoscaler. The scaling-down condition is when the minimum target number of consumers calculated by the queue model profiler c_q or the target number calculated by the buffer length profiler c_l is smaller than the current number of consumers c. The maximum of c_q and c_l is recorded as well as shown in Equation (8). If the situation persists for 10 reactW, the number of consumers is scaled down to the average of the most recent 3 records of target numbers. Using the average of the most recent three records makes the scaling actions more adaptive to the current situation. The scaling decision is simplified and shown in Algorithm 1.

Algorithm 1: Scaling Decision Maker

Input:
c_q: minimum number of consumers by the queue model profiler
c_l: target number of consumers by the buffer length profiler
c: current number of consumers
upReactW: reactive window for scale-up action
downReactW: reactive window for scale-down action
scaleUpArr: historical target number of consumers to scale up to
scaleDownArr: historical target number of consumers to scale down to
coolDownInterval: cool down interval after scaling actions

```
1  While (True):
2      optimalNum = c
3    // Scale up Evaluation
4    if c_q > c or c_l > c then
5        scaleDownArr.clear()
6        scaleUpArr.add(max(c_q, c_l))
7        if len(scaleUpArr) == upReactW then
8            optimalNum = average of scaleUpArr
```

```
// Scale down Evaluation
else if c_q < c or c_l < c then
    scaleUpArr.clear()
    scaleDownArr.add(max(c_q, c_l))
    if len(scaleDownArr) == downReactW then
        optimalNum = average of the most recent 3 records in scaleDownArr
// No scaling action is required
else
    scaleUpArr.clear()
    scaleDownArr.clear()
// Call Scaling Manager Module
if optimalNum != c then
    scaleUpArr.clear()
    scaleDownArr.clear()
    scale(optimalNum)
    sleepTime = coolDownInterval
sleep(sleepTime)
```

Scaling Manager

In the scaling manager module, scaling decisions are received from the previous decision-making module, as shown in Fig. 5. The scaling manager triggers Kubernetes' API to perform the scaling action. During each scaling-up or scaling-down step, the number of consumer containers is added or released to the received target unit from the decision maker module.

Evaluation of Proposed Framework

Comparisons of the proposed methods with the reactive version of FLAS autoscaler [5] and the most widely used scaling mechanism in event-driven architecture, Keda [4], are presented. The reactive version of the FLAS autoscaler uses response time as the threshold to make scaling decisions. The autoscaler includes two fixed scaling up and scaling down

thresholds. Further, a comparison of the proposed autoscaler with eager rebalancing protocol and with incremental cooperative rebalancing protocol, respectively, is also presented.

Experimental Setup

The experiment is performed on an Intel i5-7500 CPU 3.40GHz machine with 16 GB RAM running Ubuntu 20.04.4. Apache Kafka version 3.1.0 is used as the event-driven broker. For simplicity, a single node cluster is used. We deploy the Java-based producer and consumer group using the Docker container. The number of partitions in the experimental topic is set to 10. As for other parameters related to the scaling algorithm, the experiments are conducted with an SLA of 500 ms response time. Events with a staying time longer than 500 ms will be considered violating SLA. Furthermore, we monitored the arrival rate of the most recent 2 seconds as the current Poisson arrival rate λ. The service rate μ, the average number of requests handled per second, is 58. Some key parameters of our experiments are shown as follows:

- SLA = 500 ms
- μ = 58 requests / second
- $Buffer_t$ = μ * Current number of consumers * 1.95
- w = 1.5
- *upReactW* (with eager rebalancing protocol) = 2
- *upReactW* (with incremental cooperative rebalancing protocol) = 4
- *downReactW* = 10

As for parameters of the compared autoscaler, the SLA for the reactive version of the FLAS scaler is set to 500 ms as well. The polling interval of Keda [4] is set to 5 seconds, and the buffer length threshold is set to 290, which is five times the consumption rate of a single consumer. While some works tested their autoscaler with the synthetic workload, we choose

to test the proposed autoscaling methods with a real-world dataset as this can reflect more real-world workload situations. The FIFA World Cup 1998 dataset [31] is used in the experiment. This research randomly selected a slice of the workload and sanitized the requests to be per second based load. The selected workload is 210 seconds. Moreover, the maximum request arrival rate per second is scaled to 500. The workload pattern is shown in Fig. 6.

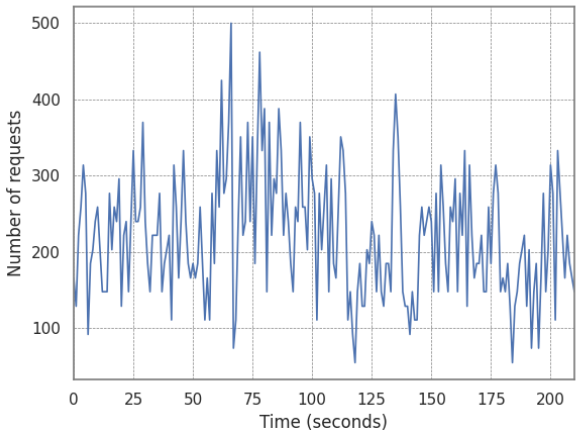

Fig. 6 Workload pattern

Performance Metrics

Definitions of some critical performance metrics for evaluation are described below:

1.Average time in the system refers to the average staying time for all requests. The staying time in the system of one request includes queue time in Kafka topic and processing time of the consumer.

2.SLA violation percentage is calculated as the number of requests violating SLA divided by the total number of requests.

3.Consumer time refers to the number of consumers provisioned at each time unit multiplied by the total time unit. This metric is used to identify the resource usage of each autoscaling approach. Shorter consumer time indicates a more cost-effective scaling mechanism.

4.Throughput indicates the average consuming rate per second of the system. This throughput metric compares the proposed autoscaling algorithms with eager rebalancing and incremental cooperative rebalancing protocols.

Evaluations of proposed autoscaler, FLAS, and Keda

In this section, comparisons between the proposed autoscaler, the reactive version of FLAS autoscaling mechanisms, and the widely used autoscaler, Keda, are presented and discussed. The average values of some important metrics from 20 executions are shown in Table 2. For each performance metric, the bold text indicates the autoscaler with the best performance.

Table 2 Comparison of autoscaling mechanisms

	Proposed autoscaler	FLAS [5]	Keda [4]
Number of scaling-up actions	3.35	3.29	1.75
Number of scaling-down actions	4.05	0.47	0.75
SLA violation percentage	14.53	**12.01**	17.22
Time in the system (ms)	317.50	**316.25**	470.99
Consumer time	**780.55**	1380.38	783.08

Comparing the proposed autoscaler with the FLAS scaler, the average SLA violation rate of the proposed methods is higher by 2.52 percent. As for the average time in the system, the result of the proposed methods is slightly higher than that of FLAS by 1.25 ms. Nevertheless, the consumer times of the proposed approaches are almost half of that of the FLAS scaler. Consumer time is strongly related to the cost of an enterprise. The larger the consumer time is, the more cost will incur. The proposed autoscaler only provisioned 57% of the consumers provisioned by FLAS scaler on average. This indicates that the proposed autoscaler can significantly save the infrastructure cost of an enterprise.

Fig. 7 demonstrates the comparison of each autoscaling mechanism concerning the

average number of consumers provisioned over time. The consumer time discussed above can be obtained from the area under each line. The scaling-down condition of the FLAS scaler is too challenging to meet. This results in overprovisioning, provisioning 8 consumers after 60 seconds without scaling down afterward. The average number of scaling down actions among 20 executions is only 0.47, as presented in Table 2. The average number of consumers provisioned over time of the proposed autoscaler can adapt more to the testing workload with fewer consumers. This indicates that the proposed autoscaler in this work is essentially more cost-efficient than the FLAS scaler.

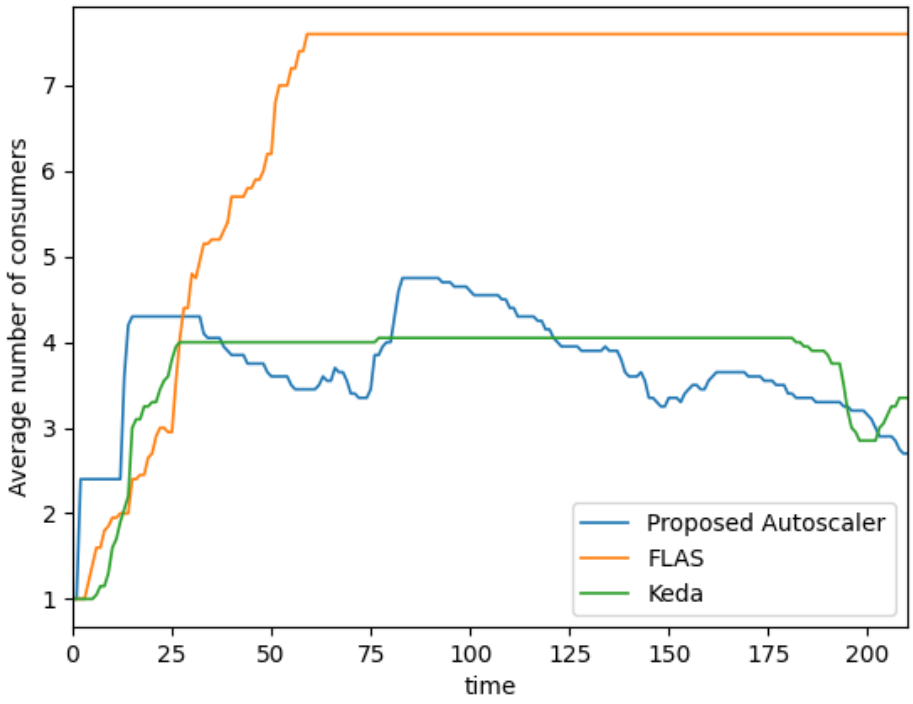

Fig. 7 Average number of consumers over time

Likewise, comparing the proposed autoscaler with the widely used Keda, the average number of consumers provisioned by Keda is more stable over time than that of the proposed autoscaler, as shown in Fig. 7. Concerning the SLA violation rate, the violation rate of Keda is higher than that of the proposed autoscaler. In addition, the overall average time in the system

of Keda is higher. This may be related to the timing of scaling actions, buffer length threshold, and polling interval configuration. As shown in Fig. 7, the Keda green line increases slower than the blue proposed autoscaler line. Further, the overall average consumer time of Keda is slightly higher than that of the proposed autoscaler. This may also be associated with the buffer length threshold configuration. These configurations should be carefully configured when running the Keda autoscaler in production.

Evaluations of proposed autoscaler with different rebalancing protocol

In the previous section, the eager rebalancing protocol was adopted for each autoscaler in the experiment. In this section, this thesis further compares the proposed autoscaler with eager rebalancing protocol and with incremental cooperative rebalancing protocol. The results are shown in Table 3. For each performance metric, the bold text indicates the protocol with better performance.

Table 3 Comparison of proposed autoscaler with different rebalancing protocols

	Proposed autoscaler with eager rebalancing protocol	Proposed autoscaler with incremental rebalancing protocol
Scale up interval (s)	10.25	11.80
Scale down interval (s)	10.80	12.10
Number of scaling-up action	3.35	2.05
Number of scaling-down action	4.05	2.50
SLA violation percentage	**14.53**	18.11
Time in the system (ms)	**317.50**	358.04
Consumer time	780.55	**710.85**
Throughput	99.45	**99.50**

First, concerning the average scaling up and scaling down time, scaling up and down time is composed of scaling API request time, container startup time, consumer initializing

time, and rebalance time. The scaling API request time is small enough to be ignored in the experiment. Container startup time is around 2 to 4 seconds, and consumer initializing time is roughly 1 to 2 seconds. The rebalance time takes up most of the scaling time.

With eager rebalance protocol, the average scaling up time is generally 10.25 seconds, as observed from the experimental results. While with incremental cooperative rebalancing protocol, the average scaling up time is 11.8. The scaling up time with incremental cooperative rebalancing protocol is longer than using eager rebalance protocol for roughly 1 second. The longer scaling time can also be observed when scaling down. This indicated that the system's rebalance time using incremental cooperative rebalancing protocol is higher. When it comes to autoscaling, this may lead to higher scaling overhead.

Although the number of scaling actions with incremental cooperative rebalancing protocol is fewer than with eager rebalance protocol, the average SLA violation percentage is still 3.58% higher and the average time in the system is also 40.54 ms higher. The conclusion is that using incremental cooperative rebalancing protocol may lead to higher time in the system and SLA violation percentage. However, if throughput is considered, using incremental cooperative rebalancing protocol performs better. The experimental results indicate that with incremental cooperative rebalancing protocol, the system can achieve approximately the same throughput as using eager rebalance protocol with fewer consumers provisioned. The consumer time of the incremental cooperative rebalancing protocol is 69.7 units fewer than that of the eager rebalance protocol. Therefore, if throughput is an important factor one wants to maintain during autoscaling, using incremental cooperative rebalancing protocol should be considered.

Summary of Experimental Results

In summary, the proposed autoscaler with queueing model and buffer length profiler is more cost-efficient. Provisioning almost half of the resources of the reactive version of the FLAS scaler provisioned while only 2 to 3 more percent of requests violate the SLA. Adopting the proposed scaling strategy can be more economical and maintain almost the same performance. In addition, we also studied how different rebalancing protocols may affect the overall performance of the proposed autoscaler. Using average SLA violation or time in the system as the performance metric, integrating the protocol is not a performant decision. Nonetheless, if throughput is considered a performance indicator, including this protocol would be a smart design. On the whole, whether to adopt incremental cooperative rebalancing protocol or not depends on the requirement of the application system.

Conclusions

In this paper, a cost-efficient reactive autoscaler for event-driven microservice architecture with Apache Kafka as the event broker is proposed. The proposed scaling method utilized multiple M/M/1 queueing models and the proposed buffer length metric to estimate the optimal number of consumers required. Comparison results of the proposed autoscaler with two reactive threshold-based autoscaler show that the proposed autoscaler in this work is more cost-efficient. The proposed autoscaler provides comparable performance to the benchmark autoscalers but with much lower resources provisioned. Furthermore, evaluations of the eager rebalancing and incremental cooperative rebalancing protocol with the proposed autoscaler are made. This thesis concluded that when time in the system and SLA violation rate are considered, the autoscaler developed in this work with incremental cooperative rebalancing protocol is not as performant as with eager rebalancing protocol. However, if throughput is considered as a performance metric, incorporating the incremental cooperative

rebalancing protocol would be a reasonable choice. The decision of whether to include the rebalancing protocol should be made after setting performance indicators and gathering enough insights from the target system.

The proposed approach focuses on reactive autoscaling and can be combined with predictive autoscaling mechanisms. Predicting workload can be integrated into the proposed method and utilize queueing model and buffer length metric to estimate the required number of consumers in the future and perform scaling actions in advance. In addition, this study focuses on scaling a single consumer microservice. One can extend the queueing model into queueing networks to solve scaling a multi-tier service. Finally, some of the parameters in the experiment are obtained empirically. One can further incorporate adaptive control mechanisms into the proposed autoscaler in this work to automatically adjust and tune those parameters.

References

[1] "Home - Docker." https://www.docker.com/ (accessed Jun. 08, 2022).
[2] "Kubernetes." https://kubernetes.io/ (accessed Apr. 08, 2022).
[3] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-Scaling Web Applications in Clouds: A Taxonomy and Survey," *ACM Comput. Surv.*, vol. 51, no. 4, Jul. 2018, doi: 10.1145/3148149.
[4] "KEDA | Kubernetes Event-driven Autoscaling." https://keda.sh/ (accessed May 25, 2022).
[5] V. Rampérez, J. Soriano, D. Lizcano, and J. A. Lara, "FLAS: A combination of proactive and reactive auto-scaling architecture for distributed services," *Future Generation Computer Systems*, vol. 118, pp. 56–72, May 2021, doi: 10.1016/j.future.2020.12.025.
[6] A. Abdel Khaleq and I. Ra, "Agnostic approach for microservices autoscaling in cloud applications," *Proceedings - 6th Annual Conference on Computational Science and Computational Intelligence, CSCI 2019*, pp. 1411–1415, Dec. 2019, doi: 10.1109/CSCI49370.2019.00264.
[7] J. Soldani, D. A. Tamburri, and W. J. van den Heuvel, "The pains and gains of microservices: A Systematic grey literature review," *Journal of Systems and Software*, vol. 146, pp. 215–232, Dec. 2018, doi: 10.1016/j.jss.2018.09.082.
[8] B. Butzin, F. Golatowski, and D. Timmermann, "Microservices approach for the internet of things," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2016, pp. 1–6. doi: 10.1109/ETFA.2016.7733707.
[9] F. Ponce, G. Márquez, and H. Astudillo, "Migrating from monolithic architecture to

microservices: A Rapid Review," in *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, 2019, pp. 1–7. doi: 10.1109/SCCC49216.2019.8966423.
[10] M. Richards, *Software architecture patterns*, vol. 4. O'Reilly Media, Incorporated 1005 Gravenstein Highway North, Sebastopol, CA …, 2015.
[11] S. Khriji, Y. Benbelgacem, R. Chéour, D. el Houssaini, and O. Kanoun, "Design and implementation of a cloud-based event-driven architecture for real-time data processing in wireless sensor networks," *Journal of Supercomputing*, vol. 78, no. 3, pp. 3374–3401, Feb. 2022, doi: 10.1007/S11227-021-03955-6.
[12] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Elasticity in Cloud Computing: State of the Art and Research Challenges," *IEEE Transactions on Services Computing*, vol. 11, no. 2, pp. 430–447, 2018, doi: 10.1109/TSC.2017.2711009.
[13] L. M. Ruíz, P. P. Pueyo, J. Mateo-Fornés, J. v Mayoral, and F. S. Tehàs, "Autoscaling Pods on an On-Premise Kubernetes Infrastructure QoS-Aware," *IEEE Access*, vol. 10, pp. 33083–33094, 2022, doi: 10.1109/ACCESS.2022.3158743.
[14] "Horizontal Pod Autoscaling | Kubernetes." https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ (accessed Apr. 09, 2022).
[15] "AWS Auto Scaling." https://aws.amazon.com/autoscaling/ (accessed Apr. 08, 2022).
[16] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly, "Machine Learning-Based Scaling Management for Kubernetes Edge Clusters," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 958–972, 2021, doi: 10.1109/TNSM.2021.3052837.
[17] Kleinrock Leonard, *Queueing Systems*, vol. I. Wiley, 1975.
[18] Z. Li, H. Jin, D. Zou, and B. Yuan, "Exploring new opportunities to defeat low-rate DDoS attack in container-based cloud environment," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 3, pp. 695–706, Mar. 2020, doi: 10.1109/TPDS.2019.2942591.
[19] M. Abdullah, W. Iqbal, A. Mahmood, F. Bukhari, and A. Erradi, "Predictive Autoscaling of Microservices Hosted in Fog Microdata Center," *IEEE Systems Journal*, vol. 15, no. 1, pp. 1275–1286, 2021, doi: 10.1109/JSYST.2020.2997518.
[20] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and Vertical Scaling of Container-Based Applications Using Reinforcement Learning," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, 2019, pp. 329–338. doi: 10.1109/CLOUD.2019.00061.
[21] P. Chindanonda, V. Podolskiy, and M. Gerndt, "Self-Adaptive Data Processing to Improve SLOs for Dynamic IoT Workloads," *Computers*, vol. 9, no. 1, p. 12, 2020.
[22] M. Ezzeddine, S. Tauvel, F. Baude, and F. Huer, "On the design of SLA-aware and cost-efficient event driven microservices," *WoC 2021 - Proceedings of the 2021 7th International Workshop on Container Technologies and Container Clouds*, pp. 25–30, Dec. 2021, doi: 10.1145/3493649.3493657.
[23] G. Shapira, T. Palino, R. Sivaram, and K. Petty, *Kafka: the definitive guide*. " O'Reilly Media, Inc.," 2021.
[24] "KIP-429: Kafka Consumer Incremental Rebalance Protocol - Apache Kafka - Apache Software Foundation." https://cwiki.apache.org/confluence/display/KAFKA/KIP-429%3A+Kafka+Consumer+Incremental+Rebalance+Protocol (accessed May 27, 2022).
[25] A. Arjona, P. G. López, J. Sampé, A. Slominski, and L. Villard, "Triggerflow: Trigger-based orchestration of serverless workflows," *Future Generation Computer Systems*, vol. 124, pp. 215–229, Nov. 2021, doi: 10.1016/J.FUTURE.2021.06.004.

[26] F. Lombardi, A. Muti, L. Aniello, R. Baldoni, S. Bonomi, and L. Querzoni, “PASCAL: An architecture for proactive auto-scaling of distributed services,” *Future Generation Computer Systems*, vol. 98, pp. 342–361, Sep. 2019, doi: 10.1016/j.future.2019.03.003.
[27] H. Dickel, V. Podolskiy, and M. Gerndt, “Evaluation of autoscaling metrics for (stateful) IoT gateways,” *Proceedings - 2019 IEEE 12th Conference on Service-Oriented Computing and Applications, SOCA 2019*, pp. 17–24, Nov. 2019, doi: 10.1109/SOCA.2019.00011.
[28] M. Matic, S. Ivanovic, M. Antic, and I. Papp, “Health monitoring and auto-scaling rabbitMQ queues within the smart home system,” *IEEE International Conference on Consumer Electronics - Berlin, ICCE-Berlin*, vol. 2019-September, pp. 380–384, Sep. 2019, doi: 10.1109/ICCE-BERLIN47944.2019.8966229.
[29] S. Yu, Y. Tian, S. Guo, and D. O. Wu, “Can We Beat DDoS Attacks in Clouds?,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 9, pp. 2245–2254, 2014, doi: 10.1109/TPDS.2013.181.
[30] J. D. C. Little, “A Proof for the Queuing Formula: L = λW,” *https://doi.org/10.1287/opre.9.3.383*, vol. 9, no. 3, pp. 383–387, Jun. 1961, doi: 10.1287/OPRE.9.3.383.
[31] “World Cup 1998 Dataset.” http://ita.ee.lbl.gov/html/contrib/WorldCup.html (accessed May 25, 2022).