Auto-sizing for Stream Processing Applications at LinkedIn

Rayman Preet Singh, Bharath Kumarasubramanian, Prateek Maheshwari, and Samarth Shetty
LinkedIn Corp
{rmatharu, bkumarasubramanian, pmaheshwari, sshetty}@linkedin.com

Abstract

Stream processing as a platform-as-a-service (PaaS) offering is used at LinkedIn to host thousands of business-critical applications. This requires service owners to manage applications' resource sizing and tuning. Unfortunately, applications have diverged from their conventional model of a directed acyclic graph (DAG) of operators and incorporate multiple other functionalities, which presents numerous challenges for sizing. We present a controller that dynamically controls applications' resource sizing while accounting for diverse functionalities, load variations, and service dependencies, to maximize cluster utilization and minimize cost. We discuss the challenges and opportunities in designing such a controller.

1 Introduction

Stream processing systems such as Spark Streaming [39], Flink [3], Heron [31], Samza [35], and Puma [23] are widely employed in the industry to cater to a wide variety of real-time data processing applications [3, 23, 31, 35]. Examples of such applications at LinkedIn include those that serve user-notifications [35], monitor service latencies [35], detect fraud/abuse [11], and many others [35].

At LinkedIn stream processing is offered using the *platform as a service (PaaS)* model to application developers, data scientists, and other users. This allows for increased development agility because users focus solely on devising their applications, while the service bears the onus of maintaining scalability, availability, resource efficiency, security, isolation, fairness, and other properties for the applications. LinkedIn's stream processing service today hosts thousands of such applications, processing millions of messages per second and hundreds of gigabytes of data per second.

This approach requires the service to manage the applications' resource sizing, such as, the amount of memory and CPU assigned, and other tuning, such as, the level of parallelism, heap memory allocations, and other parameters [20]. In doing so, the service aims to meet the applications' throughput and latency requirements in the face of input load variations, variations in the performance characteristics of applications' dependencies such as Key-Value, blob storage, or other web services, environmental variations such as network latency increases, application and hardware evolution, while maximizing cluster utilization and minimizing cost.

While over-allocation of resources to applications (or *over-sizing*) can help tolerate input load increases, it leads to lower resource utilization and higher cost. Similarly, under-sizing leads to decreased throughput and/or increased latency, or worse, result in processing stalls causing high tail-latency. The solution therefore, similar to other cloud services, lies in dynamically sizing applications [30, 33, 34, 36]. Note that, such sizing decisions are different from those required for resource placements, which are typically delegated to cluster-wide resource schedulers like YARN [4], Kubernetes [13], or Mesos [29] that bear the onus of managing isolation, fairness, and fragmentation of cluster resources [25, 28].

Existing work on scaling controllers for stream processing systems focuses on scaling parallelism for meeting applications' latency and throughput goals [20, 26, 27, 30, 32, 34, 38] while modeling applications as a DAG of operators. However, as we detail in Sections 2 and 3, production stream applications have diverged from this model in multiple ways causing them to have heterogeneous performance characteristics. For example, applications use remote services, maintain state, use user-defined functions and external frameworks [10, 18], and combine different functionalities. Consequently, only tuning applications' parallelism without taking into account other sizing parameters, service dependencies, and environmental variables, typically leads to under-sized applications causing lowered throughput, increased latencies, and processing stalls.

We analyze the wide variety of production stream processing applications at LinkedIn, and formulate the key challenges in building a controller that *right sizes* production stream applications (Sec 3). To explore these challenges concretely, we prototype *Sage*, an auto-sizing controller for *continuous-operator* stream processing systems like Samza [35] and Flink [3]. Sage is a work-in-progress that uses a rule-based approach to make sizing decisions about production applications, while accounting for applications' heterogeneous per-

formance characteristics, their dependency on other services, and correlations in their use of CPU and memory resources.

2 Background and Related Work

Stream processing systems can broadly be classified into two types – (i) *bulk-synchronous parallel* (BSP) [37] systems such as Spark Streaming [39] and Google-Dataflow [19], and (ii) long-running or *continuous operator* (CO) [37] such as Apache Flink [3] or Apache Samza [6, 35]. BSP systems convert streams into *micro-batches* (e.g., based on time) that are scheduled for processing by the underlying processing framework. In contrast, CO systems host long-running computations that directly consume streams from Kafka [5], EventHub [14], and other sources. Both types of systems rely on input checkpoints for fault tolerance and offer comparable correctness guarantees (e.g., at least-once [35], exactly once [22]). However, CO systems provide lower latency [37], in our experience are relatively easier to operate in production.

Multi-tenant CO systems typically use cluster-managers to manage the *allocation* of cluster resources to applications. Examples of such cluster-managers include Kubernetes [13], Borg [36], YARN [4], and Mesos [29]. Applications are allocated resources in equal-sized units called *containers*, with each container mapping to a subset of memory and CPU of a host's resources. Applications bear the onus of requesting the appropriate amount of resources, such as, the number of containers, container memory and CPU-cores size. In addition, applications also specify certain other parameters such as heap and native memory allocation (for Java applications), paralellism-related parameters [30], and other parameters required by the underlying CO system [20]. We collectively refer to these parameters as ***sizing parameters***.

Existing work on resource sizing for stream applications models them as combinations of well-known operators (such as, map, filter, join, etc) organized as a DAG, and use the model to tune their parallelism [20, 26, 27, 30, 32, 34, 38], e.g., using linear throughput models [34]. However, as Figures 1 and 2 show, stream applications have evolved beyond this DAG to have multiple additional functionalities. As described in Sec 3, applications are heterogenous in their combination of these functionalities, due to which tuning is required for all their sizing parameters in addition to parallelism. Other work [27, 32] uses queuing theory to tune parallelism but assumes specific distributions of arrival rates and service times, e.g, Poisson, exponential. However, applications can have significantly different distributions (e.g., multi-modal, heavy-tailed) as shown in Figures 3 and 4 for selected representative applications. Therefore such model-based approaches have limited applicability for our application workloads.

3 Challenges and Requirements

Stream processing applications are conventionally modeled as a DAG of operators with predictable latency and throughput characteristics [26, 34, 38]. However, applications have

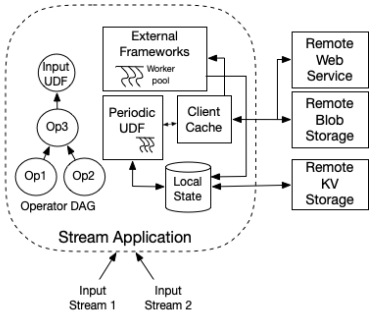

Figure 1: Overview of a sample stream processing application.

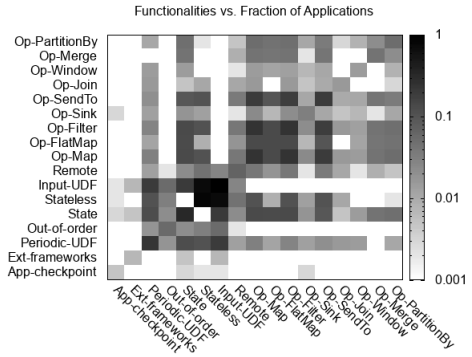

Figure 2: Number of stream applications (normalized) that combine different functionalities, measured two functionalities at a time.

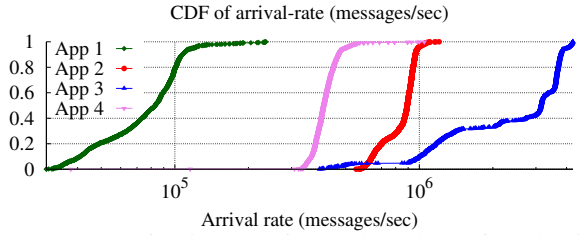

Figure 3: CDF of arrival rate for a selected set of production applications.

evolved from this model in many ways, due to which their resource footprint, performance, and workload characteristics vary significantly and present the following challenges.

Applications using remote services Figure 1 shows the different components of a stream application. For processing an input tuple, in addition to their operator DAG, applications may invoke operations on a remote KV-storage, blob-storage, or other services (either synchronously or asynchronously). For example, an application processing user-profile update streams, may invoke services that control user-notifications, detect suspicious content, monitor duplicates in streams, update metadata related to user interactions, etc. For efficiency, applications may buffer reads/writes to batch them. Moreover, applications may maintain local state (such as for counters and windows) using persistent stores like RocksDB [15], which may be replicated to remote storage [35].

The service-time and throughput of such an application depends on input load variations (illustrated in Figure 5) and on variations in the latency and error-rates (or retries) of

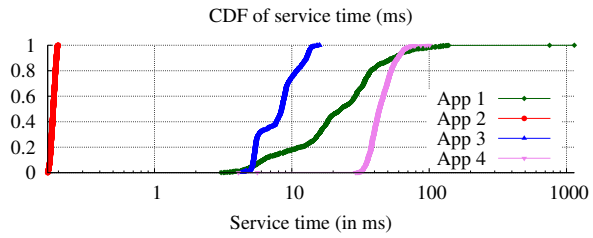

Figure 4: CDF of service time for a selected set of production applications.

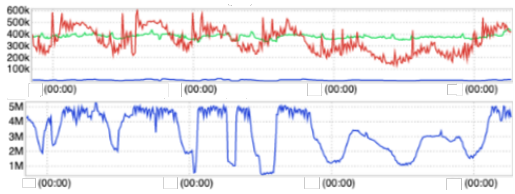

Figure 5: Time series of arrival rates for selected applications (App 1-3 above, App-4 below) for a random 1-week period.

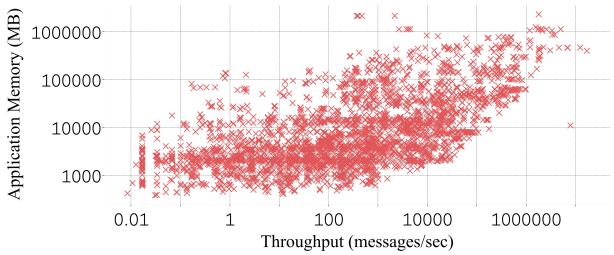

Figure 6: Applications' memory footprint (in MB) vs. throughput (in messages/sec).

the remote service. Latency and error-rate increases can last from a few seconds to several minutes. The controller should therefore be able to differentiate such scenarios, because while scaling up parallelism may reduce backlog caused due to input load increase, it may be ineffective, wasteful and even detrimental [34] in case of a faltering remote service. Figure 4 shows the service-time of an application using remote services (labeled *App 1*) compared to that using only an operator DAG (*App 2*) comprised of the partition-by, map, flat-map, send-to and merge operators. App 1 shows a relatively high variance in its service time due to its dependence on a remote service.

Heterogeneous applications Figure 2 illustrates the number of applications that use combinations of different functionalities. Observe that, applications vary widely in their use of different stream processing semantics, functionalities and capabilities. For example, applications invoking remote services (denoted *Remote*), employing conventional operators (denoted *Op-PartitionBy*, *Op-Map*, *Op-Join*, and so on), often use user-defined functions to process input (denoted *Input-UDF*) as part of their DAG, maintain state, and often also replicate the state to a remote storage service.

Applications can be configured to process sequential input out-of-order to improve throughput. They can also periodically execute UDFs exclusive of input processing (denoted *Periodic-UDF* [16] in Figure 2), for example, to detect duplicates/anomalies in buffered input, produced windowed output, update or prune state, or invoke a remote service.

Applications often control input checkpoint boundaries instead of using periodic checkpointing (denoted *App-checkpoint*), for example, to synchronize checkpoints with remote service calls. Lastly, some applications load and use other frameworks/libraries like TensorFlow [18], DL4j [10] (denoted *Ext-frameworks* in Figure 2).

The heterogeneity among applications alters their resource demands and their service time characteristics, as compared to operator-DAG applications. Figure 6 shows the average memory used and message throughput of the applications, measured over a random 20 minute interval. Observe that, two applications with similar throughput have significantly different memory footprint. Similarly, applications with similar memory footprint yield significantly different throughput. Moreover, as shown in Figure 7, applications also vary significantly in the number of input streams, the size of state for stateful applications, and median service time.

Correlations in resource use Stream applications run continuously under variable input load, that is, variable input message-rate or input byte-rate or both. Therefore, they have inherent dependencies between their resource footprint, their throughput and service time characteristics. For instance, an application under-sized on CPU, may exhibit increased input buffering thus leading to increased memory use, lowered throughput and backlog buildup. In case of Java-based systems like Flink [3] and Samza [6], an application under-sized on heap memory may exhibit frequent GC runs, thus leading to increased service times, reduced throughput and increased CPU utilization. Therefore, an autosizing controller should account for such correlations when making sizing decisions.

Given these challenges, we formulate the following *requirements* for an autosizing controller.

- **Right size** It should generate parameter values for an application's memory, CPU, heap/native allocation, thread-pool sizes, and other parameters, such that, (i) the latency and throughput of the application meet the specified upper and lower bounds, respectively, and (ii) the overallocation of cluster resources to the application is bounded.
- **Sizing time** It should minimize the time taken to reach right size values for an application.
- **Operational ease** The controller's actions should be *interpretable*, allowing reliability-engineers to understand and manually modify sizing if required. Similarly, it should minimize issuing actions that cause processing stalls, throughput drops, latency increases, for example, it should not employ an *undo-action* strategy [26].
- **Other Requirements** It should scale to a large number of applications, be fault-tolerant and resource efficient.

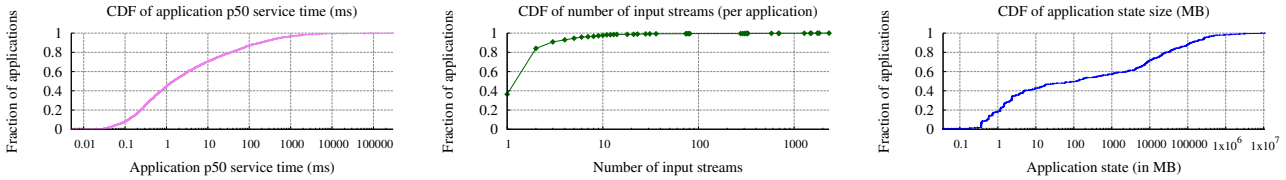

Figure 7: Distribution of median service time (ms), number of inputs, and state size (MB) across applications at LinkedIn.

4 Sage Auto-sizing Controller

Design approach One approach for an auto-sizing controller for streaming applications is to treat applications as black-boxes and define independent rules for tuning of each resource. This is akin to VM-count based auto-scaling provided by cloud services like Azure-VMSS [8] and AWS-EC2 [9]. This approach however, (i) disregards tuning application parallelism (Sec 2), and (ii) does not account for correlations in resource use of applications (Sec 3), and can consequently exhibit increased sizing time and oscillations [30, 34].

In contrast, optimization-based approaches rely on (a) profiling applications using trial-runs to generate models [20, 21, 28, 34], (b) assumptions about the optimization problem (e.g., convexity [21]), and (c) additional parameter-tuning, such as, for step-sizes [20], termination criterion [20]. They lack the interpretability of the first approach and do not account for dynamic changes of an application's service dependencies and that in the environment such as network latencies.

Therefore, in designing Sage, we choose a *gray-box* approach that relies on a pre-defined set of rules tailored to streaming applications – thus providing operational ease, navigates cyclic-dependencies in resource use of applications, while reusing existing work on tuning parallelism (Sec 2).

Sage components Figure 8 shows the different Sage components and its use for Samza [6] applications. Each application comprises of a master container that requests resources from the cluster-manager, and manages membership of data-processing containers that process a subset of inputs data streams' shards [6]. Sage comprises of components typical to a feedback-based controller. The *data collector* receives and aggregates metrics from applications, concerning resource use, container failures, and input backlog. The *resizer* converts sizing decisions issued by policies, to a *sizing action* – a change to one or more sizing parameters, while accounting for cluster capacity, cluster fragmentation, resource quotas, and resource limits of a single host. For example, a policy may decide to set an application's memory to 100 GB, the resizer maps it to 25 4GB containers. Similarly, a policy's decision to set parallelism to 2000 threads, is mapped to 125 containers with 16 threads each (determined by number of cores per host). Each sizing decision and action is recorded in a durable, time-ordered, per-application *action log*, allowing offline auditing and use by policies.

Policy engine Our approach involves encapsulating strategies for sizing a specific resource into a *policy*. Policies are applied periodically to each application in a priority order tailored for CO stream systems (detailed below). Each policy yields at most one sizing decision which is converted to an action using the resizer. Sage currently applies a policy to an application only if (a) no higher priority policy has yielded a sizing decision, and (b) there is no action in-flight for that application. The priority order allows simple design and implementation of an individual policy because, when invoked, it can rely on the policies above it in the priority order to hold. For example, a policy that increases an application's parallelism can assume that the application is not bottlenecked on CPU. Moreover, serializing sizing actions means Sage alters one resource at a time per application. This allows actions to be easily interpreted, audited, and altered by operators if needed, at the cost of a minimal increase in sizing time.

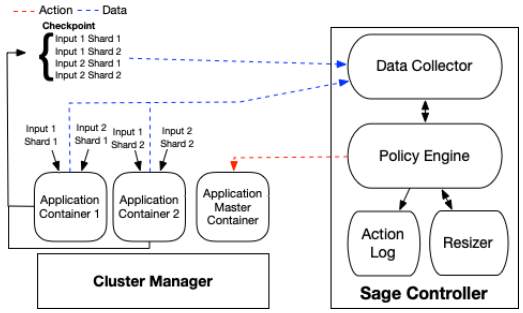

Figure 8: Sage design overview.

Tuning memory before CPU & parallelism Applications can be assigned a memory size using heuristics and models based on their operator DAG. However, as detailed in Sec 3, in addition to their operator-DAGs, applications require memory for other data processing, and are likely to experience memory pressure. An application under memory pressure will typically exhibit high latency and frequent processing-stalls, for example, due to frequent GC occurrences (in Java-based applications), or increased disk-IO in applications using RocksDB [35] for hosting state.

Increasing the parallelism of such an application would increase the service rate and the amount of memory required for the increased rate. Consequently, the amount of memory available to buffer input decreases, thus limiting the application's throughput and resulting in backlog increases. Now, since CO systems use bounded input buffers, increasing applications'

memory allocations alleviates high latency and processing-stalls due to memory pressure. Sage therefore assigns a higher priority to memory policies than CPU and parallelism.

For Java-based systems, memory tuning can be optimized further by assigning a higher priority to a heap-allocation policy than that for total memory. This allows tuning heap allocation before increasing the total memory budget of the application, for example, by ensuring upper bounds on heap-utilization, GC time, and frequency. Table 1 lists the different policies, the priority order, and the metrics they use. Policies are divided into scale-up and scale-down categories.

Bounded backlog increase rates The CPU scale-up policy (P3) alleviates CPU-time bottlenecks by increasing the application's CPU allocation. However, assigning it a higher priority than a parallelism scale-up policy (P4), allows the later to assume that, if an application has a throughput bottleneck then it is not due to memory or CPU under sizing. This allows for a simple policy implementation which monitors for monotonic increases in a sliding window of an backlog values and suitably increases the total thread-count for the application. The window size controls bounds the maximum rate at which an application's backlog can increase, the degree of overallocation and sizing time the policy incurs. The thread-count assignment to different operators and UDFs is further optimized using the three-steps-calculation strategy [30].

Applications using remote services For such applications, the policy correlates the backlog window values with the services' latency and throughput values (typically published by the services). This is because, in addition to a parallelism bottleneck, backlogs can also result from throughput drops or latency increases of a remote service [34]. Time-lagged cross-correlation with pre-configured lag and threshold values are used to measure correlation. Parallelism is increased only in case of non-correlated backlog increases. We are currently investigating other ways of determining correlations [24].

Safe scaledown Scale-down policies work similar to scale-up, and aim to reclaim resources from applications without impacting their performance characteristics. These policies aim to avoid reliance on remedial actions issued by scale-up policies, and hence avoid oscillations due to repeated successive scale-down and scale-up actions. Similarly, they minimize the number of scale-down steps to reduce sizing time; hence do not use additive-decrease strategies.

Reclaiming heap-allocation presents a memory-CPU tradeoff. Assigning relatively low values for heap memory causes frequent GC, increased CPU use, and may increase latencies beyond application requirements. Therefore, to scale-down the heap allocation with a single safe action, Sage currently relies on the heap-committed metric [12]. This metric measures the amount of heap allocated by the JVM to account for live objects and additional space for bookkeeping and GC. By bounding the heap-allocation using heap-committed, the policy minimizes the impact of scale-down actions on the application's existing characteristics, minimizes requiring

Priority	Name	Metrics
P1	Heap-allocation scale-up	Heap-used, GC-time & count
P2	Memory scale-up	Memory-used
P3	CPU scale-up	CPU-used
P4	Parallelism scale-up	Backlog
P5	Parallelism scale-down	Backlog
P6	Heap-allocation scale-down	Heap-committed
P7	Memory scale-down	Memory-used
P8	CPU scale-down	CPU-used

Table 1: Sage's policy list.

remedial actions at the cost of over allocation. Alternative policies may employ reinforcement-learning based strategies, such as those in [36] for batch and "serving" workloads.

Sage's memory scale down policy aims to lower bound the allocated and used memory sizes, using the maximum memory-used and heap-committed [12] metrics. This is because due to zero-page optimizations [1] by the kernel, heap-committed metric may be higher than the resident set size. The policy computes a likely *resident size*, assuming the heap-committed is a part of the application's resident set, which is then used to size the application's memory. The scale-down policies work analogous to their scale-up counterparts.

5 Current Implementation

The Sage prototype is implemented in Java as a Samza application, using an Input-UDF for data collector that indexes metrics into sliding-windows stored using Samza's state API [35] (one store per metric). Since state updates are idempotent using an at-least-once semantics suffices. Likewise, the policy engine is implemented as a periodic-UDF [16] that is executed exclusive of input. Policies P1-P3 use a multiplicative increase for its respective resource. Sage scales horizontally by partitioning the input streams by applications' name.

We have enabled Sage on selected production applications at LinkedIn running on Apache YARN [4] clusters. Sage incurs an average sizing time of approx. 40 minutes (across applications including scale-up and down), with the right-size at most 14% over-sized as compared to a hand-tuned optimal size. It incurs atmost one action for heap and total memory scale-down each, does not exhibit oscillations, and has prevented tens of occurrences of processing stalls. In contrast, an operator may spend several minutes or hours when hand-tuning complex applications.

6 Conclusion

Stream processing systems like Flink [3] and Samza [6, 35] provide low latency but rely on appropriate resource sizing for applications. However, applications have diverged from their conventional model of a DAG of operators, which pose multiple challenges for their resource sizing. Sage is an auto-sizing controller that dynamically controls applications' resource sizing while accounting for input load variations, applications' service dependencies, and environmental variables, while maximizing cluster utilization and minimizing cost.

7 Discussion

Stream-processing is offered as a PaaS service at many web companies [7, 35], and we expect commercial cloud providers to provide such services in the near future. Adoption of programming models like Apache Beam [2] and SQL [7, 17] by streaming systems has allowed applications to evolve in novel ways beyond a DAG of operators, and utilize new frameworks and capabilities, as we have tried to demonstrate from our analysis of production stream processing applications at LinkedIn. However, without appropriate resource sizing, applications typically run into throughput and latency issues, failures, and processing stalls, regardless of the type of streaming system – BSP or CO. In case of CO systems the onus of sizing is on the applications while in case of BSP systems the onus is on the scheduler [37].

The problem of sizing requires navigating multiple resource, performance, operability, and cost trade-offs. Consequently there are multiple possible solution approaches. Sage presents a rule-based solution in this spectrum, that allows us to trade-off a bounded amount of cost for operational ease. Sage brings multiple other practical benefits. For example, Sage stabilizes new applications to run at production scale, allows developers to roll-out new features in existing applications without performing a scaling/tuning exercise, and handles unforeseen traffic increases, such as, due to feature popularity surges, socio-economic events, etc (as shown in Figure 5). Our preliminary results necessitate evaluating Sage using a number of metrics such as, convergence time, the level of under or over-sizing as compared to hand-tuned optimal sizes, and Sage's resource footprint – which we plan to address in future work.

At the same time, our work presents a number of open questions. For example, the problem of formulating strategies for dynamic resource sizing that reduce sizing time by using learned models of applications, preventing oscillations, bounding the over and under use of resources, and accounting for variables like network latency and throughput. Similarly, the problem of formulating a framework for sizing stream applications that is provably safe – does not require remedial actions, non-oscillatory, and accurate, in the face of load and environmental variations remains open. We plan to address these problems in future work. We hope that our analysis of production applications, the challenges they bring, and our work on Sage throws light on current production stream applications and the trade-offs in hosting them over a PaaS service, and proves valuable for other work in this area.

References

[1] Adding a huge zero page. https://lwn.net/Articles/517465/.

[2] Apache Beam. https://beam.apache.org/.

[3] Apache Flink. https://flink.apache.org/.

[4] Apache Hadoop YARN. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[5] Apache Kafka. https://kafka.apache.org/.

[6] Apache Samza. http://samza.apache.org/.

[7] AthenaX, Uber Engineering's Open Source Streaming Analytics Platform. https://eng.uber.com/athenax/.

[8] Autoscale with Azure virtual machine scale sets. https://docs.microsoft.com/en-us/azure/virtual-machine-scale-sets/virtual-machine-scale-sets-autoscale-overview.

[9] AWS Auto Scaling. https://aws.amazon.com/autoscaling/.

[10] Deep Learning for Java (DL4J). https://deeplearning4j.org/.

[11] Defending Against Abuse at LinkedIn's Scale. https://engineering.linkedin.com/blog/2018/12/defending-against-abuse-at-linkedins-scale.

[12] Java Memory Usage. https://docs.oracle.com/javase/7/docs/api/java/lang/management/MemoryUsage.html.

[13] Kubernetes (K8s). https://kubernetes.io/.

[14] Microsoft Azure Event Hubs. https://azure.microsoft.com/en-us/services/event-hubs/.

[15] RocksDB: Persistent Key-Value Store for Fast Storage Environments. https://rocksdb.org/.

[16] Samza Periodic Window API. https://samza.apache.org/learn/documentation/1.0.0/container/windowing.html.

[17] Samza SQL. https://samza.apache.org/learn/documentation/latest/api/samza-sql.html.

[18] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[19] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. The dataflow model: a practical approach

to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. 2015.

[20] Muhammad Bilal and Marco Canini. Towards automatic parameter tuning of stream processing systems. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 189–200, 2017.

[21] Sarah L Bird and BJ Smith. Pacora: Performance aware convex optimization for resource allocation. In *Proceedings of the 3rd USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2011.

[22] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in apache flink®: consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment*, 10(12):1718–1729, 2017.

[23] Guoqiang Jerry Chen, Janet L Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. Real-time data processing at facebook. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1087–1098, 2016.

[24] Roger T Dean and William TM Dunsmuir. Dangers and uses of cross-correlation in analyzing time series in perception, performance, movement, and neuroscience: The importance of constructing transfer function autoregressive models. *Behavior research methods*, 48(2):783–802, 2016.

[25] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

[26] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. Dhalion: self-regulating stream processing in heron. *Proceedings of the VLDB Endowment*, 10(12):1825–1836, 2017.

[27] Tom ZJ Fu, Jianbing Ding, Richard TB Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. Drs: Auto-scaling for real-time stream analytics. *IEEE/ACM Transactions on Networking*, 25(6):3338–3352, 2017.

[28] Adem Efe Gencer, David Bindel, Emin Gün Sirer, and Robbert van Renesse. Configuring distributed computations using response surfaces. In *Proceedings of the 16th Annual Middleware Conference*, pages 235–246, 2015.

[29] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[30] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 783–798, 2018.

[31] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 239–250, 2015.

[32] Björn Lohrmann, Peter Janacik, and Odej Kao. Elastic stream processing with latency guarantees. In *2015 IEEE 35th International Conference on Distributed Computing Systems*, pages 399–410. IEEE, 2015.

[33] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, 12(4):559–592, 2014.

[34] Yuan Mei, Luwei Cheng, Vanish Talwar, Michael Y Levin, Gabriela Jacques-Silva, Nikhil Simha, Anirban Banerjee, Brian Smith, Tim Williamson, Serhat Yilmaz, et al. Turbine: Facebook's service management platform for stream processing. *Traffic*, 40:80.

[35] Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. Samza: stateful scalable stream processing at linkedin. *Proceedings of the VLDB Endowment*, 10(12):1634–1645, 2017.

[36] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. Autopilot: workload autoscaling at google, 2020.

[37] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 374–389, 2017.

[38] Le Xu, Boyang Peng, and Indranil Gupta. Stela: Enabling stream processing systems to scale-in and scale-out on-demand. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*, pages 22–31. IEEE, 2016.

[39] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Presented as part of the*, 2012.