# Exploring Stream Processing Autoscaling in Kubernetes Deployments

**Ioan-Cristian Olariu**
Practical Work in AI (Master)
Johannes Kepler Universität Linz
K12219769@students.jku.at

## Abstract

This report explores practical aspects of runtime adaptation of data stream processing (DSP) applications to variable workloads through horizontal autoscaling of processing resources. The experimental setup involves technologies and tools that are widely used across industries: cloud-native microservice architectures orchestrated through Kubernetes, Kafka Streams as a lightweight and high-performance stream processing engine, and Kubernetes' own Horizontal Pod Autoscaler (HPA). For the full Kubernetes deployment of the stream processing system and to simulate workloads typical of many real-world stream processing applications, I relied on the ShuffleBench benchmark. The report first documents the customization steps needed to adapt the benchmark to run on a local machine Minikube cluster and to perform scaling experiments with external metrics for HPA. Subsequently, we will look at experimental results and analyze the behavior observed from collecting key metrics. Additionally, to put intended further developments in context, we will briefly review AI techniques used for improving stream processing autoscaling in recent literature.

## 1 Introduction

Data stream processing (DSP) is a paradigm within the larger field of big data, which is concerned with the timely analysis of continuous, fast, and conceptually infinite information flows. DSP applications are organized as directed acyclical graphs within which vertices can be data sources, operators, and final consumers. Streams are represented as graph edges indicating the data flow between the vertices [1]. The operators apply specific transformations (e.g., filtering). By combining multiple operators, DSP applications can solve complex queries over unbounded data streams, providing results continuously. DSP applications often have strict Quality of Service (QoS) requirements, such as response time constraints, which must be maintained at runtime, despite the high volume and variability of their workloads. To address operator overloading, a commonly used optimization is to employ data parallelism. This involves running multiple parallel replicas of the same operator, allowing the incoming data flow to be distributed among different replicas that perform computations simultaneously. The parallelism of DSP applications should be adjusted dynamically at run-time to match the workload and avoid resource waste. Specifically, the number of operator replicas should be scaled up when the load increases and scaled down when the load decreases. In practice, these systems are deployed in cloud computing environments where resources can be provisioned elastically. Some of the most widely used DSP frameworks — like Apache Spark[1], Kafka Streams[2] [16], and Apache Flink[3] in reactive mode[4] — are capable of automatically redistributing and balancing the work as computing instances are added or removed. An automatic scaling controller used together with such a framework needs only to act by adjusting the replication count of identical parallel computation instances (e.g., virtual machines or Kubernetes pods) and remain agnostic of any application or framework-specific optimizations of the data flows.

[1] https://spark.apache.org/
[2] https://kafka.apache.org/documentation/streams/
[3] https://flink.apache.org/
[4] https://flink.apache.org/2021/05/06/scaling-flink-automatically-with-reactive-mode/

The purpose of the Practical Work was to understand and document firsthand some of the challenges of DSP runtime adaptation through horizontal autoscaling, using tools and practices that the industries have widely adopted. I have used Kubernetes for distributed deployments with its included Horizontal Pod Autoscaler, together with the Kafka Streams DSP framework. To simulate realistic workloads and deploy a DSP application conforming to the best practices of cloud-native microservices system architecture, I was able to benefit from employing the open-source implementation of the ShuffleBench benchmark from Henning et al. [7].

The resources necessary for replication of the results are open-source and available in a public repository[5].

## 2 Related work

Cardellini et al. [3] provides a comprehensive survey of runtime adaptation techniques for data stream processing systems and applications. In the proposed hierarchy of diverse adaptation mechanisms, the aspects that this report is focused on pertain mostly to *infrastructure scaling* as a type of *infrastructure adaptation*, but also partly to *horizontal operator scaling* as a type of *deployment adaptation*.

Siachamis et al. [14] evaluates HPA as a stream processing autoscaler, while comparing its performance to state-of-the-art autoscalers DS2[6] and Dhalion[7] — specifically to adjust operator-level parallelism in Apache Flink streaming applications. The authors created a custom version of HPA to monitor the average CPU utilization of actual operators within pods but exact implementation details are not made available. According to the authors, the custom HPA autoscaler was the best performer in the comparison in terms of latency, but it recommended more parallel instances than DS2. The influential DS2 automatic scaling controller, described in Kalavri et al. [9], relies on lightweight instrumentation to measure "useful" processing time and attempt to reactively determine the optimum level of parallelism for each operator in the data flow.

The findings in this report are meant to inform future work, which will investigate applying recent progress in Artificial Intelligence to the problem of adaptive autoscaling for DSP systems. The rest of this section briefly describes a selection of works from the scientific literature, related to the intended direction of future research.

In Imai et al. [8], the authors used linear regression with online updating to estimate a variance-of-performance model, and an ARMA model to predict load for proactive scaling. The statistical models — trained on real measurements — were tested in simulations using real-world workloads with promising results both in terms of QoS satisfaction and estimated cost.

Lombardi et al. [10] presented an elastic scaling approach integrated into Apache Storm, which relies on several small Artificial Neural Network regression models whose predictions are used as input for threshold-based scaling policies.

Several papers from researchers with the University of Rome propose Reinforcement Learning (RL) approaches to tackle DSP autoscaling, with reported good results in numerical simulations. In Cardellini et al. [1], the authors describe model-based RL techniques to self-configure the number of parallel instances at the level of individual operators. Cardellini et al. [2] compare threshold-based to model-free and model-based RL approaches on a benchmark workload. Russo et al. [12] combine model-based planning and model-free learning to improve adjusting operator parallelism in response to workload variations. In Russo Russo et al. [13], the authors propose a hybrid RL approach based on Deep Q-Learning with post-decision state and Bayesian Optimization for hierarchical autoscaling policies on heterogeneous resources.

Gontarska et al. [5] evaluate seven Time Series Forecasting (TSF) techniques to assess their accuracy on load prediction when applied to nine data sets from different DSP domains. The authors found that deep learning methods generally provide better prediction performance than classical TSF methods. In counterpoint, the classical methods are faster to train, require fewer resources to run, and are easier to configure optimally for specific use cases.

Pfister et al. [11] describe using TSF to anticipate future workloads to enable proactive scaling decisions, as a key part of the architecture of their self-adaptive autoscaling controller. The predictive accuracy of the ARIMA model is being continuously monitored, and consistently poor predictions trigger retraining in the background. The authors experimentally compared the performance of the proposed controller against an HPA-based approach and reported achieving comparable latency while significantly reducing resource usage.

[5] `https://github.com/yicristi/dsp-scaling-k8s`

[6] `https://github.com/strymon-system/ds2`

[7] `https://www.microsoft.com/en-us/research/project/dhalion/`

In Geldenhuys et al. [4] the authors propose a method that relies on TSF to predict future workloads, along with using Multi-Objective Bayesian Optimization to model runtime behaviors. The purpose is to optimize for efficiency by dynamically adjusting multiple configuration parameters — including horizontal scaling.

# 3 Experimental setup

The replication of the full ShuffleBench experiments in Henning et al. [7] involves provisioning through Amazon Elastic Kubernetes Service (EKS) several nodes in the AWS cloud, together with their storage and other cloud resources requirements. The authors acknowledge that this will incur a high cost when sustained for longer periods of time. Instead, by relaxing the requirements and reducing the workload, it is possible to have a full deployment on a single machine. This is adequate for evaluating the behavior of the tools in quick experimental cycles, considering that statistically valid benchmark results are not being sought.

I have used an Ubuntu 22 Linux workstation equipped with an Intel i7 CPU (8 performance cores with hyper-threading and 8 efficiency cores for a total of 24 virtual cores), 32 GB of DDR5 RAM memory and 2TB of SSD storage.

Table 1 summarizes the key differences between the full setup and the local setup used here.

| | **Full setup** | **Local setup** |
|---|---|---|
| **Deployment resources** | AWS cluster of 10 nodes: 4 infra, 3 kafka, 3 sut (system under test) | 1 node (label: infra) cluster – minikube on single PC with 24 (virtual) cores and 32GB RAM |
| **Messages in** | 1 M/s | 320 K/s |
| **Grouping rules** | 1M | 10 K |
| **Input topic partitions** | 100 | 20 |
| **Instances** | 9 | Start with 1 and auto-scale |

Table 1: Summary of local setup compared with full "baseline ad-hoc throughput" benchmark setup.

The following sections describe the installation and configuration of Kubernetes on the local machine and the deployment of ShuffleBench for replicating my experiments. The architecture of the complete deployment is illustrated in Figure 1.

## 3.1 Customized manifest files and setup scripts

To help replicate the experiments, all the necessary manifest files and setup scripts are provided separately in the archive file `sb_custom.zip`[8]. The archive should be unpacked in the home directory of the current user, resulting in the creation of a directory named `sb_custom`. The listings of the various files within this directory are also included as an appendix to the report.

## 3.2 Prepare Kuberenetes requirments

Minikube[9] is a distribution of local Kubernetes, focusing on facilitating learning and developing for Kubernetes, but not intended for use in production. It can use any of several container or virtual machine managers to create full Kubernetes clusters on a single Linux host computer. We will pair it with the Docker driver, therefore cluster nodes will be lightweight Docker containers.

For detailed command line steps for installation see the `minikube-docker-setup.sh` script in listing 1. The installation consists of:

1. Installing Docker.
2. Installing the `kubectl`[10] command line client for the Kubernetes API.
3. Installing minikube.
4. Installing Helm[11], a package manager for Kubernetes.

[8] `https://github.com/yicristi/dsp-scaling-k8s/raw/main/sb_custom.zip`
[9] `https://minikube.sigs.k8s.io/docs/start/`
[10] Alternatively the `minikube kubectl` wrapper is also available.
[11] `https://helm.sh/`

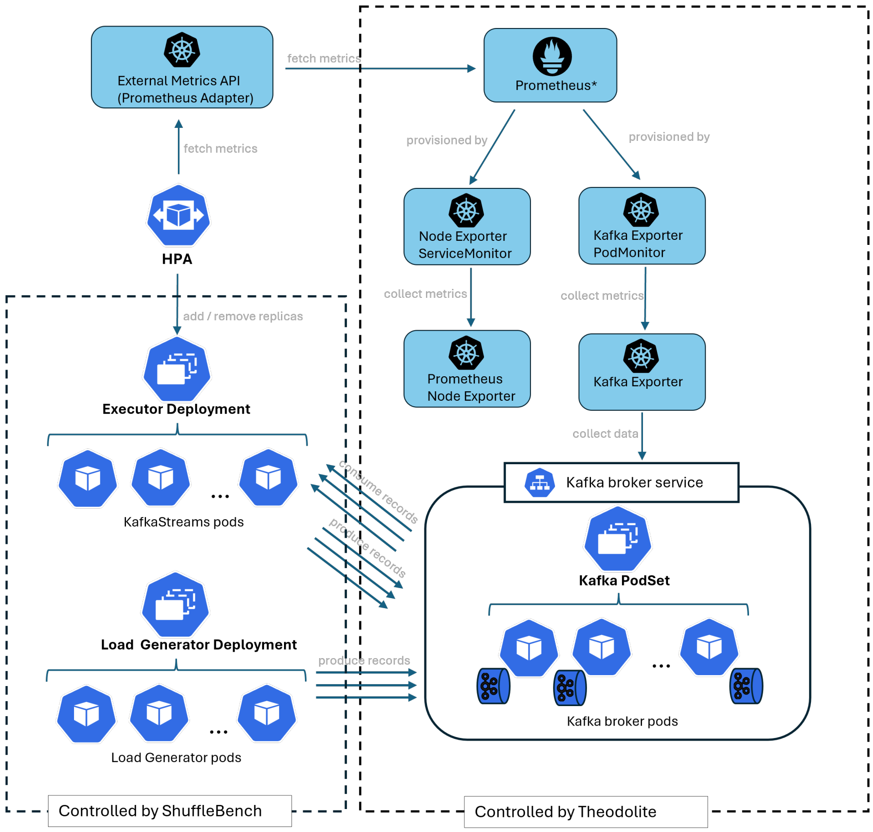


Figure 1: Overview of deployment (simplified).

5. Add some convenience settings: modify inotify limits, enable bash prompt autocompletion for most new commands.

Due to the large number of file descriptors that will be used across all containers, a very important preparation step is to change the host kernel limits that affect the `inotify` system API for monitoring file system events. The default limits in Ubuntu 22 are too low for this use case, which can cause Kubernetes pods to fail in non-obvious ways. The values 1024 for `fs.inotify.max_user_instances` and 1048576 for `fs.inotify.max_user_watches` were found to be large enough to avoid any such problems.

### 3.3 Prepare ShuffleBench and Theodolite

We will use ShuffleBench — which in turn relies on the Theodolite [6] benchmarking framework —- to create the full streaming application deployment as microservices and inject the workload.

Once the ShuffleBench repository has been cloned locally:

```
git clone https://github.com/dynatrace-research/ShuffleBench.git
```

we will assume the `kubernetes` directory inside the ShuffleBench clone to be the working directory when running all the experiments:

```
cd ShuffleBench/kubernetes
```

Inside the working directory, we will also clone the Theodoloite repository which is a dependency for ShuffleBench.

```
git clone https://github.com/cau-se/theodolite.git
```

All the customization to ShuffleBench to run in the reduced local setup is done in separate files in the sb_custom directory, some of which replace equivalent manifest files existing in the ShuffleBench repository and replication package.

### 3.4 Create cluster and deploy benchmark resources

To create the cluster and fully deploy all the benchmark resources to a state ready to run the experiments, it is sufficient to execute the shell script in listing 4 and wait until its completion (it may take around 10 minutes):

```
source ~/sb_custom/setup.sh
```

This section will further present in more detail the commands included in this script, and discuss the way ShuffleBench configurations and procedures were altered.

In the following, the variable ${cluster_name} refers to the minikube profile name (cluster name), and the variable ${base_path} refers to the path where sb_custom.zip has been unpacked. They could be set, for example, to:

```
cluster_name=sbcl
base_path=~/sb_custom
```

#### 3.4.1 Creating the cluster

Cluster creation commands in setup.sh

```
1 minikube start --nodes 1 -p ${cluster_name} --addons metrics-server↩
     storage-provisioner default-storageclass --cni calico --cpus ↩
   no-limit --memory no-limit --wait=all
2 kubectl wait --all --timeout=10m --for=condition=Ready pods
3 minikube addons -p ${cluster_name} enable dashboard
4 kubectl label nodes ${cluster_name} type=infra
```

In line 1 of the above snippet, we create a Kubernetes cluster formed of a single node. Using the default Docker driver with minikube, the node itself is a container. If a different driver was selected (e.g., VirtualBox), the cluster node could have been a virtual machine instance. Containers, as opposed to virtual machines, share the host Linux Kernel resources within separate namespaces. For this testing scenario, it was deemed unnecessary to suffer the overhead added by using virtual machines. Note that the Docker runtime is also used for all the containers managed within pods.

The minikube distribution conveniently includes many of the most commonly used Kubernetes applications and services as addons, and they only need to be explicitly enabled. Here we enable the Metrics Server (required for the HPA metrics and Kubernetes Dashboard), storage-related addons, and the Kubernetes Dashboard. Note that the latter needs to be enabled with some delay after cluster creation (line 3).

In line 4, we apply the label "infra" to the only node. In ShuffleBench and Theodolite, pods are assigned to different categories of nodes in the cluster, using label selectors to match the label assigned to each node. The possible labels are "infra", "kafka", and "sut". Since we only have one node, we will make sure that all assignments are made to the "infra" label.

#### 3.4.2 Prepare Kafka storage

In the full AWS deployment of ShuffleBench, the Kafka pods are provisioned with 1TB each of Elastic Bean Stalk SSD storage. For the local setup, we will instead provision the same pods with storage mapped to local folders on the host node, for up to 80GB each. Within minikube, we only need to create a Custom Resource of kind StorageClass matching the "kafka" name that it's already referenced in Persistent Volume Claims that ShuffleBench associates with the respective pods. We make sure to specify the provisioner to be the minikube built-in Host Path mechanism, as can be seen in listing 3. Note that this storage is not guaranteed to be persistent between node restarts (since the host paths are by default created within /tmp), but this is deemed to not be a problem for our short-lived experiments.

Additionally, we need to make sure that the storage-provisioner service account exists with a cluster-admin role, which is shown in listing 2.

We create the corresponding resources by applying their manifest files, as shown in the snippet below.

Kafka storage configurations in `setup.sh`

```
kubectl apply -f ${base_path}/rbac-storage-provisioner.yaml
kubectl apply -f ${base_path}/kafka-storage-class.yaml
```

### 3.4.3 Deploy Theodolite

Commands to deploy the Theodolite operator in `setup.sh`

```
helm dependencies update theodolite/helm
helm install theodolite theodolite/helm -f theodolite/helm/preconfigs/extended-metrics.yaml -f ${base_path}/values-theodolite-local.yaml
kubectl wait --all --timeout=15m --for=condition=Ready pods
```

To be able to run ShuffleBench, we need to first install the Theodolite operator on the running cluster, which adds several Custom Resource Definitions for benchmarking purposes (e.g., `benchmark` and `execution`) as well as deploying dependencies such as Prometheus[12], Grafana[13], and Strimzi[14]. The installation is handled by the `helm` tool, as instructed by the corresponding Helm chart. To customize the installation, we provide the `values-theodolite-local.yaml` file (listing 6). The following are noteworthy custom settings:

- Make sure the Kafka pods are associated with the node labelled "infra": `nodeSelector = infra`.
- Set the size of the Kafka storage claim to 80GB: `strimzi.kafka.storage.size = 80Gi`.
- Enable persistent storage of execution results: `operator.resultsVolume.persistent.enabled = true`.

The Theodolite presets for Grafana already include a comprehensive dashboard for monitoring key streaming performance metrics. The existing dashboard was modified to include additional plots: "CPU per Pod", "HPA CPU Metric", "Throughput 10s", "Detailed Input Throughput", and "HPA Input Lag Metric". The following replaces the Config Map that stores the presets and forces a restart of Grafana to reload its new configuration.

Commands to update the Grafana dashboard in `setup.sh`

```
kubectl replace -f ${base_path}/dashboard-config-map.yaml
kubectl scale deployment theodolite-grafana --replicas=0
kubectl scale deployment theodolite-grafana --replicas=1
```

### 3.4.4 Deploy Prometheus Adapter

Commands to deploy the Prometheus Adapter operator in `setup.sh`

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm repo update
helm install prom-adapt prometheus-community/prometheus-adapter -f ${base_path}/values-prom-adapt.yaml
```

In this step, by passing the file `values-prom-adapt.yaml` (listing 8) to `helm install`, we provide external metrics that HPA may use for scaling decisions.

The `kafka_input_lag` metric represents the lag (the difference between the last message produced by the producer and the offset committed by the consumer group) in the Kafka consumer group, averaged over 20 seconds. We define the query via a templated PromQL[15] expression. In the actual requests made through the metrics API, the `LabelMatchers` template variable will be replaced with a concrete label-matching expression.

[12] `https://prometheus.io/`
[13] `https://grafana.com/grafana/`
[14] `https://strimzi.io/`
[15] `https://prometheus.io/docs/prometheus/latest/querying/basics/`

kafka_input_lag metric definition in values-prom-adapt.yaml

```
- metricsQuery: sum(avg_over_time(kafka_consumergroup_lag{ << .LabelMatchers >> }[20s])) by (topic,consumergroup)
  name:
    as: kafka_input_lag
```

The consumer_cpu_utilization_30s represents the instant CPU utilization percent value (within the last 30 seconds) for each Kafka Streams consumer, while also omitting "sidecar" containers in those pods.

consumer_cpu_utilization_30s metric definition in values-prom-adapt.yaml

```
- metricsQuery: avg(avg by (pod) (irate(container_cpu_usage_seconds_total{pod=~"shuffle-kstreams.*"}[30s])) * 100) without (pod)
  name:
    as: consumer_cpu_utilization_30s
```

### 3.4.5 Creating Horizontal Pod Autoscaler

Commands to prepare and deploy HPA in setup.sh

```
kubectl apply -f ${base_path}/hpa-authorization.yaml
kubectl apply -f ${base_path}/hpa-custom.yaml
```

We need to give the horizontal-pod-autoscaler service account access to the external metrics server resources, which is being done in hpa-authorization.yaml (listing 14) by creating a new Cluster Role and binding it to the service account.

The autoscaler is then created from the manifest hpa-custom.yaml (listing 15). If we wish to change the HPA behavior we can change this manifest and reapply it:

```
kubectl replace -f ~/sb_custom/hpa-custom.yaml
```

It's worth discussing the HPA specification in more detail. We indicate that the target for scaling is the deployment matching the name shuffle-kstreams and that we require at least one replica but never exceed a maximum of 10 replicas.

Target and scaling limits specifications in hpa-custom.yaml

```
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: shuffle-kstreams
  minReplicas: 1
  maxReplicas: 10
```

On the scaling-up behavior, we limit adding new pods to not more than two per minute. In this way, we try to avoid overreacting to short-term spikes in the metric. There is no explicit stabilization window for the scaling-up action by default, which fits this use case since we prefer to reduce the input lag as soon as possible and not let it accumulate. We want to react quickly to the worse case for performance when we face a massive and steady increase of the workload. The trade-off is that we also react quickly to oscillations of the metrics.

Scale up specifications in hpa-custom.yaml

```
    scaleUp:
      policies:
        - type: Pods
          periodSeconds: 60
          value: 2
```

For scaling down we prefer to have a stabilization window, here 60 seconds. With this, the larger replica count calculated by the scaling algorithm in the past 60 seconds is the one applied. This reduces flapping pods. We also impose that at most 3 pods can be killed over any 15-seconds window.

Scale down specifications in `hpa-custom.yaml`

```
    scaleDown:
      stabilizationWindowSeconds: 60
      policies:
        - type: Pods
          periodSeconds: 15
          value: 3
```

In the final configuration, discussed here, we will define two external metrics. At each sampling interval, the HPA algorithm calculates a replica count for each individual metric and selects the highest count for the final decision.

The first metric we want HPA to check is the Kafka input lag, an external metric made available by the Prometheus Adapter service. The `selector` property defines label values to filter the corresponding Prometheus time series, namely to consider only the lag measured for the `input` Kafka topic and the `shufflebench-kstreams` consumer group. The target value of the lag metric is set to 3 million records.

Kafka lag external metric specifications in `hpa-custom.yaml`

```
        metric:
          name: "kafka_input_lag"
          selector:
            matchLabels:
              topic: input
              consumergroup: shufflebench-kstreams
        target:
          type: Value
          value: 3000k
```

The second metric to be monitored is CPU utilization averaged over all the executor (application) pods. Here the desired (target) utilization is 80%.

CPU utilization external metric specifications in `hpa-custom.yaml`

```
        metric:
          name: "consumer_cpu_utilization_30s"
        target:
          type: Value
          value: 80
```

### 3.4.6 Prepare ShuffleBench and create benchmark

The ShuffleBench benchmark specification in `theodolite-benchmark-kstreams-simple.yaml` (listing 12) references sets of manifest files that should be loaded into Config Maps for specific resources to be created at execution time — for System under Test and the Load Generator, in the Theodolite architecture. The specification also defines service level objective (SLO) metrics — again based on Prometheus queries — to be recorded during the benchmark execution.

Commands to deploy benchmark resources in `setup.sh`

```
kubectl create configmap shufflebench-resources-load-generator --from-file ./shuffle-load-generator/
kubectl create configmap shufflebench-resources-latency-exporter --from-file ./shuffle-latency-exporter/
kubectl create configmap shufflebench-resources-kstreams --from-file ${base_path}/shuffle-kstreams/
kubectl apply -f ${base_path}/theodolite-benchmark-kstreams.yaml
```

The above manifests are similar to the original files provided in the ShuffleBench replication package, with only a few changes. To be able to start more Kafka Streams instances without risking running out of memory we adjust the resource limits for these containers in `shuffle-kstreams-deployment.yaml` (listing 9).

Adjusting resource limits in `shuffle-kstreams/shuffle-kstreams-deployment.yaml`

```
resources:
  requests:
    memory: 1Gi
    cpu: 500m
  limits:
    memory: 2Gi
    cpu: 1000m
```

We also reduce the number of partitions from 100 to 20 for both the input topic in `shuffle-kstreams/input-topic.yaml` (listing 10), and the output topic in `shuffle-kstreams/output-topic.yaml` (listing 11).

### 3.4.7 Executing benchmarks and collecting results

With all the prerequisites deployed, running the benchmarks is done by simply creating an `execution` custom resource that Theodolite conveniently defines. We create an instance of this resource by applying the corresponding manifest file.

Command to create `execution` resource in `setup.sh`

```
kubectl apply -f ${base_path}/kstreams-baseline-atp.yaml
```

The `kstreams-baseline-atp.yaml` manifest in listing 13 is based on the ad-hoc throughput baseline recipe from the ShuffleBench replication package. These adjustments were made to fit the reduced local setup:

- reduce load to 80000 messages per second (from 250000).
- set the number of KafkaStream instances to 1 at the beginning.
- reduce the number of matching rules (simulated clients) to 10,000 from 1,000,000.
- ensure all label selectors for node labels match the label "infra" of the single cluster node.

During execution all the components depicted in Figure 1 are active. The lifetime of the KafkaStreams and load generator deployments, and of the data flows to and from Kafka, are limited to the duration specified for the execution.

The execution state can be checked indirectly by monitoring metrics as mentioned in section 3.4.8 or directly by querying the custom resource:

```
kubectl get executions
NAME                                STATUS     DURATION   AGE
shufflebench-kstreams-baseline-atp  Finished   20m        26m
```

After the execution finishes, the results are stored in a persistent volume that is not directly accessible. They should be copied to a folder local to the host machine before deleting the execution resource.

```
kubectl cp $(kubectl get pod -l app=theodolite \
            -o jsonpath="{.items[0].metadata.name}"):results \
        $output_path -c results-access
```

The results folder contains time series corresponding to each SLOs defined for the benchmark in listing 12, represented as one CSV file or more.

Note that, if the provided setup script was sourced (as recommended), the current command line session was enhanced with simplified commands for repeating executions and copying the results after each execution.

```
execute-benchmark
copy-results <output path>
```

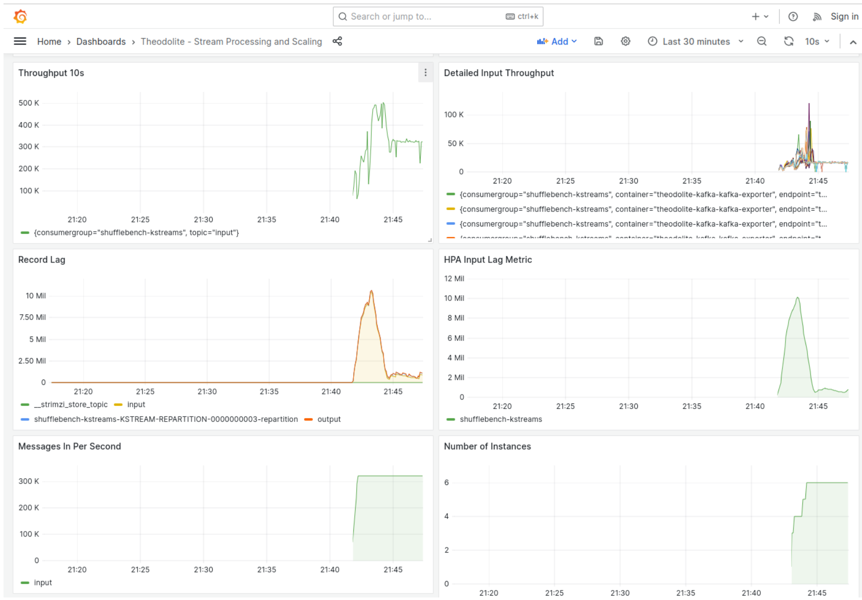


Figure 2: The Grafana Dashboard.

### 3.4.8 Live monitoring

During the execution of the benchmarks, there are several available ways of visualizing any SLO metrics, as well as monitoring the status of the many resources that are involved.

The following command is a convenient way to launch the Grafana web app in a new browser window by connecting to the Grafana service that Theodolite has configured and deployed for us.

```
minikube service -p ${cluster_name} theodolite-grafana
```

As detailed in section 3.4.3, the existing dashboard "Theodolite - Stream Processing and Scaling" includes plots that represent the evolution of metrics such as lag, throughput, CPU utilization for the executor instances, etc. Figure 2 shows a snapshot of this Grafana dashboard taken during the execution of the benchmark.

The Kubernetes Dashboard is another very useful tool that we can use, and it is conveniently included by default with the minikube distribution. The following command launches another browser window for this dashboard:

```
minikube dashboard -p ${cluster_name}
```

With it, we can easily monitor CPU and memory utilization, logs, and events. They can be viewed at the cluster node level, but also for any pod or groups of pods. It's also possible to see details about most active Kubernetes resources, within a more friendly user interface than using the many variations of the `kubectl` command. Figure 3 shows an example of using the Kubernetes Dashboard.

Of particular interest for this work is observing the evolution of the HPA state during the experiment. We can run the following command in a separate terminal window to get a glimpse at the metric values and corresponding scaling decisions taken by HPA.

```
kubectl get hpa hpa-shuffle-kstreams --watch
```

### 3.4.9 Cleanup

The script `cleanup.sh` in listing 5 is provided to delete all resources that were created and delete the cluster.

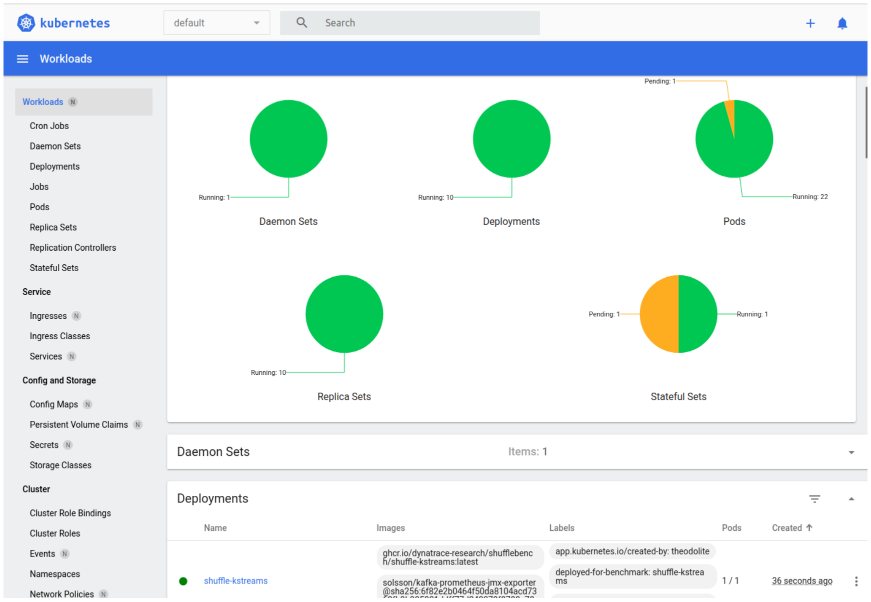


Figure 3: The Kubernetes Dashboard.

## 4 Results

Out of the multitude of scenarios of adaptability to varying workloads, I have focused on characterizing a baseline behavior: attempting to find the optimal number of parallel application instances while being subjected to a stable workload from the start. There are two important criteria for correctness which should be balanced: one is application *performance*, which is the ability to maintain service level objectives (e.g., low lag and high throughput), and the other is *efficiency*, which we interpret here as using the least number of instances.

In the following, I will describe a few of the experimental runs and discuss key observations, with a focus on performance and efficiency.

Note that the first few runs did not involve autoscaling with HPA. For replication, if HPA was already added it should be deleted:

```
kubectl delete hpa hpa-shuffle-kstreams
# ... Run experiments without HPA.
# Then add back HPA
kubectl apply -f ${base_path}/hpa-custom.yaml
```

### 4.1 Fixed number of instances

In an attempt to determine the ideal steady-state number of instances to handle the workload level described in section 3.4.7 we will look at 15 minutes executions using a fixed number of pods. Starting with one pod, which is clearly insufficient, each subsequent execution adds one more pod. Figure 4 shows the distributions of the values collected for key metrics during these experiments.

Given the workload, an optimum allocation of instances would result in:

- CPU utilization near 100%;
- Throughput around 320,000 records per second to match the input rate;
- Lag of at most one million records at any time.

We can see that with only 2 pods we already have very good characteristics, and with 3 pods both the throughput and lag metrics are stable near the desired values. At a stable state, assuming perfect balancing among Kafka Streams instances, adding more than 3 pods would be an inefficient use of resources.

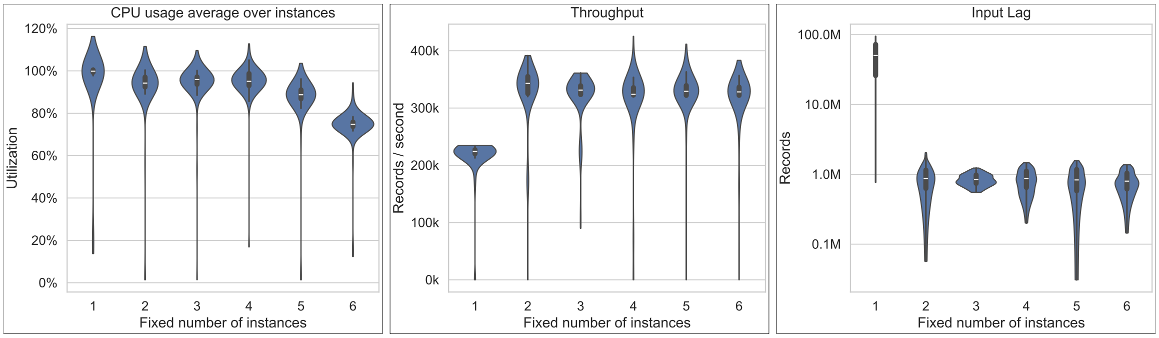


Figure 4: Distributions of measurements taken over 15 minutes of running with fixed number of instances.

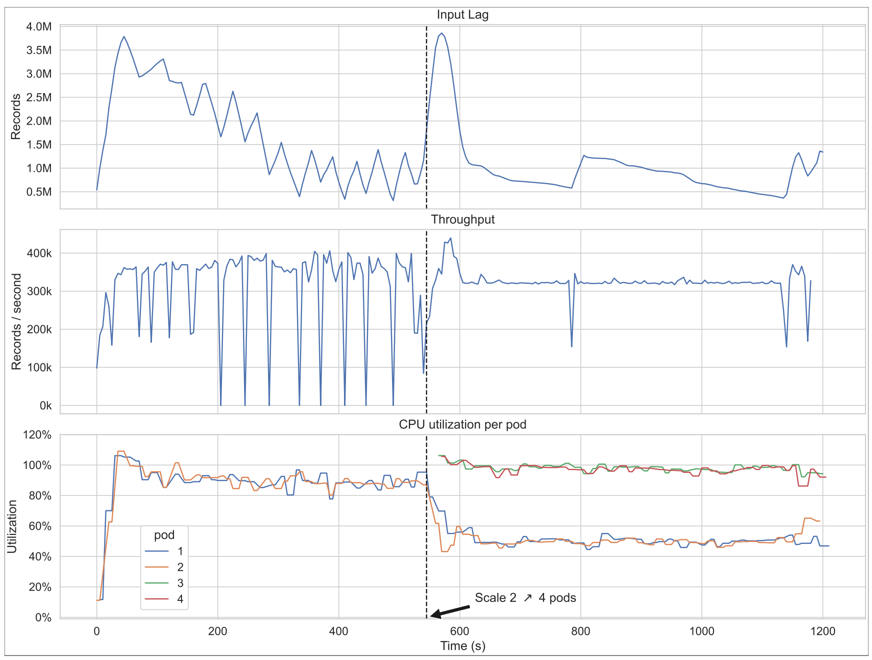


Figure 5: 20 minutes experiment starting with 2 pods and manually scaled up to 4 pods.

### 4.2 Scale up manually

The next experiment consisted of a 20 minutes execution which started with 2 Kafka Stream pods and scaled up manually to 4 pods:

```
kubectl scale deployment shuffle-kstreams --replicas=4
```

Figure 5 shows the evolution of the relevant metrics, this time detailing the CPU utilization of each pod. We can see that after a short stabilization time, both throughput and lag are more steady around their desired levels. However, we can also observe undesired behavior. Firstly, we notice a sudden increase in lag, even though the existing instances — which were previously able to maintain a good lag level — continue to run. This is an indication of a significant re-balancing overhead in Kafka Streams which reduces the records processing speed. Secondly, the CPU utilization of the existing 2 pods dropped to only 50% while the newly added pods continue at nearly 100%. The load is no longer balanced across all pods. The working hypothesis about this behavior is that it emerges from a bug or limitation in Kafka Streams. The broken assumption of a well-balanced load distribution will unfortunately *skew the results of autoscaling experiments*. Vogel et al. [15] also identified this type of problems exhibited by Kafka Streams — in the context of fault recovery — and report that significant performance improvements can be achieved through rigorous configuration tuning. In future work, Kafka Streams configuration parameters values that minimize load distribution discrepancies should be found.

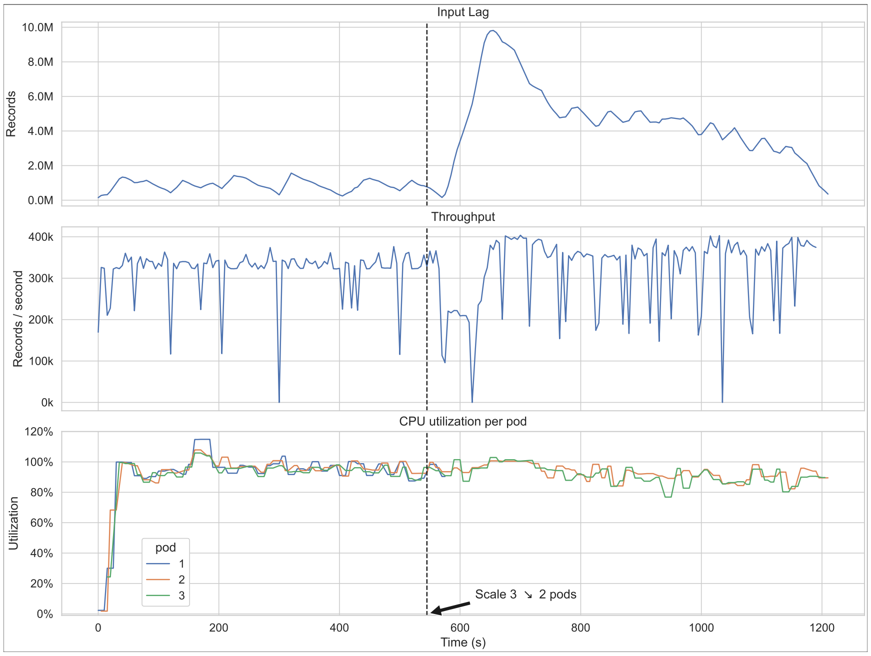


Figure 6: 20 minutes experiment starting with 3 pods and manually scaled down to 2 pods.

### 4.3 Scale down manually

The next experiments are trying to characterize scaling down behavior in a controlled way, again by triggering the scaling events manually.

Figure 6 shows the evolution of a 20 minutes execution which started with 3 pods and was scaled down to 2 pods. We notice that in the case of removing an instance, there is also a significant rebalancing overhead in Kafka Streams, which causes a quick increase in lag. The two remaining instances are subsequently unable to reduce the lag to the desired level, although we have seen that in stable conditions two instances were indeed able to sustain a low lag.

In the experiment illustrated in Figure 7 an execution started with 4 pods, and later was scaled down to 3 pods. We observe again the spike in lag linked to rebalancing overhead. This time lag was quickly reduced back to previous levels.

From the two experiments above it appears that it might be necessary to over-provision pods, to be able to absorb the overhead of rebalancing on scaling events with minimum impact on performance.

We should also note that when removing an instance, the CPU utilization levels of the remaining pods are approximately equally high, which is an indication that, in this case, the load remains well-balanced among all instances.

### 4.4 Autoscaling with HPA

The next experiments, illustrated in figure 8 introduced HPA to attempt to automatically adjust the number of pods to the workload after starting from a single instance.

#### 4.4.1 HPA using only CPU utilization

Before refining the HPA criteria it's useful to check how robust is the simplest approach, which is to base the scaling decision solely on the CPU utilization metric, taken as an average across all running instances. The panels (a) and (b) in Figure 8 illustrate two such runs with the target metric set first at 90% CPU utilization, and then at 80%.

A target CPU utilization of 90% addresses the desire to optimize for efficiency, and have fewer parallel instances which should be fully utilized. As seen in the figure, this approach yielded very poor performance

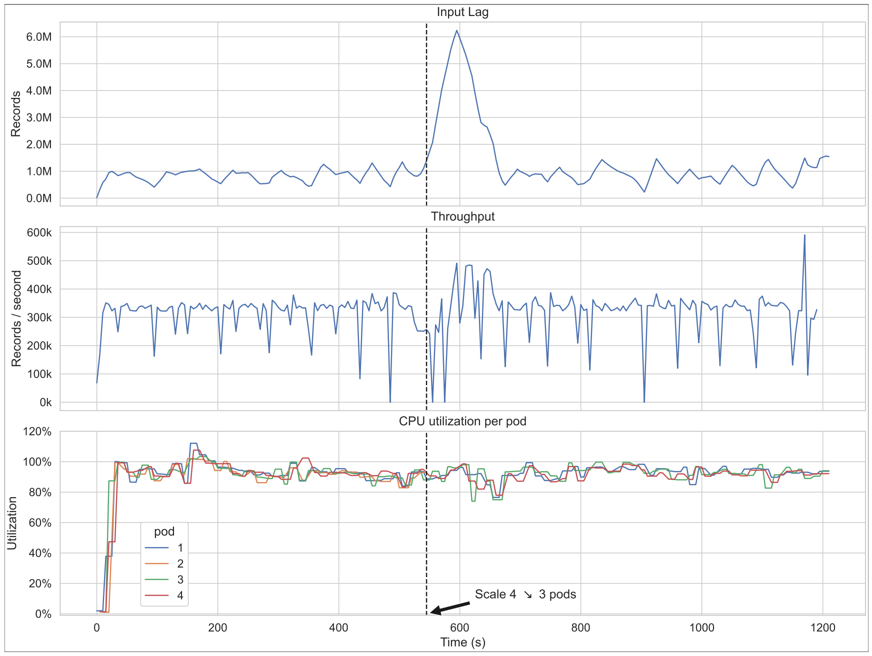


Figure 7: 20 minutes experiment starting with 4 pods and manually scaled down to 3 pods.

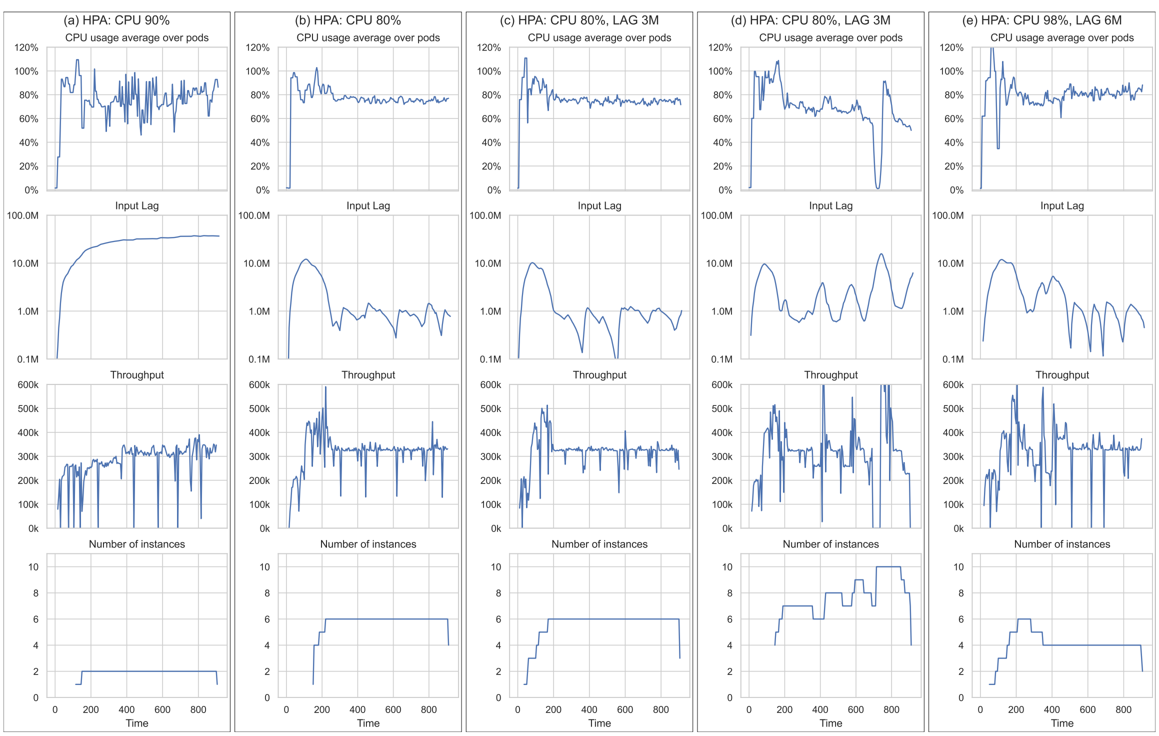


Figure 8: 15 minutes experiments with HPA autoscaling based on: (a) CPU utilization 90%; (b) CPU utilization 80%; (c) CPU utilization 80% and lag 3 million records; (d) a separate run with the setup from (c); (e) CPU utilization 98% and lag 6 million records.

and failed to keep lag under control. The lag metric after 15 minutes remained at 36 million records (note that the $y$ scale of this graph is logarithmic). Also, HPA never scaled to more than 2 instances, to have a chance to reduce this lag. This is a consequence of the unbalanced load resulting after the first scaling event (a Kafka Stream bug, as noted previously), which made the average CPU utilization (top graph) between the two pods be well below the target.

The next run has a target value for the CPU utilization metric lowered to 80%. We can see now that HPA has quickly scaled to 6 instances, which was beneficial for performance as observed from the lag and throughput metrics. Unfortunately, the deployment is rather inefficient. We know from previous tests that such a high number of instances is not truly necessary to handle the workload — 4 instances can achieve the same level of performance.

### 4.4.2 HPA using CPU utilization and lag

We would like to explicitly account for lag to both prevent it from growing too much and reduce it faster by scaling up in a timely fashion. Also, this metric should be more robust and cannot be skewed if there is an imbalance of load between instances. On the other hand, the lag metric cannot be used by itself because on the downward slope, after a peak value, it's desirable to sustain the current processing rate rather than reduce resources. The CPU utilization metric ensures that scaling down doesn't occur unless current instances are underused.

The panels (c) and (d) in Figure 8 illustrate two executions having the same HPA settings with two metrics: CPU utilization with a target of 80% and input lag with a target value of 3 million records. The outcome for the former is very similar to experiments that were based only on the CPU metric. The latter exhibited an oscillating scaling behavior, finishing with an excessive number of 10 instances running. The oscillations were triggered by the lag spike associated with removing one pod. Because scaling down causes more than 3 million records to accumulate (presumably during the time the streaming engine redistributes partitions among the remaining instances) a scaling-up decision quickly follows, which shortly recreates the conditions for scaling down. The lag spikes associated with scaling events are a complicating factor that has to be taken into account by a robust scaling policy. Remember from the manual scaling experiments in sections 4.2 and 4.3 that these spikes were identified both when adding and when removing instances.

The rightmost panel (e) in Figure 8 corresponds to a configuration with a CPU utilization target set to 98% and a lag target of 6 million records. This very high CPU utilization target allows a more aggressive scaling down to happen, and the lag target is not sensitive to the temporary spikes caused by these scaling events, thus avoiding oscillation. While we do see an initial moderate degradation of performance, HPA converges to a deployment size that is close to optimal from both the performance and efficiency perspectives.

## 5 Conclusion

A significant part of the practical work was dedicated to implementing the setup and configuration detailed in section 3. Creating and documenting the experimental setup is the main result, as it constitutes a solid foundation for further exploration of automatic scaling approaches. In particular, using HPA with ShuffleBench and provisioning external metrics from Prometheus queries is a novel approach that could potentially be integrated into rigorous benchmarking of streaming applications when subjected to horizontal autoscaling.

Through experimentation, I exposed serious limitations to simply scaling streaming applications using threshold-based policies and HPA, which will be summarized below. These findings provide a motivation to investigate more advanced alternatives in future work.

An important aspect is that horizontal scaling events are by themselves disruptive, at least temporary, to the processing flow as could be observed in sections 4.2 and 4.3. The reality of lag accumulation on scaling events makes it necessary to over-provision and dampen reactivity. This is an overhead expected from any self balancing distributed system, and it remains to be evaluated how Kafka Streams compares to other stream processing engines in this aspect.

Section 4.2 has shown that Kafka Streams currently suffers from an anomalous, unbalanced, redistribution of work between newly added instances and already running instances. Because some instances are significantly underutilized, the basic CPU utilization metric becomes unreliable. Vogel et al. [15] also identified this type of problems exhibited by Kafka Streams — in the context of fault recovery — and report that significant performance improvements can be achieved through rigorous configuration tuning. Vogel et al. [15] propose a configuration tuning approach to mitigate the problem.

Lastly, it is difficult to find an HPA configuration and particularly adequate threshold values, to satisfy both performance and efficiency constraints. Determining the configuration through trial and error is a time-consuming process. It becomes obvious that a truly self-adaptive system would be required to dynamically adjust its scaling policy.

For future work, one direction to follow is performing more experiments to comprehensively characterize DSP autoscaling with HPA. To this end, in the reduced local setup it is still necessary to generate repeatable variable workloads, and also try other DSP frameworks like Apache Flink with Reactive Mode and Apache Spark. Moving forward, for statistically significant results, experiments will be performed in provider cloud environments with larger clusters and realistic workload volumes. Another direction is to apply the findings of these experiments to inform the design of more advanced autoscaling controllers which may also make use of Artificial Intelligence techniques such as Time Series Forecasting with sequence models or Reinforcement Learning.

## References


[1] V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo Russo. Auto-Scaling in Data Stream Processing Applications: A Model-Based Reinforcement Learning Approach. In S. Balsamo, A. Marin, and E. Vicario, editors, *New Frontiers in Quantitative Methods in Informatics*, Communications in Computer and Information Science, pages 97–110, Cham, 2018. Springer International Publishing. ISBN 978-3-319-91632-3. doi: 10.1007/978-3-319-91632-3_8.

[2] V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo Russo. Decentralized self-adaptation for elastic Data Stream Processing. *Future Generation Computer Systems*, 87:171–185, Oct. 2018. ISSN 0167739X. doi: 10.1016/j.future.2018.05.025. URL https://linkinghub.elsevier.com/retrieve/pii/S0167739X17326821.

[3] V. Cardellini, F. Lo Presti, M. Nardelli, and G. R. Russo. Runtime Adaptation of Data Stream Processing Systems: The State of the Art. *ACM Computing Surveys*, 54(11s):1–36, Jan. 2022. ISSN 0360-0300, 1557-7341. doi: 10.1145/3514496. URL https://dl.acm.org/doi/10.1145/3514496.

[4] M. Geldenhuys, D. Scheinert, O. Kao, and L. Thamsen. Demeter: Resource-Efficient Distributed Stream Processing under Dynamic Loads with Multi-Configuration Optimization, Mar. 2024. URL http://arxiv.org/abs/2403.02129. arXiv:2403.02129 [cs].

[5] K. Gontarska, M. Geldenhuys, D. Scheinert, P. Wiesner, A. Polze, and L. Thamsen. Evaluation of Load Prediction Techniques for Distributed Stream Processing. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*, pages 91–98, Oct. 2021. doi: 10.1109/IC2E52221.2021.00023. URL https://ieeexplore.ieee.org/document/9610459.

[6] S. Henning and W. Hasselbring. Theodolite: Scalability Benchmarking of Distributed Stream Processing Engines in Microservice Architectures. *Big Data Research*, 25:100209, July 2021. ISSN 2214-5796. doi: 10.1016/j.bdr.2021.100209. URL https://www.sciencedirect.com/science/article/pii/S2214579621000265.

[7] S. Henning, A. Vogel, M. Leichtfried, O. Ertl, and R. Rabiser. ShuffleBench: A Benchmark for Large-Scale Data Shuffling Operations with Distributed Stream Processing Frameworks. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*, ICPE '24, pages 2–13, New York, NY, USA, May 2024. Association for Computing Machinery. ISBN 9798400704444. doi: 10.1145/3629526.3645036. URL https://dl.acm.org/doi/10.1145/3629526.3645036.

[8] S. Imai, S. Patterson, and C. A. Varela. Uncertainty-Aware Elastic Virtual Machine Scheduling for Stream Processing Systems. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 62–71, Washington, DC, USA, May 2018. IEEE. ISBN 978-1-5386-5815-4. doi: 10.1109/CCGRID.2018.00021. URL https://ieeexplore.ieee.org/document/8411010/.

[9] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. pages 783–798, 2018. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/kalavri.

[10] F. Lombardi, L. Aniello, S. Bonomi, and L. Querzoni. Elastic Symbiotic Scaling of Operators and Resources in Stream Processing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 29(3):572–585, Mar. 2018. ISSN 1558-2183. doi: 10.1109/TPDS.2017.2762683. URL `https://ieeexplore.ieee.org/document/8067517`. Conference Name: IEEE Transactions on Parallel and Distributed Systems.

[11] B. J. J. Pfister, D. Scheinert, M. K. Geldenhuys, and O. Kao. Daedalus: Self-Adaptive Horizontal Autoscaling for Resource Efficiency of Distributed Stream Processing Systems, Mar. 2024. URL `http://arxiv.org/abs/2403.02093`. arXiv:2403.02093 [cs].

[12] G. R. Russo, V. Cardellini, and F. L. Presti. Reinforcement Learning Based Policies for Elastic Stream Processing on Heterogeneous Resources. In *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems*, pages 31–42, Darmstadt Germany, June 2019. ACM. ISBN 978-1-4503-6794-3. doi: 10.1145/3328905.3329506. URL `https://dl.acm.org/doi/10.1145/3328905.3329506`.

[13] G. Russo Russo, V. Cardellini, and F. Lo Presti. Hierarchical Auto-scaling Policies for Data Stream Processing on Heterogeneous Resources. *ACM Transactions on Autonomous and Adaptive Systems*, 18 (4):14:1–14:44, Oct. 2023. ISSN 1556-4665. doi: 10.1145/3597435. URL `https://dl.acm.org/doi/10.1145/3597435`.

[14] G. Siachamis, J. Kanis, W. Koper, K. Psarakis, M. Fragkoulis, A. Van Deursen, and A. Katsifodimos. Towards Evaluating Stream Processing Autoscalers. In *2023 IEEE 39th International Conference on Data Engineering Workshops (ICDEW)*, pages 95–99, Anaheim, CA, USA, Apr. 2023. IEEE. ISBN 9798350322446. doi: 10.1109/ICDEW58674.2023.00021. URL `https://ieeexplore.ieee.org/document/10148161/`.

[15] A. Vogel, S. Henning, E. Perez-Wohlfeil, O. Ertl, and R. Rabiser. High-level Stream Processing: A Complementary Analysis of Fault Recovery, May 2024. URL `http://arxiv.org/abs/2405.07917`. arXiv:2405.07917 [cs].

[16] G. Wang, L. Chen, A. Dikshit, J. Gustafson, B. Chen, M. J. Sax, J. Roesler, S. Blee-Goldman, B. Cadonna, A. Mehta, V. Madan, and J. Rao. Consistency and Completeness: Rethinking Distributed Stream Processing in Apache Kafka. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2602–2613, Virtual Event China, June 2021. ACM. ISBN 978-1-4503-8343-1. doi: 10.1145/3448016.3457556. URL `https://dl.acm.org/doi/10.1145/3448016.3457556`.

## Listings

Listing 1: minikube installation in Ubuntu 22 (minikube-docker-setup.sh).

```
#!/bin/bash

if [ $(lsb_release -is) = "Ubuntu" ]
then
    echo "Running on Ubuntu $(lsb_release -rs)"
else
    echo "I'm sorry, this script is made only for Ubuntu (confirmed on version 22)"
fi

### installing Docker
sudo apt-get update -y
sudo apt-get install ca-certificates curl gnupg lsb-release -y
sudo mkdir -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update -y
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-compose-plugin -y

sudo usermod -aG docker $USER
newgrp docker

### installing kubectl
curl -LO https://storage.googleapis.com/kubernetes-release/release/`curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt`/bin/linux/amd64/kubectl
chmod +x ./kubectl
sudo mv ./kubectl /usr/local/bin/kubectl

### installing minikube
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube_latest_amd64.deb
sudo dpkg -i minikube_latest_amd64.deb

### installing Helm
curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3
chmod +x ./get_helm.sh
./get_helm.sh


# Modify inotify limits
sudo tee /etc/sysctl.d/55-custom-inotify.conf > /dev/null <<EOT
fs.inotify.max_user_instances = 1024
fs.inotify.max_user_watches = 1048576
EOT
sudo sysctl -p /etc/sysctl.d/55-custom-inotify.conf

rm -f minikube_latest_amd64.deb get_helm.sh

echo "Enabling bash autocompletion for new commands (ready to use in new shell)."
cat >> ~/.bashrc <<EOT

# Autocompletion for kubectl and friends
source <(kubectl completion bash)
```

```
source <(minikube completion bash)
source <(helm completion bash)
EOT
```

Listing 2: Role binding for storage provisioner (rbac-storage-provisioner.yaml).

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: storage-provisioner-cluster-role-binding
  namespace: kube-system
subjects:
- kind: ServiceAccount
  name: storage-provisioner
  namespace: kube-system
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
```

Listing 3: Storage class for Kafka pods (kafka-storage-class.yaml).

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: kafka
  annotations:
    storageclass.kubernetes.io/is-default-class: "false"
provisioner: k8s.io/minikube-hostpath
reclaimPolicy: Delete
volumeBindingMode: WaitForFirstConsumer
```

Listing 4: Setup ShuffleBench for running experiments (setup.sh).

```
#!/usr/bin/env bash
# It's recommended to source this script

## The first argument passed (if any) is the name of the cluster
cluster_name=${1:-sbcl}

if [[ $0 = ${BASH_SOURCE[0]} ]]
then
    echo -e "\a\nIt's recommended to source this script!\n"
fi

if [[ $(pwd) =~ ShuffleBench/kubernetes$ ]]
then
    echo "Running in ShuffleBench/kubernetes, good."
    if [[ ! -d "$(pwd)/theodolite" ]]
    then
        echo "Please also clone theodolite into ShuffleBench/kubernetes ↩
    folder"
        echo "e.g.: git clone https://github.com/cau-se/theodolite.git"
        if [[ $0 = ${BASH_SOURCE[0]} ]]; then
            exit 1
        else
            return 1
        fi
    fi
else
    echo "Please invoke this script from ShuffleBench/kubernetes"
    echo "If necessary, clone ShuffleBench from https://github.com/↩
    dynatrace-research/ShuffleBench.git"
    if [[ $0 = ${BASH_SOURCE[0]} ]]; then
```

```
        exit 1
    else
        return 1
    fi
fi

base_path=$(dirname ${BASH_SOURCE[0]})

## Uncomment export statement if using HPA own CPU metric per container (instead  of per pod):
## metrics.type = ContainerResource
# export KUBE_FEATURE_GATES=HPAContainerMetrics=true

minikube start --nodes 1 -p ${cluster_name} --addons metrics-server dashboard storage-provisioner default-storageclass --cni calico --cpus no-limit --memory no-limit --wait=all

kubectl wait --all --timeout=10m --for=condition=Ready pods

minikube addons -p ${cluster_name} enable dashboard

kubectl label nodes ${cluster_name} type=infra
# kubectl label nodes ${cluster_name}-m02 type=sut
# kubectl label nodes ${cluster_name}-m03 type=kafka


kubectl apply -f ${base_path}/rbac-storage-provisioner.yaml
kubectl apply -f ${base_path}/kafka-storage-class.yaml


helm dependencies update theodolite/helm

helm install theodolite theodolite/helm -f https://raw.githubusercontent.com/cau-se/theodolite/main/helm/preconfigs/extended-metrics.yaml -f ${base_path}/values-theodolite-local.yaml

kubectl wait --all --timeout=15m --for=condition=Ready pods

# Add a few more plots to the dashboard in Grafana
kubectl replace -f ${base_path}/dashboard-config-map.yaml
kubectl scale deployment theodolite-grafana --replicas=0 && kubectl scale deployment theodolite-grafana --replicas=1

helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm repo update
helm install prom-adapt prometheus-community/prometheus-adapter -f ${base_path}/values-prom-adapt.yaml


kubectl delete configmaps --ignore-not-found=true shufflebench-resources-load-generator shufflebench-resources-latency-exporter shufflebench-resources-kstreams
kubectl create configmap shufflebench-resources-load-generator --from-file ${base_path}/shuffle-load-generator/
kubectl create configmap shufflebench-resources-latency-exporter --from-file ./shuffle-latency-exporter/
kubectl create configmap shufflebench-resources-kstreams --from-file ${base_path}/shuffle-kstreams/

kubectl apply -f ${base_path}/theodolite-benchmark-kstreams.yaml

## The execution should be started separately (e.g. by using the execute-banchmark function)
# kubectl apply -f ${base_path}/kstreams-baseline-atp.yaml
```

```
# kubectl autoscale deployment shuffle-kstreams --cpu-percent=70 --min=1 --max=10

# kubectl apply -f ${base_path}/hpa-sb-container-cpu.yaml

kubectl apply -f ${base_path}/hpa-authorization.yaml
kubectl apply -f ${base_path}/hpa-custom.yaml

execute-benchmark() {
  kubectl delete --ignore-not-found=true execution shufflebench-kstreams-baseline-atp
  kubectl apply -f ${base_path}/kstreams-baseline-atp.yaml
}

copy-results() {
  # The first argument is the name of the local output folder
  output_path=${1:-results}
  kubectl cp $(kubectl get pod -l app=theodolite -o jsonpath="{.items[0].metadata.name}"):results $output_path -c results-access
}
```

Listing 5: Delete all pods and cluster (cleanup.sh).

```
#!/usr/bin/env bash

## The optional first argument is the cluster name.
cluster_name=${1:-sbcl}

kubectl delete --ignore-not-found=true execution shufflebench-kstreams-baseline-atp
kubectl delete --ignore-not-found=true  benchmark shuffle-kstreams
kubectl delete configmaps --ignore-not-found=true shufflebench-resources-load-generator shufflebench-resources-latency-exporter shufflebench-resources-kstreams shufflebench-resources-hzcast shufflebench-resources-flink shufflebench-resources-spark
helm uninstall prom-adapt
helm uninstall theodolite
sleep 20s
minikube delete -p $cluster_name
```

Listing 6: Theodolite custom values for Helm install (values-theodolite-local.yaml).

```
kafkaClient:
  enabled: true
  nodeSelector:
    type: infra
kcat:
  enabled: true
  nodeSelector:
    type: infra

strimzi:
  kafka:
    replicas: 3
    config:
      "auto.create.topics.enable": false
      "log.retention.ms": "7200000" # 2h
      "transaction.max.timeout.ms": "7200000" # 2h
    jvmOptions: {}
    storage:
      type: persistent-claim
      size: 80Gi
      class: kafka
```

```yaml
      deleteClaim: true
  zookeeper:
    zooEntrance:
      enabled: false
      zookeeperClient:
        enabled: false
  schemaRegistry:
    enabled: false

kube-prometheus-stack:
  prometheus:
    prometheusSpec:
      scrapeInterval: 5s

operator:
  resultsVolume:
    persistent:
      enabled: true
  # We do not use the streaming benchmarks from Theodolite
  theodoliteBenchmarks:
    resourceConfigMaps:
      uc1LoadGenerator: false
      uc1Kstreams: false
      uc1Flink: false
      uc1Hazelcast: false
      uc1BeamFlink: false
      uc1BeamSamza: false
      uc2LoadGenerator: false
      uc2Kstreams: false
      uc2Flink: false
      uc2Hazelcast: false
      uc2BeamFlink: false
      uc2BeamSamza: false
      uc3LoadGenerator: false
      uc3Kstreams: false
      uc3Flink: false
      uc3Hazelcast: false
      uc3BeamFlink: false
      uc3BeamSamza: false
      uc4LoadGenerator: false
      uc4Kstreams: false
      uc4Flink: false
      uc4Hazelcast: false
      uc4BeamFlink: false
      uc4BeamSamza: false
    benchmarks:
      uc1Kstreams: false
      uc1Flink: false
      uc1Hazelcast: false
      uc1BeamFlink: false
      uc1BeamSamza: false
      uc2Kstreams: false
      uc2Flink: false
      uc2Hazelcast: false
      uc2BeamFlink: false
      uc2BeamSamza: false
      uc3Kstreams: false
      uc3Flink: false
      uc3Hazelcast: false
      uc3BeamFlink: false
      uc3BeamSamza: false
      uc4Kstreams: false
      uc4Flink: false
      uc4Hazelcast: false
      uc4BeamFlink: false
```

```
      uc4BeamSamza: false
```

Listing 7: New Grafana dashboard (partial content) (dashboard-config-map.yaml).

```
apiVersion: v1
data:
  k8s-dashboard.json: |-
    {
      "annotations": {
        "list": [
          {
            "builtIn": 1,
            "datasource": {
              "type": "datasource",
              "uid": "grafana"
            },
            "enable": true,
            "hide": true,
            "iconColor": "rgba(0, 211, 255, 1)",
            "name": "Annotations & Alerts",
            "target": {
              "limit": 100,
              "matchAny": false,
              "tags": [],
              "type": "dashboard"
            },
            "type": "dashboard"
          }
        ]
      },
      "editable": true,
      "fiscalYearStartMonth": 0,
      "graphTooltip": 1,
      "id": 2,
      "links": [],
      "liveNow": false,
      "panels": [
        {
          "datasource": {
            "type": "prometheus",
            "uid": "prometheus"
          },
          "fieldConfig": {
            "defaults": {
              "color": {
                "mode": "palette-classic"
              },
              "custom": {
                "axisCenteredZero": false,
                "axisColorMode": "text",
                "axisLabel": "",
                "axisPlacement": "auto",
                "barAlignment": 0,
                "drawStyle": "line",
                "fillOpacity": 0,
                "gradientMode": "none",
                "hideFrom": {
                  "legend": false,
                  "tooltip": false,
                  "viz": false
                },
                "lineInterpolation": "linear",
                "lineWidth": 1,
                "pointSize": 5,
                "scaleDistribution": {
```

```
                "type": "linear"
              },
              "showPoints": "auto",
              "spanNulls": false,
              "stacking": {
                "group": "A",
                "mode": "none"
              },
              "thresholdsStyle": {
                "mode": "off"
              }
            },
            "mappings": [],
            "thresholds": {
              "mode": "absolute",
              "steps": [
                {
                  "color": "green",
                  "value": null
                },
                {
                  "color": "red",
                  "value": 80
                }
              ]
            }
          },
          "overrides": []
        },
        "gridPos": {
          "h": 8,
          "w": 12,
          "x": 0,
          "y": 0
        },
        "id": 15,
        "options": {
          "legend": {
            "calcs": [],
            "displayMode": "list",
            "placement": "bottom",
            "showLegend": true
          },
          "tooltip": {
            "mode": "single",
            "sort": "none"
          }
        },
        "targets": [
          {
            "datasource": {
              "type": "prometheus",
              "uid": "prometheus"
            },
            "editorMode": "code",
            "expr": "avg by (pod) (irate(container_cpu_usage_seconds_total{pod=~\"shuffle-kstreams.*\"}[30s])) * 100",
            "instant": false,
            "range": true,
            "refId": "A"
          }
        ],
        "title": "CPU per Pod",
        "type": "timeseries"
```

```
        },
        {
          "datasource": {
            "type": "prometheus",
            "uid": "prometheus"
          },
          "fieldConfig": {
            "defaults": {
              "color": {
                "mode": "palette-classic"
              },
              "custom": {
                "axisCenteredZero": false,
                "axisColorMode": "text",
                "axisLabel": "",
                "axisPlacement": "auto",
                "barAlignment": 0,
                "drawStyle": "line",
                "fillOpacity": 0,
                "gradientMode": "none",
                "hideFrom": {
                  "legend": false,
                  "tooltip": false,
                  "viz": false
                },
                "lineInterpolation": "linear",
                "lineWidth": 1,
                "pointSize": 5,
                "scaleDistribution": {
                  "type": "linear"
                },
                "showPoints": "auto",
                "spanNulls": false,
                "stacking": {
                  "group": "A",
                  "mode": "none"
                },
                "thresholdsStyle": {
                  "mode": "off"
                }
              },
              "mappings": [],
              "thresholds": {
                "mode": "absolute",
                "steps": [
                  {
                    "color": "green",
                    "value": null
                  },
                  {
                    "color": "red",
                    "value": 80
                  }
                ]
              }
            },
            "overrides": []
          },
          "gridPos": {
            "h": 8,
            "w": 12,
            "x": 12,
            "y": 0
          },
          "id": 16,
```

```
          "options": {
            "legend": {
              "calcs": [],
              "displayMode": "list",
              "placement": "bottom",
              "showLegend": true
            },
            "tooltip": {
              "mode": "single",
              "sort": "none"
            }
          },
          "targets": [
            {
              "datasource": {
                "type": "prometheus",
                "uid": "prometheus"
              },
              "editorMode": "code",
              "expr": "avg(avg by (pod) (irate(container_cpu_usage_seconds_total{pod=~\"shuffle-kstreams.*\"}[30s])) * 100) without (pod)",
              "instant": false,
              "range": true,
              "refId": "A"
            }
          ],
          "title": "HPA CPU Metric",
          "type": "timeseries"
        },
        {
          "datasource": {
            "type": "prometheus",
            "uid": "prometheus"
          },
          "fieldConfig": {
            "defaults": {
              "color": {
                "mode": "palette-classic"
              },
              "custom": {
                "axisCenteredZero": false,
                "axisColorMode": "text",
                "axisLabel": "",
                "axisPlacement": "auto",
                "barAlignment": 0,
                "drawStyle": "line",
                "fillOpacity": 0,
                "gradientMode": "none",
                "hideFrom": {
                  "legend": false,
                  "tooltip": false,
                  "viz": false
                },
                "lineInterpolation": "linear",
                "lineWidth": 1,
                "pointSize": 5,
                "scaleDistribution": {
                  "type": "linear"
                },
                "showPoints": "auto",
                "spanNulls": false,
                "stacking": {
                  "group": "A",
                  "mode": "none"
```

```
              },
              "thresholdsStyle": {
                "mode": "off"
              }
            },
            "mappings": [],
            "thresholds": {
              "mode": "absolute",
              "steps": [
                {
                  "color": "green",
                  "value": null
                },
                {
                  "color": "red",
                  "value": 80
                }
              ]
            },
            "unit": "short"
          },
          "overrides": []
        },
        "gridPos": {
          "h": 8,
          "w": 12,
          "x": 0,
          "y": 8
        },
        "id": 13,
        "options": {
          "legend": {
            "calcs": [],
            "displayMode": "list",
            "placement": "bottom",
            "showLegend": true
          },
          "tooltip": {
            "mode": "single",
            "sort": "none"
          }
        },
        "targets": [
          {
            "datasource": {
              "type": "prometheus",
              "uid": "prometheus"
            },
            "editorMode": "code",
            "expr": "sum by(consumergroup,topic) (rate(kafka_consumergroup_current_offset{topic='input'}[10s]) >= 0)",
            "instant": false,
            "range": true,
            "refId": "A"
          }
        ],
        "title": "Throughput 10s",
        "type": "timeseries"
      },
    ],
    "refresh": "10s",
    "schemaVersion": 38,
    "style": "dark",
    "tags": [],
    "templating": {
```

```
          "list": []
        },
        "time": {
          "from": "now-30m",
          "to": "now"
        },
        "timepicker": {
          "refresh_intervals": [
            "5s",
            "10s",
            "30s",
            "1m",
            "5m",
            "15m",
            "30m",
            "1h",
            "2h",
            "1d"
          ]
        },
        "timezone": "",
        "title": "Theodolite - Stream Processing and Scaling",
        "uid": "dad0CNlSs",
        "version": 1,
        "weekStart": ""
      }
kind: ConfigMap
metadata:
  annotations:
    meta.helm.sh/release-name: theodolite
    meta.helm.sh/release-namespace: default
  labels:
    app.kubernetes.io/managed-by: Helm
    grafana_dashboard: "1"
  name: theodolite-grafana-scalability
```

Listing 8: Prometheus adapter custom values for Helm install (values-prom-adapt.yaml).

```
#logLevel: 6
prometheus:
  url: http://prometheus-operated
  port: 9090
  path: /
rules:
  default: true
  external:
    - metricsQuery: sum(avg_over_time(kafka_consumergroup_lag{ << .LabelMatchers >> }[20s])) by (topic,consumergroup)
      name:
        as: kafka_input_lag
      resources:
        template: <<.Resource>>
      seriesQuery: 'kafka_consumergroup_lag{topic="input",consumergroup="shufflebench-kstreams"}'
    - metricsQuery: 'avg(avg by (pod) (irate(container_cpu_usage_seconds_total{pod=~"shuffle-kstreams.*"}[30s])) * 100) without (pod)'
      name:
        as: consumer_cpu_utilization_30s
      resources:
        template: <<.Resource>>
      seriesQuery: container_cpu_usage_seconds_total
```

Listing 9: KafkaStream deployment (shuffle-kstreams/shuffle-kstreams-deployment.yaml).

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: shuffle-kstreams
spec:
  selector:
    matchLabels:
      app: shuffle-kstreams
  replicas: 9
  template:
    metadata:
      labels:
        app: shuffle-kstreams
    spec:
      terminationGracePeriodSeconds: 0
      containers:
        - name: shuffle-kstreams
          image: ghcr.io/dynatrace-research/shufflebench/shuffle-kstreams:latest
          ports:
            - containerPort: 5555
              name: jmx
          env:
            - name: KAFKA_BOOTSTRAP_SERVERS
              value: "theodolite-kafka-kafka-bootstrap:9092"
            - name: MATCHER_ZIPF_NUM_RULES
              value: "1000000"
            - name: MATCHER_ZIPF_TOTAL_SELECTIVITY
              value: "0.2"
            - name: MATCHER_ZIPF_S
              value: "0.0"
            - name: CONSUMER_INIT_COUNT_RANDOM
              value: "true"
            - name: "KAFKASTREAMS__COMMIT_INTERVAL_MS__"
              value: "5000"
            # - name: "KAFKASTREAMS__MAX_POLL_RECORDS__"
            #   value: "500"
            # - name: "KAFKASTREAMS__LINGER_MS__"
            #   value: "0"
            # - name: "KAFKASTREAMS__NUM_STREAM_THREADS__"
            #   value: "4"
            - name: JAVA_TOOL_OPTIONS
              value: "-Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false -Dcom.sun.management.jmxremote.port=5555"
          resources:
            requests:
              memory: 1Gi
              cpu: 500m
            limits:
              memory: 2Gi
              cpu: 1000m
        - name: prometheus-jmx-exporter
          image: "solsson/kafka-prometheus-jmx-exporter@sha256:6f82e2b0464f50da8104acd7363fb9b995001ddff77d248379f8788e78946143"
          command:
            - java
            - -XX:+UnlockExperimentalVMOptions
            - -XX:+UseCGroupMemoryLimitForHeap
            - -XX:MaxRAMFraction=1
            - -XshowSettings:vm
            - -jar
            - jmx_prometheus_httpserver.jar
            - "5556"
            - /etc/jmx-aggregation/jmx-kafka-prometheus.yml
```

```
          ports:
            - containerPort: 5556
          resources:
            requests:
              memory: 1500Mi
              cpu: 500m
            limits:
              memory: 2Gi
              cpu: 1000m
          volumeMounts:
            - name: jmx-config
              mountPath: /etc/jmx-aggregation
      volumes:
        - name: jmx-config
          configMap:
            name: shuffle-kstreams-jmx-configmap
```

Listing 10: Kafka input topic (shuffle-kstreams/input-topic.yaml).

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: input
  labels:
    strimzi.io/cluster: theodolite-kafka
spec:
  partitions: 20
  replicas: 1
  config:
    message.timestamp.type: LogAppendTime
```

Listing 11: Kafka output topic (shuffle-kstreams/output-topic.yaml).

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: output
  labels:
    strimzi.io/cluster: theodolite-kafka
spec:
  partitions: 20
  replicas: 1
  config:
    message.timestamp.type: LogAppendTime
```

Listing 12: Benchmark definition (theodolite-benchmark-kstreams-simple.yaml).

```
apiVersion: theodolite.rocks/v1beta1
kind: benchmark
metadata:
  name: shuffle-kstreams
  labels:
    sut: kstreams
spec:
  sut:
    resources:
      - configMap:
          name: shufflebench-resources-kstreams
          files:
          - "input-topic.yaml"
          - "output-topic.yaml"
          - "shuffle-kstreams-deployment.yaml"
          - "shuffle-kstreams-service.yaml"
          - "shuffle-kstreams-jmx-configmap.yaml"
```

```
        - "shuffle-kstreams-service-monitor.yaml"
    - configMap:
        name: shufflebench-resources-latency-exporter
        files:
        - "shuffle-latency-exporter-deployment.yaml"
        - "shuffle-latency-exporter-service-monitor.yaml"
        - "shuffle-latency-exporter-service.yaml"
  afterActions:
    - delete:
        selector:
          apiVersion: kafka.strimzi.io/v1beta2
          kind: KafkaTopic
          nameRegex: "^shufflebench-kstreams-.*"
loadGenerator:
  resources:
    - configMap:
        name: shufflebench-resources-load-generator
        files:
        - "shuffle-load-generator-service.yaml"
        - "shuffle-load-generator-deployment.yaml"
resourceTypes:
  - typeName: "Instances"
    patchers:
      - type: "ReplicaPatcher"
        resource: "shuffle-kstreams-deployment.yaml"
loadTypes:
  - typeName: "MessagesPerSecond"
    patchers:
      - type: "EnvVarPatcher"
        resource: "shuffle-load-generator-deployment.yaml"
        properties:
          container: "shuffle-load-generator"
          variableName: "NUM_RECORDS_PER_SOURCE_SECOND"
slos:
  - name: "lag trend"
    sloType: "lag trend"
    prometheusUrl: "http://prometheus-operated:9090"
    offset: 0
    properties:
      consumerGroup: shufflebench-kstreams
      thresholdRelToLoad: 0.01
      externalSloUrl: "http://localhost:80/evaluate-slope"
      warmup: 30 # in seconds
  # Normal, "read-only" metrics:
  - name: "throughput"
    sloType: generic
    prometheusUrl: "http://prometheus-operated:9090"
    offset: 0
    properties:
      externalSloUrl: "http://localhost:8082"
      promQLQuery: "sum by(consumergroup,topic) (rate(kafka_consumergroup_current_offset{topic='input'}[10s]) >= 0)"
      queryAggregation: mean
      repetitionAggregation: median
      operator: "true"
      threshold: 0
      warmup: 30 # in seconds
  - name: "detailedInputThroughput"
    sloType: generic
    prometheusUrl: "http://prometheus-operated:9090"
    offset: 0
    properties:
      externalSloUrl: "http://localhost:8082"
      promQLQuery: "rate(kafka_consumergroup_current_offset{topic='input'}[10s])"
```

```
        queryAggregation: mean
        repetitionAggregation: median
        operator: gte
        threshold: 0
        warmup: 30 # in seconds
        # promQLStepSeconds: 1 # <--
        # takeOnlyFirstMetric: "false" # <--
    - name: "CPUsPercentageUtilizationPerPod30s"
      sloType: generic
      prometheusUrl: "http://prometheus-operated:9090"
      offset: 0
      properties:
        externalSloUrl: "http://localhost:8082"
        promQLQuery: "avg by (pod) (irate(↩
    container_cpu_usage_seconds_total{pod=~\"shuffle-kstreams.*\"}[30s])) *↩
     100"
        queryAggregation: mean
        repetitionAggregation: median
        operator: gte
        threshold: 0
        warmup: 30 # in seconds
        # promQLStepSeconds: 1 # <--
        # takeOnlyFirstMetric: "false" # <--
    - name: "CPUsTotalUtilization30s"
      sloType: generic
      prometheusUrl: "http://prometheus-operated:9090"
      offset: 0
      properties:
        externalSloUrl: "http://localhost:8082"
        promQLQuery: "avg(avg by (pod) (irate(↩
    container_cpu_usage_seconds_total{pod=~\"shuffle-kstreams.*\"}[30s])) *↩
     100) without (pod)"
        queryAggregation: mean
        repetitionAggregation: median
        operator: gte
        threshold: 0
        warmup: 30 # in seconds
        # promQLStepSeconds: 1 # <--
```

Listing 13: Execution definition (kstreams-baseline-atp.yaml).

```
apiVersion: theodolite.rocks/v1beta1
kind: execution
metadata:
  name: shufflebench-kstreams-baseline-atp
spec:
  benchmark: shuffle-kstreams
  load:
    loadType: "MessagesPerSecond"
    loadValues: [80000] # reduce - 50000 # original 250000
  resources:
    resourceType: "Instances"
    resourceValues: [1] # reduce # original 9
  slos:
    - name: throughput
      properties: {}
  execution:
    metric: capacity
    strategy:
      name: "LinearSearch"
    duration: 1200 # in seconds # original 900
    loadGenerationDelay: 30 # in seconds # original 30
    repetitions: 1 # 1 should be enough # original 3
  configOverrides:
    - patcher:
```

```yaml
        type: "EnvVarPatcher"
        resource: "shuffle-kstreams-deployment.yaml"
        properties:
          container: "shuffle-kstreams"
          variableName: "MATCHER_ZIPF_NUM_RULES"
      value: "10000" # less, maybe 100000 # original 1000000
      # autoscaling config might be needed here
    - patcher:
        type: "NodeSelectorPatcher"
        resource: "shuffle-kstreams-deployment.yaml"
        properties:
          variableName: "type"
      value: "infra" # original "sut"
    - patcher:
        type: "NodeSelectorPatcher"
        resource: "shuffle-load-generator-deployment.yaml"
        properties:
          variableName: "type"
      value: "infra"
    - patcher:
        type: "NodeSelectorPatcher"
        resource: "shuffle-latency-exporter-deployment.yaml"
        properties:
          variableName: "type"
      value: "infra"
```

Listing 14: Configure authorization for HPA (hpa-authorization.yaml).

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: external-metrics-server-resources
rules:
  - apiGroups:
      - external.metrics.k8s.io
    resources: ["*"]
    verbs: ["*"]

---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: hpa-controller-external-metrics
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: external-metrics-server-resources
subjects:
  - kind: ServiceAccount
    name: horizontal-pod-autoscaler
    namespace: kube-system
```

Listing 15: HPA manifest (hpa-custom.yaml).

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: hpa-shuffle-kstreams
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: shuffle-kstreams
  minReplicas: 1
```

```
  maxReplicas: 10
  behavior:
    scaleUp:
#      stabilizationWindowSeconds: 30
      policies:
        - type: Pods
          periodSeconds: 60
          value: 2
    scaleDown:
      stabilizationWindowSeconds: 60
      policies:
        - type: Pods
          periodSeconds: 15
          value: 3
  metrics:
    - type: External
      external:
        metric:
          name: "kafka_input_lag"
          selector:
            matchLabels:
              topic: input
              consumergroup: shufflebench-kstreams
        target:
          type: Value
          value: 3000k
    - type: External
      external:
        metric:
          name: "consumer_cpu_utilization_30s"
#          selector:
#            matchLabels:
#              topic: input
        target:
          type: Value
          value: 80
```