# Cost-Efficient and Latency-Aware Event Consuming in Workload-Skewed Distributed Event Queues

Mazen Ezzeddine, Gael Migliorini, Françoise Baude, Fabrice Huet

# Cost-Efficient and Latency-Aware Event Consuming in Workload-Skewed Distributed Event Queues

**Mazen Ezzeddine**
Université Côte d'Azur, CNRS, I3S
HighTech Payment Systems, HPS
Nice, France
mazen.ezzeddine@ univ-cotedazur.fr

**Gaël Migliorini**
HighTech Payment Systems, HPS
Aix En Provence, France
gael.migliorini@hps-worldwide.com

**Françoise Baude**
Université Côte d'Azur, CNRS, I3S
Nice, France
francoise.baude@univ-cotedazur.fr

**Fabrice Huet**
Université Côte d'Azur, CNRS, I3S
Nice, France
fabrice.huet@univ-cotedazur.fr

## ABSTRACT

Distributed event queues have emerged as a central component in building large scale cloud applications. In distributed event queues, guaranteeing a maximum event processing latency for high percentile of events in a cost-efficient manner is of paramount interest. This is achieved through efficient and accurate solutions to autoscale event consumers to meet the incoming workload. However, most of current solutions to autoscale event consumers are threshold-based that add/remove consumer replicas based on a metric of interest. These autoscalers lack an accurate estimation on the number of replicas that is just enough to keep up with the arrival rate of events and are not cost-efficient. Moreover, threshold-based autoscalers are not designed with workload-skewness in mind. When the workload is skewed few partitions of the distributed queue will receive higher percentile of the events produced. In such cases, the autoscale process must be complemented with a load-aware assignment of event consumer replicas to queue partitions. However, load-aware assignment is not performed by threshold-based autoscalers as they assume a uniform event load across the partitions of the queue. Hence, in this work, we first express the problem of cost-efficient scaling of event consumers to achieve a desired latency as a bin pack problem. This bin pack problem depends on the arrival rate of events, consumption rate of consumers, and on the events backlog in the queues. Next, we show that the process of scaling event consumers in face of skewed workload is performed by a controller/autoscaler and by one of the consumer replicas namely the leader. The controller monitors the cluster state and launches the appropriate number of consumer replicas. Next, the leader consumer performs a load-aware assignment of partitions to consumer replicas. In face of skewed workloads, observed results show order of magnitude gains in terms of latency guarantee as compared to an autoscale methodology that is not complemented by a load-aware assignment.

## CCS CONCEPTS

• Applied computing~Service-oriented architectures; • Applied computing~Event-driven architectures; • Computer systems organization~Cloud computing

## KEYWORDS

Distributed Event Queue, Autoscale, Rebalancing, Producer, Consumer, Event Driven Microservices, Kubernetes, SLA, Kafka, Bin Packing, Microservice, Event Driven Microservices.

**ACM Reference format:**

Mazen Ezzeddine, Gaël Migliorini, Françoise Baude, and Fabrice Huet. 2022. Cost-Efficient and Latency-Aware Event Consuming in Workload-Skewed Distributed Event Queues. *In 2022 6th International Conference on Cloud and Big Data Computing (ICCBDC 2022), August 18–20, 2022, Birmingham, United Kingdom. ACM, New York, NY, USA, 9 pages*. https://doi.org/10.1145/ 3555962.3555973

## 1 INTRODUCTION

Distributed event queues have emerged as a central component in building large scale cloud applications. They are currently being used in many real time cloud applications such as recording and analyzing web accesses for recommendations and ad placement [1], fraud detection [2] and health care monitoring [3]. Furthermore, distributed event queues are the backbone for the event driven microservices software architectural style where an application is composed of several small services communicating by exchanging events across a distributed event queue [4][5][6][7]. As such, many cloud providers already offer a distributed event queue as a service [8][9][10].

Distributed event queues are composed of several partitions distributed over a cluster of servers. Event producers generate an event and publish it into a certain partition of the queue. Consumer applications pull the events of interest and process them as required by the application logic. Event consumers are composed of one or more replicas that jointly consume events out of the event queue, with each consumer being assigned an exclusive set of partitions.

Also, no more than one consumer can be assigned to a partition. The set of consumers that are pulling events from the partitions of the event queue is called consumer group CG. In most applications (banking, health monitoring), same-key events need to be routed to the same queue partition and eventually processed by the same consumer

For event consumer applications, minimizing the time an event spends waiting in the queue and its processing time is critical to achieve low response time for high percentile of events at low cost. Hence, there is an urgent need for frameworks and solutions that autoscale event consumer applications. In fact, there has been some research efforts to autoscale event consumer applications that consume from distributed event queues [8][9][10][11][12]. However, these efforts propose threshold-based autoscalers that add or remove additional replicas when a metric of interest (e.g., CPU utilization) reaches a certain threshold. Unfortunately, such autoscalers lack an accurate estimation on the number of replicas that is just enough to keep up with arrival rate of events. In fact, recent research [13] has shown that these threshold-based autoscalers are not cost efficient, and hence, clients are over-charged for under-utilized resources. Also, threshold based autoscalers may rely on misleading metrics. For example, Amazon Kinesis [8] is not always capable of accurately identifying bottlenecks as relying on CPU policy can be misleading [13].

On the other hand, these threshold-based autoscalers assume that the workload is not skewed and the arrival rate per key (alternatively the arrival rate per partition) is uniform. However, in practice cloud applications workloads are skewed [14][15][16]. When the workload is skewed, having an autoscaler that launches additional consumer replicas based on a metric threshold without load-aware assignment of consumer replicas to partitions is sub-optimal. This is because threshold-based autoscalers assume a uniform arrival rate into all the event queue partitions, and they do not make any effort to assign consumer replicas to partitions in a load aware manner. However, when the workload is skewed, the autoscale process must be complemented with a load-aware assignment of consumer replicas to queue partitions.

In essence, state of the art distributed event queues such as Kafka [17][18] does not currently provide any out of the box solution for consumer group autoscaling. Neither they provide a load-aware rebalancing (consumers-partitions assignment) to assign partitions to consumers fairly when the workload is skewed. Besides Kafka leverages a recent rebalancing logic called Incremental Cooperative [19] that promotes data locality over load-awareness. This Incremental Cooperative rebalancing promotes sticking partitions to their already assigned consumers without taking load-awareness into consideration, assuming thus that the arrival rate into each partition is uniform.

Therefore, in this work, we target cost efficient autoscaling of event consumer applications in face of skewed workloads. We first express the problem of minimally scaling event consumers to achieve a desired latency as a bin pack problem that depends on the arrival rate of events, consumption rate of consumers, and on the event backlog in the queues. We mathematically formulate the problem and express it as an integer linear programming ILP model. Next, we describe in detail the system architecture implemented as solution for autoscaling workload-skewed event consumers. We discuss the role of the *Controller/Autoscaler* process which decides, using the formulated ILP model, on the minimum number of event consumers needed to achieve a desired latency as well as the assignment of the scaled consumers to queue partitions. We also discuss the role of the event consumer group leader which performs the partitions-consumers assignment as suggested by the *Controller/Autoscaler* i.e., in a load aware manner.

Finally, it is important to note that in our work, we assume that the event queue is partitioned in a way that guarantees that the arrival rate into an individual partition of the event queue (whether with skewed or non-skewed arrival rate) is less than the maximum consumer consumption rate. This is because no more than one consumer can be assigned to a single queue partition. Further, we do not investigate dynamic distributed event queue repartitioning/scaling as part of event consumer autoscaling, while we opt for steady-state stable queues/partitions when a partition is assigned to exactly one consumer.

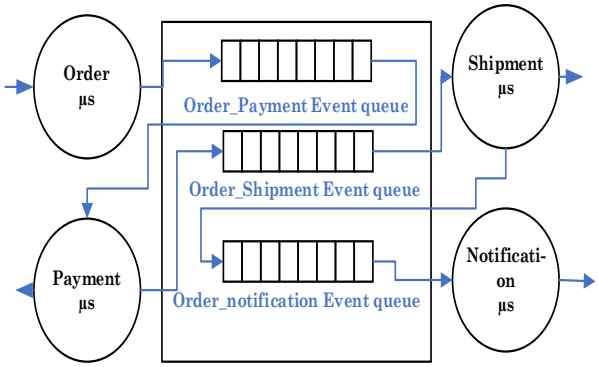


**Figure 1 : An example of an event driven microservices architecture**

## 1.1 Context and Motivational Scenario

We target an event driven microservices architecture where microservices communicate through a distributed event broker. Each microservice processes the event it pulls and produces a result event so that the next microservice in the business workflow could pull the resulting event and perform the required business logic. As an example, in Figure 1, the *Order* microservice creates an *OrderPayment* event and places it in the *OrderPayment* event queue. Next, the *OrderPayment* event is picked by the Payment microservice which processes it and creates an *OrderShipment* event, placing it in the *OrderShipment* event queue and so on.

In this architecture, each latency-critical consumer microservice is configured with a maximum event processing latency which identifies the maximum time that an event belonging to that microservice can exhibit without violating the latency SLA. For example, if the *Payment* microservice is configured with maximum

of five seconds event processing latency, then if an *OrderPayment* event is delayed for more than five seconds, the event is considered to violate the latency SLA. Note that the maximum event processing latency includes the waiting time in the queue and the processing time which depends on the business logic required by the microservice.

In the business workflow shown Figure 1, the *Payment* microservice is latency-critical and cannot tolerate a maximum event processing latency of more than five seconds. On the other hand, the *Shipment* and *Notification* microservices are less latency-critical and can tolerate a maximum event processing latency up to few minutes. Hence, in accordance with the business requirements and without loss of the generality, we focus on guaranteeing the event processing latency SLA of the *Payment* microservice. That is, the *Order* microservice acts as an event producer microservice, and the *Payment* microservice acts as a latency-sensitive consumer microservice configured with a maximum event processing latency of five seconds.

## 2 RELATED WORK

Performance SLAs are hard to guarantee. Cloud providers rarely provide end to end performance guarantee or focus on overprovisioning of resources and isolation of services to meet a desired SLA [20][21]. With the proliferation of microservices architecture, performance SLAs became even more challenging. In the context of event driven microservices architectures there has been some effort to autoscale event driven microservices communicating over a distributed event queue [11][12]. To our knowledge, none of the published work considers autoscaling event driven microservices in face of skewed workloads where a load-aware assignment of event queue partitions to consumers must be performed. Closest to our work is stream processing autoscaling which is a central problem in data stream processing research [22]. Prior work has proposed various predictive [23] [24] and heuristic policies [25] to decide when and how much to scale when input rate changes. Cloud providers offering stream processing as a service [8][9][10] provide support for autoscaling. However, they propose threshold-based autoscalers that add or remove additional replicas when a metric of interest (e.g., CPU utilization) reaches a certain threshold. Unfortunately, such autoscalers lack an accurate estimation on the number of replicas that is just enough to keep up with arrival rate of events. In fact, recent research [13] has shown that these threshold-based autoscalers are not cost efficient, and hence, clients are over-charged for under-utilized resources. Also, threshold based autoscalers may rely on misleading metrics. For example, Amazon Kinesis [8] is not always capable of accurately identifying bottlenecks as relying on CPU policy can be misleading [13]. Furthermore, none of cloud providers offers a solution for stream processing autoscale with support of skewed workloads where a specific instance of a stream operator exhibits much input rate as compared other instances of the same operator. As with event driven microservices, this problem is not optimally solved by simply increasing the parallelism of the operator in question. Rather, it must be complemented through a fair assignment of downstream to upstream operators. Research works such as [26] propose support for workload skewness in stream processing by dynamic stream rebalancing/reassignment from source operators to downstream operators. However, [26] does not consider dynamic autoscaling of operators, rather it investigates dynamic stream rebalancing on existing operators when the stream is skewed.

Also, notice the difference between scaling stream operators where downstream operators typically receive events from upstream operators in a push-based manner, and between scaling event driven microservices that consume always from upstream distributed event queues in a pull manner. Kafka distributed event queue [17][18] is currently one of the most used distributed event queue in the industry. At this stage, Kafka does not provide out of the box solution for automatic consumer group autoscaling. Also, Kafka does not currently provide a load aware rebalancing logic that can assign consumers to partitions in a load-aware manner when the workload is skewed. In contrast, Kafka recommendation is to use the *Incremental Cooperative* rebalancing logic [19] that promotes data locality (sticking consumers with their already assigned partitions) over load-awareness. As we show in the experimental section, when the workload is skewed promoting data locality with the *Incremental Sticky* rebalancing logic results in much less latency SLA guarantee as compared to a load aware rebalancing logic.

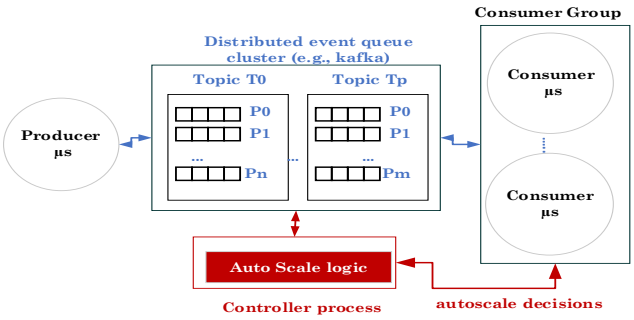


**Figure 2 : The architecture of Kafka distributed event queue. Shown also in red the newly added *Controller* module.**

## 3 BACKGROUND AND SYSTEM ARCHITECTURE

In our implementation prototype we selected Kafka as the distributed event queue. We first introduce few notions on the concept of topic (distributed event queue), partition, consumer, and consumer group. In Kafka, a topic can be thought as a mailbox into which producer applications write their events. A topic is composed of several partitions distributed over a cluster of computing servers. A consumer group is a set of event consumers that jointly and cooperatively consume events from a certain topic. Generally, for a topic *T*, we have *n* partitions and *m* consumers that read and consume events. The (re-)assignment of the *m* consumers to the *n* partitions (or inversely) is called rebalancing (also called

assignment). Rebalancing/assignment might happen several times during the lifetime of a consumer group such as when the group is initiated, or when a consumer leaves or joins the group (e.g., an instance is scaled up/down). Figure 2 shows an example of topics, partitions and consumer group in Kafka. References [17][18] provide in depth description of the overall Kafka concepts and architecture. Figure 2 also shows our newly designed *Controller* process which runs independently on the cluster. It queries the event queue (topic) for information on the arrival rate into each partition of the topic, maximum consumption rate by each consumer in the group etc. The *Controller* query rate is configurable, and we call it decision interval. In the next subsection we discuss more in depth the functionality and design of the *Controller* process.

## 3.1 Design of the Controller process

The newly added *Controller* process is the central module that monitors the cluster and runs the logic necessary for autoscaling of latency-sensitive event consumers in face of skewed workloads. In the next subsections, we first describe the queries launched by the *Controller* into the cluster to get the required state of the cluster. Next, we mathematically formulate the consumer group autoscaling problem and describe it as integer linear programming ILP model. Finally, we discuss the logic that the *Controller* runs to decide on the minimal number of consumers needed to respect the WSLA event processing latency, and the assignment of scaled consumers to partitions.

**Queries by the *Controller* into Broker**. As discussed previously, a topic can be thought as a mailbox into which producer applications write events. Each topic is composed of several partitions. Each partition p has two offset pointers to reference the last produced and the last committed (processed) events. We define the per partition event lag at time *t*, $lag_p^t$, as the number of events waiting in the partition at time *t* . It is the difference between the last produced and last committed pointers at time *t* as depicted in equation 1. Similarly, we calculate the arrival rate per partition at time *t*, $\lambda_p^t$, as the difference between the produced offsets over an interval of time as shown in equation 2. $\delta$ is the decision interval or the rate at which the *Controller* queries the event queue for state updates.

$$lag_p^t = \left(offset^p_{lastProduced}\right)_t - \left(offset^p_{lastCommitted}\right)_t \quad (1)$$

$$\lambda_p^t = \frac{\left(offset^p_{lastProduced}\right)_t - \left(offset^p_{lastProduced}\right)_{t-\delta}}{\delta} \quad (2)$$

$$\mu = \frac{\#\ events\ polled\ per\ consumer}{ProcessingTime} \quad (3)$$

## 3.2 Mathematical formulation of the consumer group autoscaling problem

For a topic *T* with *n* partitions, we define the set of arrival rates into each partition at time *t* as $\lambda_P^t = \{\lambda_{p_1}^t, \lambda_{p_2}^t, \ldots, \lambda_{p_n}^t\}$, and similarly, we define the set of lags of each partition at time *t* as $lag_P^t = \{lag_{p_1}^t, lag_{p_2}^t, \ldots, lag_{p_n}^t\}$. Also, let $c_j^t$ denotes a $j^{th}$ consumer used at time *t*. $lag_{c_j}^t$ is the lag of the consumer $c_j^t$ at time *t*. It is defined as the sum of lags of each partition assigned to $c_j^t$ as shown in equation 4 below.

$$lag_{c_j}^t = \sum_{p_i \in c_j^t} lag_{pi}^t \quad (4)$$

$$\lambda_{c_j}^t = \sum_{p_i \in c_j^t} \lambda_{pi}^t \quad (5)$$

Similarly, $\lambda_{c_j}^t$ is the arrival rate into the consumer $c_j^t$ at time *t*. It is defined as the sum of arrival rates of each partition assigned to $c_j^t$ at time *t* as shown in equation 5. Now consider a time *t* where the *Controller* has to decide on the minimal number of event consumers in the consumer group (we call it $G^t$) while still being able to respect the event latency SLA $W_{SLA}$. The condition that shall be preserved to respect $W_{SLA}$ can be formulated as per equation 6 below.

$$\forall\ c_j^t \in G^t,\ lag_{c_j}^t < \mu \times W_{SLA}\ AND\ \lambda_{c_j}^t < \mu \quad (6)$$

Informally, equation 6 states the following: at time t, when deciding on the new group of consumers, ensure (in a best effort) that: (1) each consumer in the group has its arrival rate less than its consumption rate, and (2) each consumer in the group is lagging by a number of events less than $\mu \times W_{SLA}$. $\mu \times W_{SLA}$ is the maximum number of events that can be served in less than or equal to $W_{SLA}$. Recall that $W_{SLA}$ is the maximum latency an event might exhibit without violating the latency SLA. Also, note that we use the term best effort to refer to the case where the *Controller* finds that the lag of a partition *p* is greater than $\mu \times W_{SLA}$, in such case the partition already has an SLA-violating lag. Hence, the *Controller* sets the lag of that partition to $\mu \times W_{SLA}$ (while calculating the number of consumers needed) and eventually assigns a dedicated consumer to it.

On the other hand, the cost-efficiency dictates to minimize the number of event consumers in the group $G^t$ and this translates to the equation 7 below.

$$\boldsymbol{min}\,|\boldsymbol{G^t}| \quad (7)$$

Therefore, preserving the requirements for cost-efficient latency-aware event consumer group in distributed event queues is equivalent to solving the following optimization:

$$min|G^t|\ such\ that\ \ \forall\ c_j^t \in G^t,\ \ lag_{c_j}^t < \mu \times W_{SLA}\ AND\ \lambda_{c_j}^t < \mu \quad (8)$$

In this context, let $G^t = \{c_1^t, c_2^t, \ldots, c_j^t\}$ denotes the set of consumers needed at time t to preserve the latency SLA. Also, let us denote $G^{t-1}$ the already existing set of consumers in the consumer group. The aim now is to decide on the minimum number of consumers needed at time *t*, that is, the cardinality of $G^t$, so that no event processing latency in the event queue exhibits more than the maximum event processing latency $w_{SLA}$. In fact, the optimization problem in (8) can be formulated as an Integer Linear Programming (ILP) model. In the formulation below $c_j^t$, $p_{ij}^t$ are binary variables indicating respectively whether a *jth* consumer is used at time *t*, and whether partition *i* is assigned to consumer *j* at time *t*.

$$Min\ |G^t| = \sum_{j=1}^{nb\ topic\ partitions} c_j^t$$

*such that*

$$\sum_j p_{ij}^t = 1 \; \forall \, i \; ; \; \text{(a)}$$

$$\sum_i p_{ij}^t lag_{pi}^t \leq \; c_j^t \; \times \; \mu \times W_{SLA} \; \forall \, j; \; \text{(b)}$$

$$\sum_i p_{ij}^t \lambda_{pi}^t \leq \; c_j^t \; \times \mu \; \forall \, j; \; \text{(c)}$$

$$c_j^t, \; p_{ij}^t \; ; \; binary \; variables$$

(a) ensures that each partition is assigned to only one consumer, (b) ensures that the sum of lags of the partitions assigned to each consumer is less than the SLA-violating lag (i.e., the lag that can be served in $W_{SLA}$ which is equivalent to $\mu \times W_{SLA}$). Similarly, (c) ensures that the sum of arrival rates of the partitions assigned to each consumer is less than the consumer service rate $\mu$. The ILP formulation shows that the problem of assigning partitions to consumers while guaranteeing the $W_{SLA}$ latency requirement is NP-complete. This assignment problem is equivalent to a two-dimensional bin packing where the items are the partitions described by their arrival rates $\lambda_{pi}$ and by their lags $lag_{pi}$, and the bins are the event consumers described by their service (consumption) rate $\mu$ and by their service rate multiplied by $W_{SLA}$ i.e., $\mu \times W_{SLA}$. As this assignment problem must be solved online, we resorted into an approximation algorithm that can solve the problem in polynomial time. Namely, we used the *First Fit Decreasing FFD* bin pack approximation. *FFD* is a 11/9OPT+ 6/9 approximation of the optimal bin packing solution [27]. To always guarantee load fairness across the formed bins (event consumers), we leveraged an *FFD* variant called *least-load FFD* . *Least-load FFD (LL-FFD)* was proposed by [28] for packing VMs in the datacenter into a minimal number of physical servers, while guaranteeing a fair (load-balanced) assignment of VMs to these physical servers.

**Algorithm 1 : The bin pack autoscaler logic executed by the Controller at each decision interval.**

**AutoScaleCG** ($\lambda_P^t$, $lag_P^t$ ,$W_{SLA}$)

Set $G^{t-1}$ to the current/existing set of consumers
Set $G^t$ = *Least-Load FirstFitDecreasing* ($\lambda_P^t$, $lag_P^t$ , $W_{SLA}$)
**IF** $G^t > G^{t-1}$
    Scale up by $G^t \backslash G^{t-1}$
**ELSE IF** $G^t < G^{t-1}$
    Scale down by $G^{t-1} \backslash G^t$
**ELSE**
    **IF** *currentAssignmentDoesNotViolateTheSLA*()
        return
    **ELSE**
        Trigger a rebalance/reassignment
    **END IF**
**END IF**

In essence, at each decision interval the *Controller* samples the state of the cluster and runs the *LL-FFD* based on the sampled values of partitions lag, partitions arrival rate, and consumers rate to get $G^t$ the set of consumers needed at decision interval *t*. As shown in Algorithm 1 (*AutoScaleCG*), the *Controller* runs the *LL-FFD* to get $G^t$ and produce scaling recommendations accordingly. Precisely, if $|G^t| > |G^{t-1}|$, then a scale up is needed. The set $G^t \backslash G^{t-1}$ denotes the set of the consumers to be added. Similarly, if $|G^t| < |G^{t-1}|$, then a scale down is needed. The set $G^{t-1} \backslash G^t$ denotes the set of the consumers to be removed. Otherwise, if $|G^t| = |G^{t-1}|$ then neither an addition nor removal of consumers is needed. Still, we check if the current assignment of partitions to existing consumers violates the latency SLA. In the positive case, we trigger a rebalance/reassignment so that the suggested assignment as outputted by the *Controller* takes place. Otherwise, if the existing consumers-partitions assignment does not violate the latency SLA, the scale logic exits with no action as shown in Algorithm 1 (*AutoScaleCG*). *AutoScaleCG* shows how the *Controller* calls the *LL-FFD* to calculate $G^t$, and next produces scaling recommendations accordingly as discussed above. Note that the algorithms for the procedures *Least-LoadFirstFitDecreasing* and *currentAssignmentDoesNotViolateTheSLA*() are shown in the Appendix section.

## 3.3 The load-aware assignment of consumers in the group to queue partitions

As discussed in subsection 3.2, the *Controller* executes the procedure *AutoScaleCG* at each decision interval to decide on the number of consumers needed to guarantee the latency SLA and their assignment to partitions. However, the role of the *Controller* is restricted to launching or removing event consumer replicas out of the consumer group. In fact, after that the *Controller* removed or added consumers into the existing set of consumers, a rebalancing/assignment process will implicitly take place. As mentioned before, rebalancing is the process of assigning partitions to consumers. It is performed by one of the consumers. Precisely, as shown in Figure 3, upon rebalancing all the existing consumers are requested to re-join the consumer group. As per our modification to the group membership protocol, we have modified the rebalancing logic so that the consumer group leader contacts the *Controller* for its recommended load-aware assignment. This process is shown in Figure 3 where the consumer group leader upon rebalancing calls the *Controller* for the load-aware assignment of consumers to partitions. In Figure 3, all the modules in red are newly added to the event queue for load-aware autoscaling.

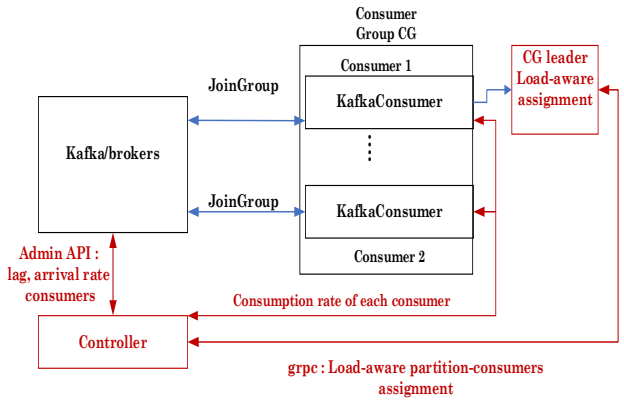


**Figure 3 : The CG leader calls the Controller for its recommended assignment of consumers-partitions. All the red modules are newly added to the architecture for autoscaling of workload-skewed distributed event queues.**

## 4 EXPERIMENTAL WORK

In this section we report some of the experiments we performed using the bin pack autoscaler. We also discuss the workloads we used in the experiments. All the experiments were performed on Google Cloud using a Kubernetes cluster composed of 5 virtual machines of type e2-standard-2 (2vcpu, 8GB RAM), Kubernetes version 1.20.6-gke.1400 and Kafka version 2.7. As discussed in section 2, we deployed an *Order* microservice that acts as a workload-skewed event producer, and a *Payment* microservice that acts as a latency-sensitive event consumer microservice configured with a maximum event processing latency $W_{SLA}$ of five seconds. The *Payment* consumer microservice operated at 100 events/seconds (using *Thread.sleep*). Throughout the experiments we used an event queue with 5 partitions. The decision interval is defaulted to one second unless otherwise stated.

### 4.1 Binpack autoscaling of a workload-skewed event consumer with load-aware assignment, compared to a non-load-aware threshold-like autoscaler

In the first set of experiments, we leveraged a 10-minute skewed workload with an event queue of 5 partitions namely P0, P1, P2, P3, P4. In the first two minutes the arrival rate into each partition is set to 15 events per second for a total of 75 events/sec. Next, we increased the arrival rate into P0 and P1 at rate of 1 events/sec until they reached 60 events/sec in 45 seconds (time 165 seconds). The arrival rate into P0 and P1 remains at 60 events/sec thereafter. At minute 4, we increased the arrival rate into P2 at rate of 1 event/sec to reach 60 events/sec in 45 seconds (time 285 seconds), P2 remains at 60 events thereafter. Similarly, at t=6 minutes, we increased P3 events arrival at a rate of 1 event/sec to reach 60 events/sec in 45 seconds (time 405 sec), P3 remains at 60 events/sec until the 8th minute where all partitions arrival rates fall back to 15 events/second. In this experiment we used a 1 second decision interval.

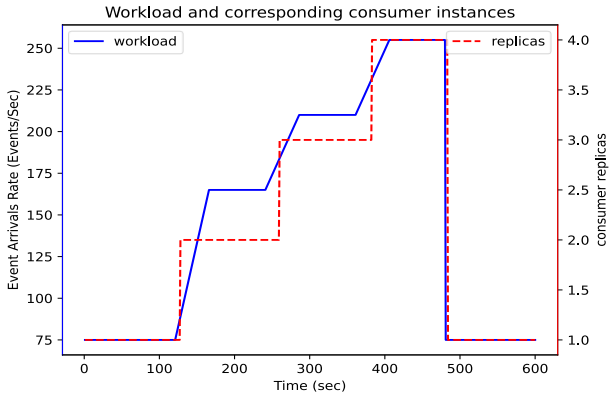


**Figure 4 : Workload total arrival rate and corresponding scaled replicas.**

Figure 4 shows the workload total arrival rate as well as the corresponding scaled replicas and their timings. In fact, we repeated this autoscale experiment twice. In the first trial we ran the bin pack autoscaler complemented with a load-aware assignment as suggested by our newly designed *Controller*. In the second trial, we wanted to emulate a simple threshold-like autoscaler without load-aware consumers-partitions assignment. Hence, we leveraged a *Controller/Autoscaler* that scales the event consumer group exactly similar to the first experiment. However, instead of the load-aware assignment as suggested by our work, we leveraged the default rebalancing/assignment logic in Kafka namely the *Incremental Cooperative*. This last promotes stickiness and data locality i.e., sticking consumers to their already assigned partitions without taking care of load-awareness. In both experiment trials, the scaled replicas are shown in Figure 4 . Note that the consumption rate of consumers is reported dynamically to the *Controller* through an RPC call issued by the *Controller*. The reported event consumption rate to the *Controller* was slightly less than 100 events/sec and on average equivalent to 95 events/second, due to the overhead.

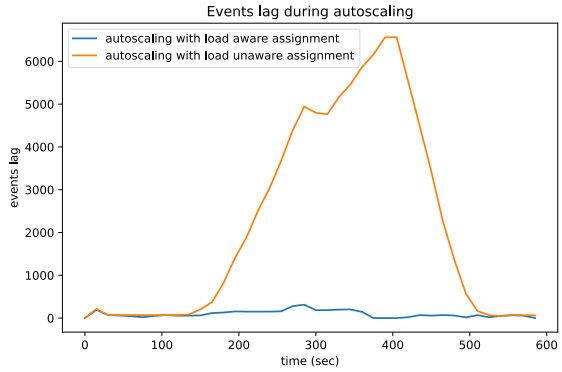


**Figure 5 : : Events backlog during autoscaling a workload-skewed event consumer both with load-aware (newly designed) and incremental sticky assignment (Kafka default).**

Figure 5 shows the difference in events backlog when running the bin pack autoscaler complemented with a load-aware assignment

as compared to when leveraging the default Kafka assignment logic, namely the *Incremental Cooperative*. As shown, in the first case the event lag was always in the region of few tens, thus respecting the 5 seconds SLA latency for the whole time of the experiment. This translates to 100% latency SLA guarantee at the cost of 18 replica.minutes. At this same cost of replica.minutes, the threshold-like autoscaler complemented with the non-load-aware *Incremental Cooperative* assignment logic exhibited a lag of up to 6K, and hence only 44% of total events fallen below the 5 seconds latency and respected the SLA.

This shows the importance of load-aware consumers-partitions assignment during autoscaling of skewed workloads. As such, promoting load-awareness over data locality when the workload is skewed will have better overall effect on the latency SLA. This is true even if the partitions that will be reassigned to new consumers while leveraging load-aware rebalancing will exhibit a short time of event consumption blockage. This short time of consumption blockage happens during the partitions reassignment process where the partitions are first revoked by their old consumers and reassigned to their new consumers. In all experiments, the 90$^{th}$ percentile of the duration of the partitions reassignment process during which event consumption is blocked was less than 1 second.

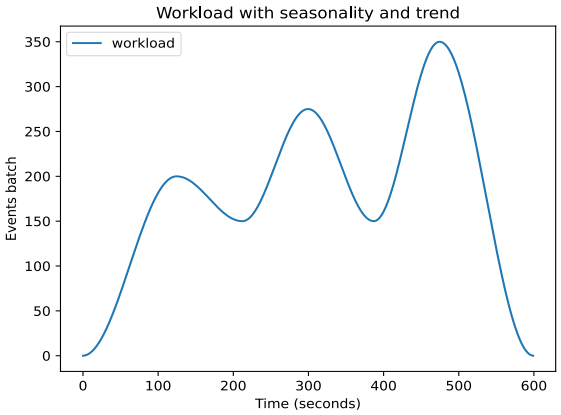


**Figure 6 : Workload used in the second experiments.**

## 4.2 Binpack autoscaling with a second skewed workload and the effect of the decision interval on the binpack autoscaler

In this set of experiments, we used the same workload employed in the research work of [11] as shown in Figure 6. To introduce skewness in the workload, for each batch of events in the workload, we sent 27% of events to P0 and 27% to P1, while we sent the 46% remaining events uniformly to P2, P3, P4. 27% was selected in order to guarantee that over the lifetime of the workload, the arrival rate into each individual partition will not bypass the event consumer rate i.e., 100 events/seconds. In this set of experiments, we employed a cooldown period of 30 seconds. A cooldown period is an interval of time following a scale operation during which no scale action scale is triggered.

In a first experiment, we ran the experiment with the bin pack autoscaler complemented with a load-aware partitions-consumers assignment as described throughout this paper. We set the decision interval to 1 second. Figure 7 shows the scale up and down actions and their timings. In this experiment, we observed that 0 events violated the latency SLA, that is, all the events processing latency fallen below 5 seconds. This came at the cost of 26.1 replica.minutes. On the other hand, overprovisioning dictates to provision 5 replicas for the lifetime of the workload. This translates to 50 replicas.minutes. Hence, the binpack autoscale solution showed 52.2% replica.minutes reduction as compared to the overprovisioning solution.

In a second experiment, we leveraged a threshold-like *Controller/Autoscaler* that scales the event consumer group exactly similar to the first experiment but without load-aware partitions-consumers assignment. Instead, we employed the *Incremental Cooperative* non-load-aware assignment logic. The scaling actions and their timings are similar to those in Figure 7. On the other hand, Figure 8 shows the events lag when the binpack autoscaler ran with a load-aware assignment as proposed in this work, as compared to the non-load-aware *Incremental Cooperative* assignment.

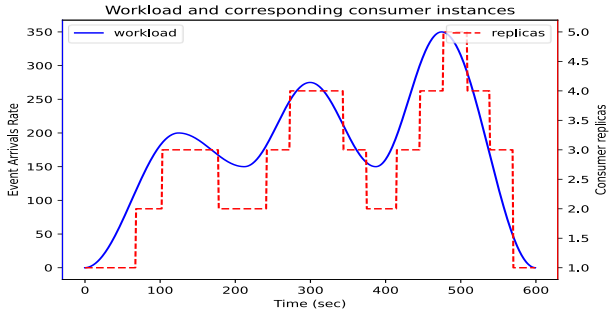


**Figure 7 : Scaling actions and their timings when running the bin pack autoscaler**

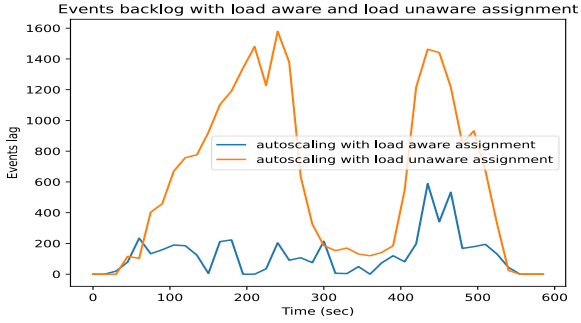


**Figure 8 : Events lag when the bin pack autoscaler is complemented with a load-aware assignment as compared to the case where the bin pack autoscaler is complemented with incremental cooperative assignment.**

As shown in Figure 8, the events backlog when autoscaling while leveraging the *Incremental Cooperative* assignment is much higher as compared to that with load-aware assignment. In fact, with the non-load-aware *Incremental Cooperative* assignment only 77% of events respected the 5 seconds latency SLA. In contrast, when

leveraging the load-aware assignor all the events latency fallen below 5 seconds which translates to 100% SLA guarantee.

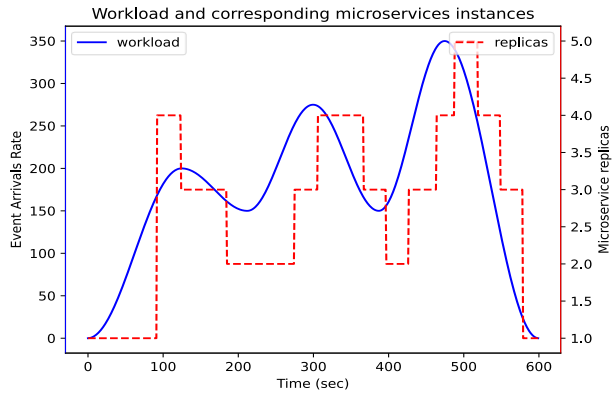


**Figure 9 : Scaling actions and their timings when running the bin pack autoscaler at 30 seconds decision interval.**

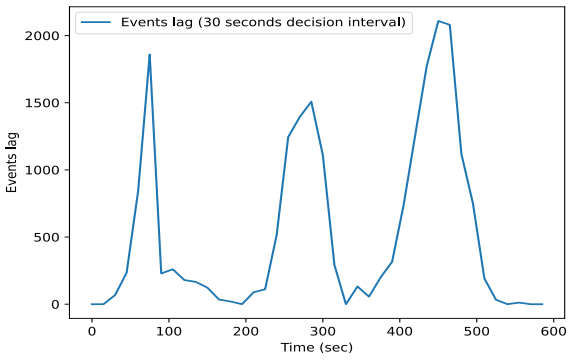


**Figure 10: Events lag when running the bin pack autoscaler with load-aware assignment at 30 seconds decision interval.**

In a third experiment we ran the binpack autoscaler with load aware assignment at a decision interval of 30 seconds instead of one second. The scaling actions and their timings are shown in Figure 9. Notice how at time 95 seconds the replicas scaled from 1 to 4 in one shot. This is because with 30 seconds decision interval, the lag has accumulated and the bin pack scaler recommended 4 replicas for being able to serve this lag in $W_{SLA}$ i.e., 5 seconds. Similarly, notice in Figure 10 how at time 95 seconds the lag dropped in few seconds to a non SLA-violating value. In this experiment the percentage of events that respected the 5 seconds SLA latency was equivalent to 86%.

In fact, throughout the experiments we observed that when the decision interval is small, e.g., less than 10 seconds, partitions lag is typically small and the bin pack autoscaler recommendation on the number of replicas is mostly driven by the partitions arrival rate. On the other hand, when the decision interval is high, e.g., 30 seconds, partitions lag is typically high and the bin pack autoscaler recommendation on the number of replicas is driven by the partitions lag. For example, the first decision to scale from 1 to 4 in Figure 9 is mainly influenced by the partitions lag. In contrast, in Figure 7 where the decision interval was 1 second, the intial transition from 1 to 2 replicas was mainly influenced by the partitions arrival rates.

## ACKNOWLEDGMENTS

This research is funded by the enterprise HighTech Payment Systems HPS and ANRT through a PhD CIFRE collaboration with UCA and CNRS I3S laboratory.

## REFERENCES

[1] Ken Goodhope, Joel Koshy, Jay Kreps, Neha Narkhede, Richard Park, Jun Rao, and Victor Yang Ye. Building LinkedIn's real-time activity data pipeline. IEEE Data Eng. Bull., 35(2), 2012.
[2] M. Mohammadi, A. Al-Fuqaha, S. Sorour, and M. Guizani. Deep learning for iot big data and streaming analytics: A survey. IEEE Communications Surveys & Tutorials, 20(4):2923– 2960, 2018.
[3] K. Al-Aubidy, A. Derbas, and A. Al-Mutairi. Real-time healthcare monitoring system using wireless sensor network. International Journal of Digital Signals and Smart Systems, 1(1):26–42, 2017.
[4] R. Laigner, K. Marcos , D. Pedro, B. Leonardo, C. Carlos, M. Lemos, A. Darlan, L. Sérgio and Y. Z. Yongluan, "From a monolithic big data system to a microservices event-driven architecture," IEEE 46th Euromicro Conference on Software Engineering and Advanced Applications, pp. 213-220., 2020.
[5] R. Laigner, K. Marcos , D. Pedro, B. Leonardo, C. Carlos, M. Lemos, A. Darlan, L. Sérgio and Y. Z. Yongluan, "From a monolithic big data system to a microservices event-driven architecture," IEEE 46th Euromicro Conference on Software Engineering and Advanced Applications, pp. 213-220., 2020.
[6] Q. Xiang, P. Xin, H. Chuan, W. Hanzhang, X. Tao, L. Dewei, Z. Gang and C. Yuanfang, "No Free Lunch: Microservice Practices Reconsidered in Industry," arXiv preprint arXiv:2106.07321, 2021.
[7] C. Richardson, "Building microservices: Inter-process communication in a microservices architecture," 24 July 2015. [Online]. Available: https://www.nginx.com/blog/building-microservices-inter-process-communication/. [Accessed 6 June 2022].
[8] Amazon Kinesis. https://aws.amazon.com/kinesis/.
[9] Google Cloud Pub/Sub. https://cloud.google.com/pubsub/
[10] Microsoft Event Hubs. https://azure.microsoft.com/en-us/ services/event-hubs/
[11] 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.
[12] KEDA, Kubernetes-based event-driven autoscaling, https://keda.sh/.
[13] Wang Y, Lyu B, Kalavri V. The non-expert tax: quantifying the cost of auto-scaling in cloud-based data stream analytics. InBiDEDE@ SIGMOD 2022 Jun 12 (pp. 7-1).
[14] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data Center TCP (DCTCP)," in SIGCOMM, 2010.
[15] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "PACMan: Coordinated Memory Caching for Parallel Jobs," in NSDI, 2012.
[16] W. Reda, M. Canini, L. Suresh, D. Kostić, and S. Braithwaite, "Rein: Taming Tail Latency in Key-Value Stores via Multiget Scheduling," in Twelfth European Conference on Computer Systems. ACM, 2017. doi: 10.1145/3064176.3064209 p. 95–110.
[17] N. Narkhede, G. Shapira and T. Palino, Kafka: the definitive guide: real-time data and stream processing at scale, O'Reilly Media, Inc., 2017.
[18] G. Shapira , T. Palino, R. Sivaram and K. Petty, Kafka: The Definitive Guide Real-Time Data and Stream Processing at Scale, second edition, O'Reilly Media, Inc., 2021.
[19] S. BLEE-GOLDMAN, "From Eager to Smarter in Apache Kafka Consumer Rebalances," Confluent, 11 5 2020. [Online]. Available: https://www.confluent.io/blog/cooperative-rebalancing-in-kafka-streams-consumer-ksqldb/. [Accessed 29 6 2022].
[20] C. Qu, . R. N. Calheiros and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey.," ACM Computing Surveys (CSUR), vol. 51, no. 4, 2018.
[21] S. A. Baset, "Cloud SLAs: present and future," ACM SIGOPS Operating Systems Review, vol. 46, no. 2, pp. 57-66, 2012.
[22] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. 2020. A Survey on the Evolution of Stream Processing Systems. (08 2020). https://arxiv.org/pdf/2008.00842.pdf
[23] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three Steps is All You Need: Fast, Accurate, Automatic Scaling Decisions for Distributed Streaming Dataflows. In

Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation
[24] Mei, Luwei Cheng, and Vanish et al. Talwar. 2020. Turbine: Facebook's Service Management Platform for Stream Processing. In 2020 IEEE 36th International Conference on Data Engineering (ICDE). 1591–1602.
[25] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. 2017. Dhalion: Self-Regulating Stream Processing in Heron. Proc. VLDB Endow. 10, 12 (aug 2017), 1825–1836
[26] Fang, J., Zhang, R., Fu, T.Z., Zhang, Z., Zhou, A. and Zhu, J., 2017, June. Parallel stream processing against workload skewness and variance. In Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (pp. 15-26).
[27] Dósa, György (2007). "The Tight Bound of First Fit Decreasing Bin-Packing Algorithm Is FFD(I) ≤ 11/9OPT(I) + 6/9". Combinatorics, Algorithms, Probabilistic and Experimental Methodologies.
[28] Ajiro Y, Tanaka A. Improving packing algorithms for server consolidation. Inint. CMG Conference 2007 Dec 2 (Vol. 253, pp. 399-406).

## Appendix

For the sake of completeness, we provide the algorithms for the *Least-Load First Fit Decreasing FFD* bin pack algorithm. We also provide the algorithm for the function *currentAssignmentDoesNotViolateTheSLA()*.

***CurrentAssignmentDoesNotViolateTheSLA()***

*Set* $G^{t-1}$ *to the existing set of consumers*
***For*** *each* $c_j^{t-1}$ *in* $G^{t-1}$ ***do***
  ***IF*** $lag^t_{c_j^{t-1}} > \mu \times W_{SLA}$ *OR* $\lambda^t_{c_j^{t-1}} > \mu$
    *return* ***false***
  ***END IF***
***END FOR***
*return* ***true***

***Least-Load FirstFitDecreasing*** $(\lambda_P^t, lag_P^t, W_{SLA})$

*Let* $G^t$ *the set of consumers needed at time t, initially empty*
*Add a consumer* $\boldsymbol{c_0^t}$ *to* $G^t$
$c_0^t.remainingLagCapacity = \mu \times W_{SLA}$
$c_0^t.remainingArrivalCapacity = \mu$
$lag_P^t = \{lag_{p_1}^t, lag_{p_2}^t, \dots, lag_{p_n}^t\}$
$\lambda_P^t = \{\lambda_{p_1}^t, \lambda_{p_2}^t, \dots, \lambda_{p_n}^t\}$
***Sort*** *the set* $\lambda_P^t$ *in decreasing order*
***While true***
  *Remove any previous partitions-consumers assignment*
  ***For*** *i = 0 to* $\lambda_P^t.size()$ *-1* ***do***
    ***Sort*** $G^t$ *in ascending order*
    ***For*** *j = 0 to* $G^t.size()$ *-1* ***do***
      ***IF*** $c_j^t.remainingLagCapacity \geq lag_{pi}^t$ *and* $c_j^t.remainingArrivalCapacity \geq \lambda_{p_i}^t$
        $c_j^t.assign(p_i)$
        $c_j^t.remainingLagCapacity -= lag_{pi}$
        $c_j^t.remainingArrivalCapacity -= \lambda_{p_i}^t$
        ***break***
      ***End IF***
    ***END FOR***
    ***IF*** $j == \boldsymbol{G^t.size()}$
      *Add a consumer* $c_j^t$ *to* $G^t$
      $c_j^t.remainingLagCapacity = \mu \times W_{SLA}$
      $c_j^t.remainingArrivalCapacity = \mu$
      ***break***
    ***End IF***
  ***END FOR***
  ***If*** $i == \lambda_P^t.size()$
    ***break;***
  ***END IF***
***END While***
***return*** $G^t$