The *Non-Expert Tax*: Quantifying the cost of auto-scaling in Cloud-based data stream analytics

Yuanli Wang Baiqing Lyu Vasiliki Kalavri
Boston University
{yuanliw,baiqing,vkalavri}@bu.edu

ABSTRACT

All major Cloud providers offer *Data Stream Analytics* as managed services, allowing *non-expert* users to easily extract knowledge from data streams. These services lift the burden of job deployment and maintenance off users by provisioning resources automatically. However, this automation often comes at a cost, as auto-scaling policies may choose to over-provision to meet performance goals. We term this cost the *Non-Expert Tax*: the relative error between the ideal cost of deployment if an expert user would carefully configure resource allocation and the actual cost incurred by executing the same job with auto-scaling enabled. We conduct an empirical evaluation study of auto-scaling in two popular Cloud-based data stream analytics services and we find that they aggressively scale out, allocating resources quickly to meet high demand, but are conservative when scaling down, thus, charging users for underutilized resources. We quantify the *Non-Expert Tax* and show it can be as high as 544% for short-term jobs and up to 332% per month for periodic workloads.

CCS CONCEPTS

• **Information systems** → **Stream management**; • **Networks** → **Cloud computing**.

KEYWORDS

Stream Processing, Cloud Computing, Auto Scaling

ACM Reference Format:
Yuanli Wang, Baiqing Lyu and Vasiliki Kalavri. 2022. The *Non-Expert Tax*: Quantifying the cost of auto-scaling in Cloud-based data stream analytics. In *Big Data in Emergent Distributed Environments (BiDEDE'22), June 12, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3530050.3532925

1 INTRODUCTION

Data stream processing allows extracting knowledge from continuous event streams and enables low-latency decision making in a variety of emerging applications, such as business analytics, recommendation systems, IoT, fraud detection, and continuous monitoring [8, 10]. However, modern stream processing frameworks are complex distributed systems that require careful expert configuration and oversight to achieve good performance. In contrast to batch applications, streaming workloads are long-running, dynamic, and often unpredictable, eventually rendering any initial configuration out-of-tune.

To alleviate the complexity of deploying, monitoring, and tuning streaming applications, all major Cloud providers offer data stream analytics as managed services [1, 2, 15, 20]. Users submit their application code and the Cloud-hosted service automatically allocates and adjusts resources according to the dynamic workload. For example, the service might increase the number of allocated CPU cores to keep up with a high event rate or release underutilized resources when the rate decreases. Such auto-scaling features are attractive to users who wish to benefit from the *pay-as-you-go* model of Cloud computing. At the same time, they pose a legitimate concern, as the Cloud provider's incentive to over-provision is in conflict with the user's goal to reduce costs.

Unfortunately, the cost-benefit trade-off of automatic scaling in Cloud-hosted data stream services is not well understood. Documentation on policy decisions is often scarce and might intentionally obscure details about internal design to protect business interests.

Our contributions. In this paper, we present an empirical evaluation study of auto-scaling policies in two popular streaming analytics services: Amazon Kinesis Data Analytics [4] and Google Cloud Dataflow [15]. Using an established streaming benchmark [25], we perform experiments with varying input load and we report performance metrics, resource provisioning decisions, and associated costs. Further, we use carefully hand-tuned deployments to measure the *ideal* cost that each service would incur, if resource provisioning was decided by a perfect oracle. We term the relative error between the actual and ideal cost the *Non-Expert Tax* and we compute it for three workload scenarios: (i) short-term experiments, (ii) periodic rate fluctuations, and (ii) short-lived rate spikes.

Major findings. Interestingly, we find that existing auto-scaling policies increase resources eagerly when scaling out but make conservative and slow decisions when scaling down. As a result, users may be considerably over-charged for under-utilized resources that can incur a monthly *Non-Expert Tax* of over 300%. Further, our experiments show that auto-scaling features are not always effective at eliminating bottlenecks and decisions may rely on misleading metrics. Finally, we identify inconsistencies between the public documentation of auto-scaling features and their observed behavior.

2 BACKGROUND

We first revisit the fundamentals of dataflow stream processing and then describe the main features of Amazon Kinesis Data Analytics and Google Cloud Dataflow.

		Q1		**Q3**		**Q5**		**Q8**	
		high	*low*	*high*	*low*	*high*	*low*	*high*	*low*
Kinesis	***Bids***	200K	1K	-	-	40K	1K	-	-
	Auctions	-	-	50K	1K	-	-	28K	1K
	Persons	-	-	10K	200	-	-	8K	300
Dataflow	***Bids***	1M	5K	-	-	80K	2K	-	-
	Auctions	-	-	100K	2K	-	-	56K	2K
	Persons	-	-	20K	400	-	-	16K	600

Table 1: Target source rate (rec/s) configuration for the Nexmark queries used in the experiments. The rates used for Kinesis do not trigger autoscaling in Dataflow, thus, we increased the rates correspondingly.

Dataflow stream processing. Streaming dataflow programs are represented as logical directed acyclic graphs, where vertices denote operators and edges are data dependencies. Upon deployment, the logical graph is translated to a *physical* execution plan, which maps dataflow operators to provisioned compute resources (or workers). The assignment remains static unless a reconfiguration occurs. Source operators generate records at a rate defined by application data sources (sensors, stock market feeds, etc.). To maximize system throughput, the execution plan must sustain the input rate. If an operator cannot process records at the rate it receives them, it will stall its upstream operators from producing output and the dataflow will experience *backpressure*.

Amazon Kinesis Data Analytics is a service that manages Apache Flink [6] jobs automatically. Flink executes operators in worker processes called *Task Managers* (TM) which are configured with a predefined number of *slots*. A slot represents a fixed subset of resources on the TM and can execute one or more parallel subtasks of dataflow operators. When a user submits a Flink job, Kinesis automatically allocates TMs and slots, uses Flink's built-in strategy for operator assignment, and deploys the application. To submit a job, users need to package their Flink application as jar, upload it to the S3 Storage Service, and provide its path to Kinesis. According to the Kinesis documentation [4], the service continuously monitors CPU utilization to make automatic scaling decisions. In particular, Kinesis promises to scale up (increase parallelism) when the CPU utilization remains at 75% or above for 15 minutes, and scale down (decrease parallelism) when the CPU utilization remains below 10% for six hours.

Google Cloud Dataflow [15] is a service that manages Apache Beam [5] pipelines on the Google Cloud Platform. Upon submitting a job, the service automatically provisions VM instances and supports horizontal scaling [14]. The user can either select a VM instance type or let the Dataflow service choose a reasonable default. Queries are specified in the Apache Beam programming model [5]. Then users can use Gradle to run a provided action which compiles and submits the query to Dataflow. To make scaling decisions, Google Dataflow monitors *backlog*, that is the amount of unprocessed input in bytes or units of time, and CPU utilization. According to the documentation [14], Dataflow makes scaling decisions as fast as *a couple of minutes*. Scale-up is triggered if the estimated backlog time stays above $15s$ and average CPU utilization across workers is above 20%. The policy is supposed to decide to scale down if the backlog is lower than $10s$ and CPU utilization is below 75%. Finally, Dataflow claims to make predictive scaling decisions allowing it to autoscale in advance to meet expected demand.

Other Cloud streaming services. We also investigated the scaling policy of Microsoft Azure Stream Analytics [20] and Alibaba Realtime Compute [1]. Azure Stream Analytics requires users to construct rule-based auto-scaling policies by selecting metrics to monitor and defining thresholds for triggering scale-up and down. It further allows users to set a bound on parallelism or schedule a scaling decision at a specific time. We do not consider the Azure service in this study, as it relies on user input for its scaling policy configuration. Like Kinesis, the Alibaba service manages Apache Flink jobs and relies on a threshold-based policy that is evaluated on continuously collected metrics, such as latency and CPU/memory utilization. By default, the service increases resources if the CPU utilization is above 80% for $6min$ and decreases parallelism when it is lower than 20% for $24h$. As the Alibaba policy is similar to that of Kinesis, we do not include it in this study.

3 EXPERIMENTAL METHODOLOGY

We evaluate the auto-scaling policy of Amazon Kinesis and Google Dataflow using various queries and dynamic workloads. Our goal is to understand *when* scaling decisions are triggered, *what metrics* and thresholds are used to make those decisions, and *how much* additional cost they may impose to users.

Workload. To evaluate the auto-scaling behavior of Kinesis and Google Dataflow, we use four queries form the Nexmark benchmark [7, 25] with various operators and state characteristics. $Q1$ is a lightweight stateless query with map and filter operators, $Q3$ contains a stateful incremental join, $Q5$ includes a sliding window, and $Q8$ is tumbling window join query. We implemented custom source operators that can be configured with a target event rate and running period. Each experiment consists of two stages. In the *high-rate* stage, we start from an under-provisioned state and observe how the service reacts to reach the target rate. After $70min$, we switch to the *low-rate* stage, dropping the event rate to a low value for the subsequent $7h$, and observe how quickly the service releases unnecessary resources. The target rates are shown in Table 1.

Metrics. For Kinesis, we collect the following metrics: (1) *Throughput* as the number of records the source operator emits per second, (2) *backpressured time* as the time (in ms) a source operator is backpressured per second, and (3) aggregate *CPU utilization* across workers. For Google Dataflow, we further collect (4) *backlog* as the estimated total size (in byes) of elements waiting to be processed.

The Non-Expert Tax. We consider a configuration *ideal* if it can keep up with the target rate without inducing backpressure or backlog, while using the minimum amount of resources. We then define the *Non-Expert Tax* as the relative error between the expected (monetary) cost of the ideal configuration and the actual cost charged by the Cloud provider when the auto-scaling feature is enabled:

$$\textit{Non-Expert Tax} = \frac{actual\_cost - expected\_cost}{expected\_cost} * 100\%$$

	Kinesis	**Dataflow**
Policy	Heuristic	Heuristic and predictive
Metrics	CPU utilization	CPU utilization and backlog
Scale-out interval	15 min	10 min
Scale-down interval	6 hours	10 min
Scaling step	Symmetric by a factor of 2	Asymetric and custom

Table 2: A comparison of auto-scaling policies in Amazon Kinesis and Gogole Dataflow.

We empirically identify the ideal configuration for each query by running a series of experiments with an increasing number of workers until we verify the absence of backpressure and backlog.

Configuration. For the Kinesis experiments, we enabled the Auto Scaling option and set the parallelism parameter to default. We further fixed the source and sink parallelism in all queries to 1, to avoid affecting the input rate after scaling decisions. We disabled operator chaining and set the watermark interval to 5s. For Dataflow, we set the maximum number of workers to 32 and use the default number of event generators (100). We select the `n1-standard-1` instance for $Q1$, which is similar to the default Kinesis worker type. However, $Q3$, $Q5$, and $Q8$ terminate with out-of-memory problems when run with this instance type. Thus, we use `n1-standard-2` for these queries.

4 EXPERIMENTAL RESULTS

We first describe the performance and auto-scaling results and then quantify the *Non-Expert Tax* for the two services. Table 2 summarizes the main auto-scaling features and take-aways for Kinesis and Dataflow.

4.1 Auto-scaling in Amazon Kinesis

Figure 1 shows the auto-scaling behavior of Kinesis for Nexmark queries. We plot observed throughput, allocated and used resources, backpressure, and CPU utilization over time. Kinesis auto-scaling decisions are marked with vertical dashed lines.

Q1, shown in Figure 1a, is the simplest query, consisting of stateless map and filter operators. After submission, the query runs for 15min on 1 slot. With this initial configuration, the query cannot reach the target rate of 200K rec/s, experiences high backpressure (50%), and has CPU utilization close to 100%. Kinesis makes 3 scaling decisions (roughly every 15min), before the backpressure disappears and the target rate is reached. At every scaling step, Kinesis stops and restarts the application, doubling the number of slots. The reconfigurations cause brief periods of downtime and corresponding throughput drops. After the third scaling decision, CPU utilization has dropped close to 30% and the dataflow can comfortably keep up with the input rate. Nevertheless, about 69min after the experiment start, Kinesis made another scaling decision, increasing the allocated slots to 18, while only using 16 (2x higher than the previous configuration). At $t = 70min$, the target rate drops to 1K rec/s. However, Kinesis continues executing the query on 16 underutilized slots for the next 6h. The service eventually makes a scale-down decision at 7h15min, releasing 8 slots, though the CPU utilization remains under 5%.

Q5 computes a stream of auction bids over the last 5s with a slide of 1s. Figure 1b shows the collected metrics over time for this experiment. The service makes 3 decisions during the first phase of the experiment, doubling the number of allocated slots at each step. However, in contrast to $Q1$, we observe that latency, backpressure, and CPU utilization remain high even after reaching the target rate. After the rate drops, the service again takes 6h to reduce the amount of allocated slots. At the low rate of 1K rec/s, the dataflow can comfortably keep up with 1 slot. For this query, Kinesis would take 18h to release all unnecessary resources.

Take-away #1: *The scale-up interval is* 15min *but the scale-down interval is* 6h. While Figure 1a plots metrics for the first 8h of the experiment, we empirically confirmed that the scale-down period is 6h. At the low rate of 1K rec/s, the dataflow can comfortably keep up using 1 slot. As a result, Kinesis runs the application in over-provisioned state for 24h before arriving to the ideal configuration.

Take-away #2: *Kinesis may scale up even if CPU utilization is below* 75%. In $Q1$, we observe a scale-up decision when CPU utilization is 30%. This behavior is inconsistent with the documentation [4].

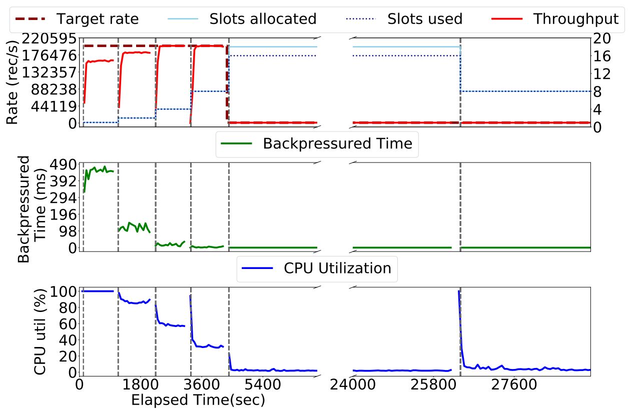
Take-away #3: *The Kinesis policy varies resources by a factor of 2*, doubling the number of slots when scaling up and reducing them to half when scaling down.

Q3 is a two-input query that joins auction events with person events incrementally over time. The throughput and backpressure metrics shown in Figure 1c correspond to the auctions source. In the first phase of the experiment, Kinesis makes 3 scaling decisions, yet without reaching the target rate after 70min with 8 allocated slots. The fourth scale-up decision comes at an inopportune moment, right after the input rate drops to 1K rec/s.

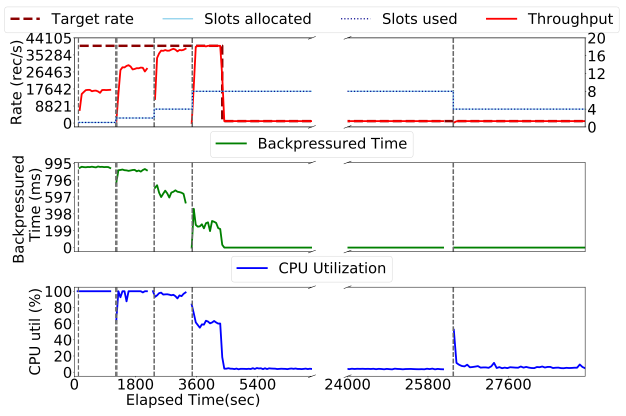
Take-away #4: *Policy decisions may be based on stale metrics.* This experiment shows that Kinesis makes an incorrect scale-up decision after the rate drops. We conjecture this is due to the model using metrics collected during the first phase of the experiment, when the rate was high. Even though one worker can handle the low rate, the job remains over-provisioned until the end of the experiment.

Q8 consists of a windowed join operator between streams of auctions and persons. Throughput and backpressure in Figure 1d correspond to the high-rate auctions source. All other metrics characterize the entire query. The scaling behavior we observe is similar to that of the previous experiment. When starting from an under-provisioned state, Kinesis doubles the allocated resources every 15min, yet without reaching the target rate by the end of the first phase. Interestingly, this experiment shows that increasing the resources does not always have a positive effect on performance. In fact, after Kinesis scales from 4 to 8 slots, backpressure remains high at 80% and throughput stays at 8K rec/s, far below the target. At the same time, CPU utilization has dropped to 26%.

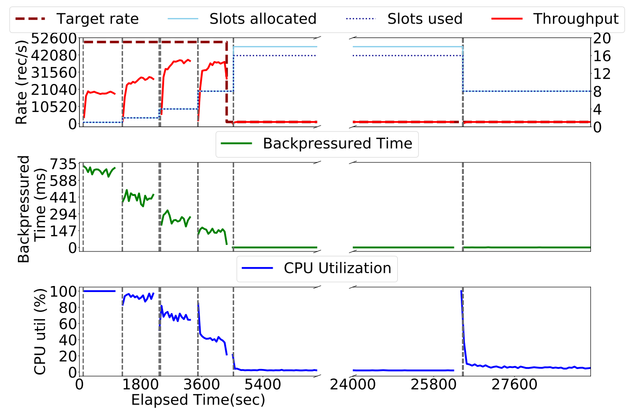
Take-away #5: *Kinesis is not always capable of accurately identifying bottlenecks.* Relying on CPU utilization to inform the scaling



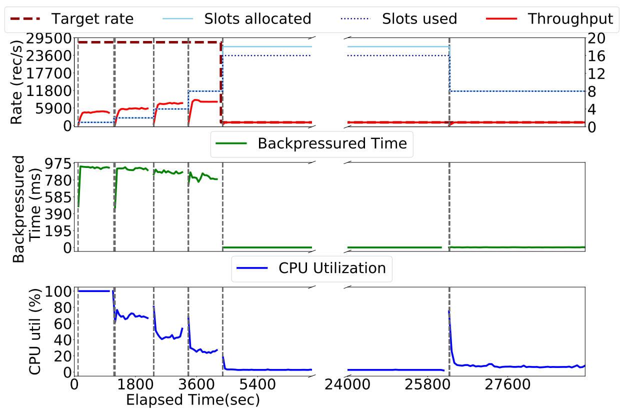
(a) Q1



(b) Q5



(c) Q3



(d) Q8

Figure 1: Scaling behavior of Amazon Kinesis on Nexmark queries

policy can be misleading and increasing the allocated resources does not guarantee improved performance.

4.2 Auto-scaling in Google Dataflow

Figure 2 shows the auto-scaling behavior of Google Dataflow for Nexmark queries. We plot metrics for the first 2 hours of each experiment as Dataflow makes scaling decisions faster than Kinesis. It is also worth highlighting that the Apache Beam Nexmark implementation generates events in an *open-loop* fashion. As a result, backlog can accumulate and observed throughput may exceed the target rate when scaling up. Vertical dashed lines indicate the time of scaling decisions, while the workers subplot shows the time when these decisions take effect.

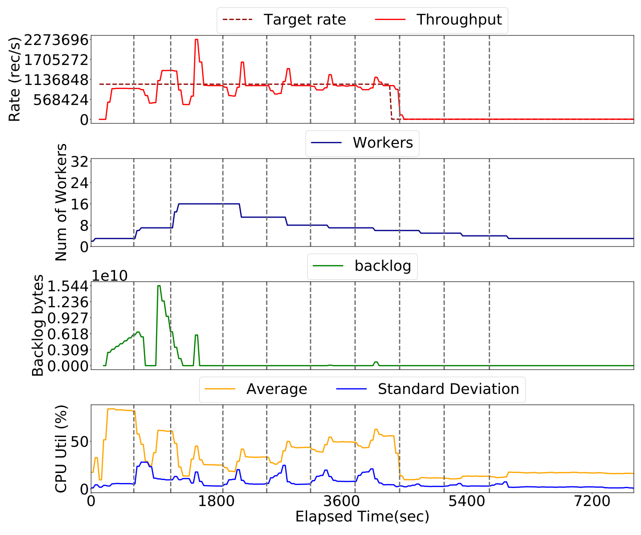
Q1 results are shown in Figure 2a. Dataflow initially provisions 3 workers. As the rate is high and this configuration cannot keep up, backlog is quickly accumulated and the average CPU utilization reaches 75%. The service makes 2 scaling decisions, roughly one every $10min$, reaching a peak at 16 workers. After the backlog is consumed, CPU utilization drops below 50%. From this point on, Dataflow makes one scale-down decision every $10min$ until it converges to a 6 workers for the high rate. At $70min$, we lower the input rate and Dataflow takes 3 steps to gradually reduce the number of workers to 3. This experiment verifies that Dataflow scaling decisions rely on the backlog size and CPU utilization.

Take-away #6: *Dataflow makes scaling decisions every* $10min$.

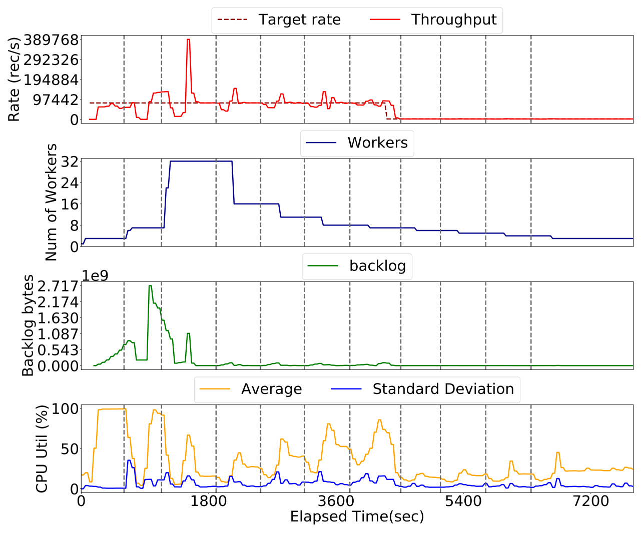
Q3 and Q5 results are shown in Figures 2c and 2b, respectively. In both cases, Dataflow makes 2 scale-up decisions in the first phase and peaks at 32 workers, which is the upper limit set in this experiment. After consuming the accumulated backlog, it gradually reduces the number of workers (every $10min$), eventually converging to 8 workers for $Q3$ and 7 workers for $Q5$. The scale down behavior in the second phase is similar to the previous experiment.

Q8 results are shown in Figure 2d. Here we see Dataflow scaling up from 3 workers directly to 32 as the backlog accumulates more quickly than in previous experiments. The job runs with this configuration for $20min$ before it scales down to 16 workers. In the second phase, the policy scales gradually to 3 workers in 7 steps ($11 \longrightarrow 8 \longrightarrow 7 \longrightarrow 6 \longrightarrow 5 \longrightarrow 4 \longrightarrow 3$).

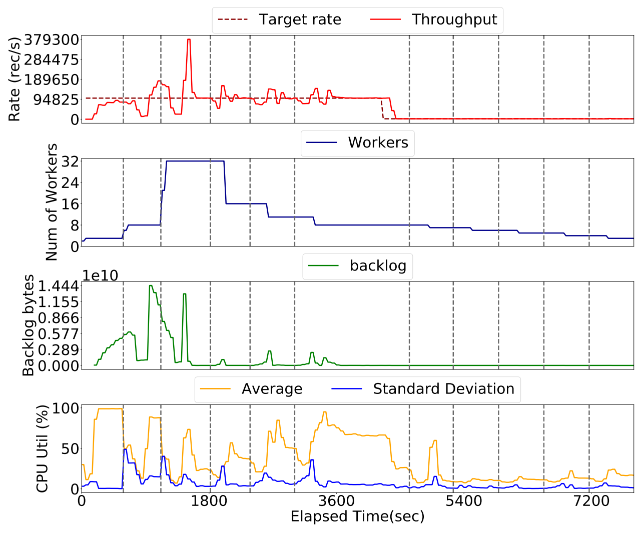
Take-away #7: *The scale-up and down policies are asymmetric.* Dataflow increases resources aggressively, adding multiple workers



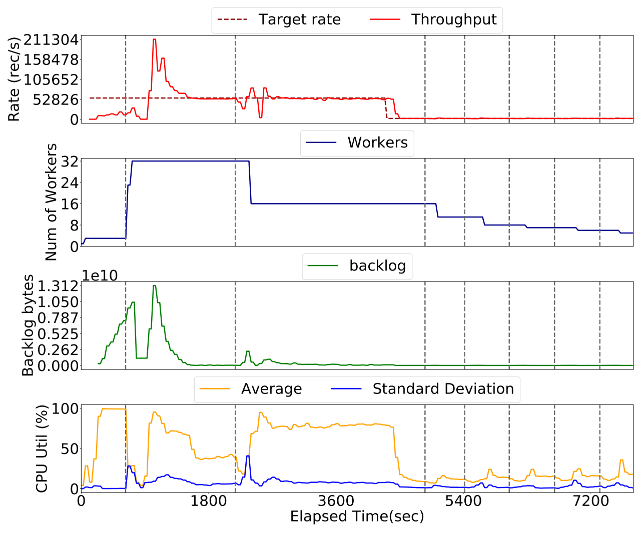
(a) Q1



(b) Q5



(c) Q3



(d) Q8

Figure 2: Scaling behavior of Google Dataflow on Nexmark queries

per step (from 3 to 6 to 16). On the other hand, it is conservative when scaling down, removing a single worker per step.

Take-away #8: *Dataflow remembers previous scale-up decisions.* Though not shown in the figures, the second time we increase the rate, the service directly provisions the peak number of workers. This action indicates that the policy learns how many workers are needed to clear the backlog.

4.3 Quantifying the Non-Expert Tax

Table 3 shows the actual cost, ideal cost, and incurred *Non-Expert Tax* for the experiments of Figures 1 and 2, alongside estimated costs and tax per month for two common workload patterns [18]. The *periodic* pattern simulates cyclic workloads and alternates between high and low rate every 12 hours. The *spike* workload simulates abrupt and short rate variations by increasing the rate to its high value for 1 hour every day.

To estimate the monthly costs for these two workloads, we construct rule-base policy models, according to the insights we gained from the experiments in Sections 4.1 and 4.2. In particular, we make the following assumptions. The Kinesis scale-up period is 15*min* and the scale-down period is 6*h*. The policy always varies resources by a factor of 2. The Dataflow policy interval is 10*min*, regardless if scaling up or down. When scaling up for the first time, Dataflow follows the steps observed in Figure 2 but any future scale-up decisions directly bring the configuration to the peak

		Experiment workload (per query)			Periodic workload for a month			Spike workload for a month		
		Actual Cost ($)	Ideal Cost ($)	Non-Expert Tax (%)	Actual Cost ($)	Ideal Cost ($)	Non-Expert Tax (%)	Actual Cost ($)	Ideal Cost ($)	Non-Expert Tax (%)
Kinesis	Q1	13.98	2.17	544.23	1252.7	289.8	332.26	419.55	173.97	141.16
	Q3	13.98	2.87	387.1	1252.7	542.52	130.90	419.55	195.03	115.12
	Q5	7.9	2.64	199.24	666.77	458.28	45.49	419.55	188.01	123.15
	Q8	13.98	2.87	387.1	1252.7	542.52	130.90	419.55	195.03	115.12
Dataflow	Q1	2.68	1.06	152.83	275.03	190.08	44.69	206.3	73.92	179.09
	Q3	6.68	2.21	202.26	642.8	428.4	50.05	425.88	147.9	187.95
	Q5	6.16	2.04	201.96	586.9	367.2	59.83	425.85	142.8	198.21
	Q8	8.04	3.91	105.63	1209.55	1040.4	16.26	464.24	198.9	133.40

These results do not provide a reliable way to compare Kinesis with Dataflow in terms of cost. The services were evaluated using different input rates and instance types.

Table 3: Actual cost, ideal cost, and incurred Non-Expert Tax for the Nexmark queries in three workload scenarios.

number of workers, as the policy remembers previous actions. Scaling down is gradual and always follows the pattern observed in the experiments, that is, Dataflow does not remember previous scale-down actions. Finally, even though we sometimes observed Dataflow making scale-up decisions during low rates, we ignore such events as they are infrequent and probably triggered by noisy metrics. We compute the costs using these rules and the resource costs provided by Amazon [3] and Google [16].

Example. Consider the cost calculation of $Q1$ on Kinesis for the periodic workload. We have empirically verified that ideal configurations provision 1 and 4 workers, for the low and high rates, respectively. In reality, Kinesis also starts with 1 worker but when the rate changes to high, it makes a scale-up decisions, doubling the amount of resources every $15min$, until it reaches 16 workers after 1 hour. It then remains at this configuration for 11 hours. When the rate drops, Kinesis takes 6 hours to scale down to 8 workers and remains at this configuration until the end of the low rate phase. This pattern repeats every day for a month.

Results. The Kinesis *Non-Expert Tax* is high for all queries and workloads, ranging from 45% to 544%. It is especially surprising that the highest tax is incurred for $Q1$, which is the simplest among queries. $Q1$ performs a map transformation and does not include any stateful, window, or two-input operators that may be hard to model. Further, we see that the tax is higher for the periodic workload. This is due to Kinesis taking $6h$ to make a scale-down decision when the rate drops. Since the low rate period is 12 hours for this workload, two decisions are not enough for Kinesis to reach the ideal configuration for any of the queries. For example, in the case of $Q1$, the policy scales from 16 to 8 workers after $6h$ and then to 4 workers, $6h$ later, while the ideal configuration is 1 worker.

Google Dataflow incurs lower *Non-Expert Tax*, though still substantial in all cases. This is due to Dataflow's short reaction time, as it makes reconfiguration decisions every $10min$. As a result, some of the over-provisioning cost is amortized in the monthly workloads. Yet, Dataflow never provisions fewer than 3 workers, even if a single worker is sufficient to keep up with the low input rate.

5 RELATED WORK

Automatic scaling and resource configuration is a central problem in data stream processing research [12]. Prior work has proposed various predictive [17, 19] and heuristic policies [11, 26] to decide when and how much to scale when input rate changes. In this study, we found that Amazon Kinesis, Azure Stream Analytics, and Alibaba Realtime Compute rely on heuristic policies, while Google Cloud Dataflow combines both heuristic policies and a predictive model. Further, there exist plenty of empirical studies on Cloud infrastructure and managed services, evaluating network performance [21], the efficiency of serverless platforms [27], job scheduling [24], and spot instance pricing [13]. Closest to our work are studies concerned with the cost of Cloud databases [22] and those evaluating resource management and auto-scaling of internal Cloud systems, like Autopilot [9, 23]. To the best of our knowledge, this paper is the first to assess the cost-benefit trade-off of auto-scaling in managed data stream services.

6 DISCUSSION AND CONCLUSION

In this paper, we presented an empirical evaluation study of auto-scaling policies in two popular data stream analytics services. We showed that existing policies scale out aggressively to meet performance demands but are conservative when scaling down, often incurring considerable monetary cost for underutilized resources.

The results of our study show there is large room for improving the cost and resource efficiency of Cloud-hosted streaming analytics services. Current auto-scaling features hastily scale-up to meet performance targets even if bottlenecks are not caused by resource scarcity. For example, we observed how adding more workers to $Q8$ on Kinesis does not eliminate backpressure and how Dataflow suffers from imbalance and low CPU utilization at configurations with a large number of workers.

Our experience conducting this study further reveals that these services are not easy to use, even by experts. In many cases, user input is required to achieve good performance or avoid failures. For example, some of our jobs ran out of memory when not selecting an appropriate instance type. In other cases, user settings might conflict with policy decisions and silently cause unexpected behavior. Specifically, setting parameters such as `Parallelism` or `ParallelisPerKPU` in Kinesis (when auto-scaling is enabled) leads to a perpetual cycle of unsuccessful scaling decisions.

REFERENCES

[1] Alibaba. 2022. *Configure Autopilot for fully managed Flink.* https://www.alibabacloud.com/help/en/doc-detail/173651.htm Last access: March 2022.
[2] Amazon. 2022. *Amazon Kinesis.* https://aws.amazon.com/kinesis/ Last access: March 2022.

[3] Amazon. 2022. *Amazon Kinesis Data Analytics pricing*. https://aws.amazon.com/kinesis/data-analytics/pricing/ Last access: March 2022.

[4] Amazon. 2022. *AWS Kinesis Data Analytics Automatic Scaling*. https://docs.aws.amazon.com/kinesisanalytics/latest/java/how-scaling.html#how-scaling-auto Last access: March 2022.

[5] Apache. 2022. *Apache Beam*. https://beam.apache.org/ Last access: March 2022.

[6] Apache. 2022. *Apache Flink*. https://flink.apache.org/ Last access: March 2022.

[7] Apache Beam. 2022. *Nexmark Benchmark Suite*. https://beam.apache.org/documentation/sdks/java/testing/nexmark/ Last access: March 2022.

[8] Paris Carbone, Marios Fragkoulis, Vasiliki Kalavri, and Asterios Katsifodimos. 2020. Beyond Analytics: The Evolution of Stream Processing Systems. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. 2651–2658.

[9] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. 153–167. https://doi.org/10.1145/3132747.3132772

[10] Miyuru Dayarathna and Srinath Perera. 2018. Recent Advancements in Event Processing. *ACM Comput. Surv.* 51, 2, Article 33 (2018). https://doi.org/10.1145/3170432

[11] 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. https://doi.org/10.14778/3137765.3137786

[12] 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

[13] Gareth George, Rich Wolski, Chandra Krintz, and John Brevik. 2019. Analyzing AWS Spot Instance Pricing. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*. 222–228. https://doi.org/10.1109/IC2E.2019.00036

[14] Google. 2022. *Autotuning features - Google Cloud Dataflow*. https://cloud.google.com/dataflow/docs/guides/deploying-a-pipeline#autotuning-features Last access: March 2022.

[15] Google. 2022. *Google Cloud Dataflow*. https://cloud.google.com/dataflow Last access: March 2022.

[16] Google. 2022. *SKU Groups - Dataflow - Google Cloud*. https://cloud.google.com/skus/sku-groups/dataflow Last access: March 2022.

[17] 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*.

[18] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. 2018. Query-Based Workload Forecasting for Self-Driving Database Management Systems. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. 631–645.

[19] Yuan 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.

[20] Microsoft. 2022. *Autoscale Stream Analytics jobs using Azure Automation*. https://docs.microsoft.com/en-us/azure/stream-analytics/stream-analytics-autoscale Last access: March 2022.

[21] Ricky K. P. Mok, Hongyu Zou, Rui Yang, Tom Koch, Ethan Katz-Bassett, and K C Claffy. 2021. Measuring the Network Performance of Google Cloud Platform. In *Proceedings of the 21st ACM Internet Measurement Conference*. Association for Computing Machinery, 54–61. https://doi.org/10.1145/3487552.3487862

[22] Andy Pavlo. 2022. *You are Overpaying Jeff Bezos for Your Databases (and the things he does with that extra money)*. https://ottertune.com/blog/overpaying-jeff-bezos-for-aws-databases/ Last access: March 2022.

[23] Krzysztof Rzadca, Paweł Findeisen, and Jacek Świderski et al. 2020. Autopilot: Workload Autoscaling at Google Scale. In *Proceedings of the Fifteenth European Conference on Computer Systems*.

[24] Huangshi Tian, Yunchuan Zheng, and Wei Wang. 2019. Characterizing and Synthesizing Task Dependencies of Data-Parallel Jobs in Alibaba Cloud. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. 139–151. https://doi.org/10.1145/3357223.3362710

[25] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2002. *NEXMark—A Benchmark for Queries over Data Streams*. Technical Report. OGI School of Science & Engineering at OHSU.

[26] Le Xu, Boyang Peng, and Indranil Gupta. 2016. Stela: Enabling Stream Processing Systems to Scale-in and Scale-out On-demand. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*. 22–31. https://doi.org/10.1109/IC2E.2016.38

[27] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing Serverless Platforms with Serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. 30–44.