

## Supporting Elasticity in Flink

---

Panagiotis Giannakopoulos

Committee:

Professor Euripides G.M. Petrakis (Supervisor)

Professor Michalis Zervakis

Associate Professor Antonios Deligiannakis

Chania, October 2020

---

## Abstract

Apache Flink is an open source framework that supports high-throughput, low latency data processing, as well as event processing. Flink executes arbitrary dataflow programs in a data-parallel and pipelined manner. At a basic level, Flink programs consist of streams and operators which apply transformations on data. The number of operator subtasks correspond to the parallelism of that particular operator and regulates the allocated resources for the execution of the operator. As a result, it is possible to set the desirable overall parallelism of the program by adjusting accordingly the parallelism of each operator. Although, Flink currently lacks the ability to automatically adjust to the resource needs of a running program and therefore it cannot adapt the program to varying workload. Therefore, such an adjustment can only be done with human intervention. The lack of dynamic resource allocation could lead to performance drop or to allocated resources remaining unused for long time (in the case of over or under utilization of resources respectively). In order to address this issue, we propose a statistical machine learning methodology which is implemented as a software agent that runs in parallel with Flink. The agent monitors the running program and adjusts the allocated resources to the incoming workload. The agent acts proactively by predicting the forthcoming workload in order to maintain the performance of the application within acceptable limits (i.e. defined in the form of SLAs) while minimizing the utilization of resources. This is achieved by adjusting (i.e. scaling-up or down) the computational resources to the actual and future needs of the application. To do so, a statistical machine learning model is used with online training in order to approach an optimal policy for scaling. As a proof of concept, we designed and implemented an infrastructure on the cloud which assess the efficiency of such scaling method in a Flink cluster. We run an exhaustive set of experiments using synthetic and real workloads available on the internet. The experimental results are a good support to our claims of efficiency.

---

## Contents

1. INTRODUCTION.....	5
1.1. Technology environment and applications.....	5
1.2. Problem description and needs .....	5
1.3. Solution and prospects.....	6
1.4. Contributions .....	7
1.5. Thesis structure.....	8
2. BACKGROUND.....	9
2.1. User cases .....	9
2.2. Cloud Computing .....	10
2.3. Apache Flink .....	10
2.4. Prometheus.....	18
2.5. Apache Kafka.....	19
2.6. Apache Zookeeper.....	22
2.7. Dynamic Scaling .....	23
2.8. Faban .....	25
3. AUTOSCALER FOR FLINK .....	29
3.1. System requirements .....	29
3.2. Reactive Scaler .....	31
3.3. Proactive Scaler .....	35
4. IMPLEMENTATION .....	49
4.1. Flink deployment options .....	49
4.2. Architecture .....	51
4.3. Components.....	53
5. EXPERIMENTS.....	62

---

5.1. Application: Clicks Fraud Detection .....	62
5.2. Experiment: Exploration Mode .....	63
5.3. Experiment: Optimal Control .....	67
5.4. Experiment: Synthetic Workload Distribution .....	70
5.5. Experiment: Autoscaler Load .....	76
6. CONCLUSIONS .....	79
7. FUTURE WORK .....	80
8. REFERENCES .....	81

## 1. Introduction

### 1.1. Technology environment and applications

Apache Flink<sup>1</sup> is a distributed framework for stream processing. It can be deployed in the cloud and execute programs in a parallel and distributed way. Flink provides an extensive toolbox of operators for implementing transformations on data streams (e.g., filtering, updating state, defining windows, aggregating). The Flink cluster consists of a manager and a number of workers. The manager regulates both the entire cluster and the operation of running applications. On the other hand, workers are the resources of the cluster which run the applications. The parallelism of such application is translated to the amount of resources allocated to that application. Consequently, Flink can be used to develop complex data stream programs that respond to high-throughput and low-latency specifications.

Applications deployed in the cloud, are able to provide high demand services. Such a service is stream processing where data are continuously generated by various sources. When streaming data are generated at high speed, traditional data-processing application software is no longer able to process the data within the acceptable time limit. The data processing speed can be dramatically improved by taking advantage of parallel execution environments where data are distributed across multiple servers.

### 1.2. Problem description and needs

The relation of service provider and service client is not arbitrary but shaped by a contract, referred to as service-level agreement (SLA). The type of service, its aspects (like quality or availability) and how they are measured, are agreed and formulated in SLA (e.g. 95% of the requests per seconds must be responded in less than 1 second or, service availability must be above 95% of the time of operation). In addition, the contract includes both obligations and penalties in case of non-compliance with the agreement. As an example, internet service providers will commonly include service level agreements within the terms of their contracts with customers to define the level of service being provided.

The quality of service is strongly related to the demand from clients as well as the allocated resources to that particular service. It is common knowledge that the workload of an application is not static but it can change throughout time, creating the need for dynamic configuration of the computing resources. Providers has to

---

<sup>1</sup> <https://flink.apache.org>

---

constantly adjust the allocated resources per service in response to application demands in order to maintain the quality of service. Considering that the target service is a website, the resources can be modeled as the number of machines serving requests. In case of increased workload, if application resources are not scaled up in response to applications needs, the targeted application ends up overloaded and under-served (i.e. the application is exhausting its computational resources, requests are slowed down or some requests cannot be served at all). Therefore, high workload in combination with inadequate number of allocated servers cause performance drop. The issue could be easily solved by having a relatively large number of servers permanently allocated to the application. Although the performance issue will be settled, the infrastructure will end up underutilized. Since allocated resources come at a price, maintaining dormant resources will make the service unsustainable in the long run.

In order to deploy services which are both sustainable and performant, a variety of distributed frameworks have developed resource adjusting agents. The agents maintain the quality of service with as few allocated resources as possible. In a cloud environment, the problem can be solved by horizontal or vertical scaling<sup>2</sup>. For example, Kubernetes<sup>3</sup> which is a container orchestrator, offers a mechanism for adjusting the number of active containers according the application's load. However, Flink cannot be easily configured to support vertical or horizontal scaling solutions such as the above.

### 1.3. Solution and prospects

In this thesis, we propose an autonomous agent for automatically adjusting resources used by Flink applications. The abstract architecture of the system is depicted in Figure 1. The agent monitors the targeted application and modifies the allocated resources accordingly. To do so, dynamic scaling mechanisms are implemented. The first one, which is the reactive, makes a decision after the performance falls under the acceptable limit. The reactive scaling is easy to implement but it leaves room for both over-utilization and under-utilization of resources. The second option is the proactive scaling, which analyzes previous measurements of performance, resources and workload, and is capable of predicting possible SLA violations. Proactive scaling makes optimal decisions but it comes with a complex implementation and requires a pre-existing dataset for the model training.

Both solutions have their pros and cons. In this work, we take advantage of both mechanisms and combine them into one integrated agent. At the beginning, the

---

<sup>2</sup> <https://en.wikipedia.org/wiki/Scalability>

<sup>3</sup> <https://kubernetes.io/>

agent explores the performance of the application under varying workload and different number of workers (e.g. servers). Specifically, the reactive scaler collects workload and performance information and adjusts (scales-up or down) the number of workers based on the actual workload and as soon as the SLA is violated. At the same time, a statistical machine learning model is trained that captures the relation between workload, resources and performance. Afterwards, the proactive scaler takes over the resource management and the agent's actions are based on the performance model. In this phase, the agent adjusts more efficiently the resource utilization and SLA violations are prevented. It is worth mentioning that modifications to the application's environment (e.g. hardware failure) or in the application itself (e.g. support of a new feature) can lead to change of its overall behavior. For example, extending an existing operation of the application would increase the required computational resources of that particular operation and thus affect the performance per request. In such a case, the existing machine learning model is not capable to describe the targeted application anymore, which in turn, could lead to inaccurate scaling actions. Such change detection forces the agent to return in reactive mode and start the collection of new data. The integration of both mechanisms in the overall architecture, results to self-adapting agent which maximizes the resource utilization and minimizes the SLA violations.

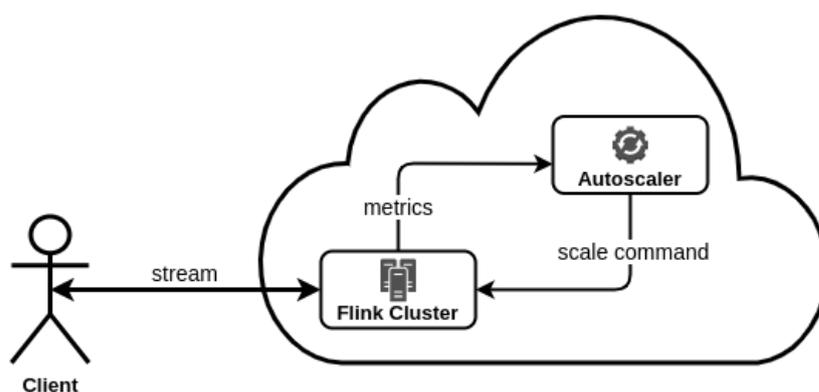


Figure 1 - Abstract architecture

## 1.4. Contributions

The main idea of the statistical machine learning approach for the scaling of resources has been described initially in [1] and [2]. These publications describe a controller which employs a performance model of the system to make decisions about the optimal allocation of resources for Web 2.0 applications deployed on cloud. However, the present work shows how the original idea can be exploited to support scaling decisions for Flink proactively rather than reactively. To the best our

---

knowledge, existing approaches (e.g. Ververica's Autopilot<sup>4</sup>) are mainly reactive and do not support proactive scaling. In addition, the scaling mechanism has been implemented using a statistical machine learning model and has been fully integrated within Flink. As a result, the new proactive Flink infrastructure can work for any application and any workload. This required that Flink jobs be constantly monitored using state-of-the-art tools (e.g. Prometheus) in order to take advantage of multiple systems measurements (e.g. queue length of unprocessed events referred to as "slow-events", throughputs referred to as number of workers or nodes, and resource utilization per worker).

The autoscaler decides when resource allocation modification has to be applied to the running application. We utilize modern low-footprint virtualization technologies based on Docker containerization. The actual and future need for Flink workers (i.e. TaskManagers) are computed either reactively (during exploration mode) or proactively (during optimal control). To be more specific, when there is a need to deploy more Flink workers to the cluster, the required resources are predicted by the model before they are allocated to Flink. Similarly, when a considerable part of workers remains idle, they are removed and the resources are deprovisioned. Consequently, the application provider is charged only for necessary resources.

## 1.5. Thesis structure

- Chapter 2 provides the knowledge background required for understanding this work and presents the software tools that are used for the completion of this thesis.
- Chapter 3 analyzes the components of the controller and how they interact with each other.
- Chapter 4 describes in detail the architecture of the infrastructure which extends Flink and enable stream processing with dynamic scaling.
- Chapter 5 evaluates the performance of the controller through experiments.
- Finally, Chapter 6 summarizes the conclusions and Chapter 7 offers recommendations for future work.

---

<sup>4</sup> <https://www.ververica.com/blog/introducing-ververica-platform-2.2-with-autoscaling-for-apache-flink>

---

## 2. Background

### 2.1. User cases

Stream processing enables a variety of brand-new applications characterized by increased data generation and the low latency response. Stream data could have the form of GPS signals, financial transactions, communication signals, web traffic or measurements from sensors. The efficient analysis of such data could lead to critical-mission applications. Some types of such applications are listed below:

#### Fraud detection

The use of data is spreading more and more in every sector of the society. It is no surprise that data can be altered in order to bypass the targeted system. Fraud detection undertakes the evaluation and identification of such data by marking them as a fraud. This can be done by training a model based on historical data which is able to identify patterns forming different types of fraud. For instance, a bank system could apply fraud detection for assessment of credit card transactions.

#### Anomaly detection

In various domains such as statistics, signal processing and finance, data follows a particular distribution but rare events or observations could occur. Anomaly detection is the identification of such cases which raise suspicions by differing significantly from the majority of the data. For example, in a health monitoring system, an anomaly in a patient's data could be translated to a medical problem.

#### Rule-based alerting

In this user case the goal is to identify data which satisfy one or more rules. When such an event is captured, an alert is raised. Such an application could be a smart home system which receives several sensor readings from the rooms of a house. A flag for fire alarm could be generated when air-associated data satisfy specified conditions.

#### Quality monitoring

A deployed application could generate real-time metrics consisting a data stream. The processing of the data stream produces a continuously updated report about the quality of service. Such a report contains valuable information and indicators about the targeted service. The service provider could utilize that report to assess the quality of service and identify possible improvement points

## 2.2. Cloud Computing

Cloud computing provides on-demand availability to computer resources, such as computation power and data storage. Cloud providers follow a "pay-as-you-go" policy, which means that clients are charged only for the allocated resources. The datacenter could be physically geographically distributed but be presented to the client of the system as a single entity. In this way, each client is able to request the preferable resources that fits the hosted application better. Resources in cloud are scalable, they can be easily adjusted by removing or adding resources. The client can scale the allocated resources depending on the needed utilization, resulting to a cost-efficient plan.

Microservices architectural style arranges an application as a collection of loosely coupled services. The services can run on separate machines and through a communication protocol over a network form one single application. For example, the application could be deployed on the cloud and be distributed on separate virtual machines. The decomposition releases the application from the boundaries of a single machine and thus enables distributed and parallel applications. The parts of the application can be updated independently from each other and yet cause no disruption to overall operation the application.

## 2.3. Apache Flink

*The content of the following section is based on Apache Flink documentation<sup>5</sup>.*

Flink<sup>6</sup> is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams. Flink has been designed to run in all common cluster environments, perform computations at in-memory speed and at any scale.

### Stream Processing

Streams are data's natural habitat. Whether the events are generated from web servers, trades from a stock exchange, or sensor readings from a machine on a factory floor, data is created as part of a stream, as shown in Figure 2. Therefore, data analysis can be organized in two forms:

- Batch processing, where the data are processed as a bounded data stream. In this mode of operation, the entire dataset is ingested to Flink before producing any results. It is possible, for example, to sort the data, compute global statistics, or produce a final report that summarizes all of the input.

---

<sup>5</sup> <https://ci.apache.org/projects/flink/flink-docs-release-1.11>

<sup>6</sup> <https://flink.apache.org>

- Stream processing, on the other hand, involves unbounded data streams. Conceptually, the input may never end, and so the data has to be continuously processed as they arrive.

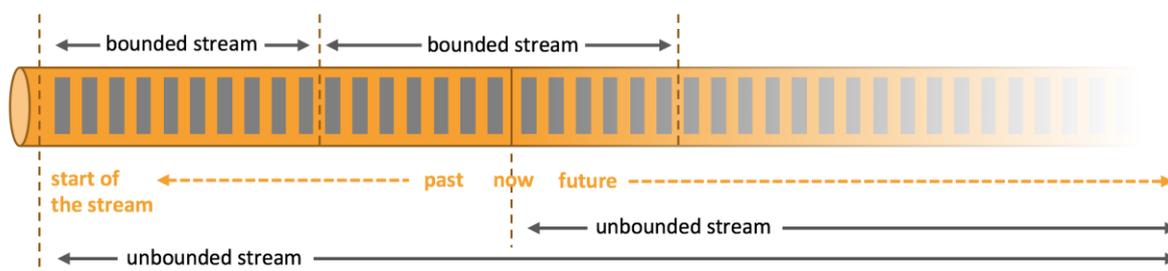


Figure 2 - Stream data

### Applications

In Flink, applications, referred as jobs, are composed of streaming data flows that may be transformed by user-defined operators. These dataflows form directed graphs, called JobGraph, that start with one or more sources, and end in one or more sinks. An example of a directed graph is shown in Figure 3. In the particular example, the data enter the dataflow through source, they are transformed by two operators and finally exit the dataflow through the sink. Last but not least, Flink offers different levels of abstraction for developing streaming/batch applications (e.g. SQL API).

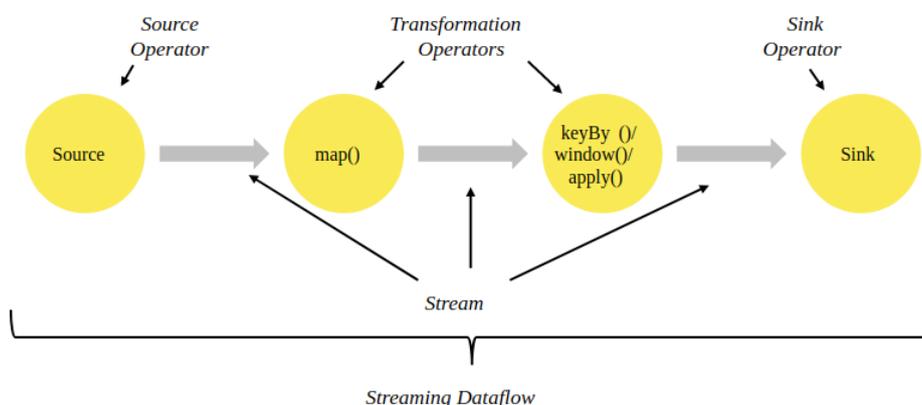


Figure 3 - Directed graph of Flink application

An application may consume real-time data from streaming sources such as message queues or distributed logs, like Apache Kafka<sup>7</sup> or Kinesis<sup>8</sup>. But Flink can also

<sup>7</sup> <https://kafka.apache.org/>

<sup>8</sup> <https://aws.amazon.com/kinesis/data-streams/>

consume bounded, historic data from a variety of data sources. Similarly, the streams of results being produced by a Flink application can be sent to a wide variety of systems that can be connected as sinks. An abstract diagram of several connectors types for source or sink operators are depicted in Figure 4.

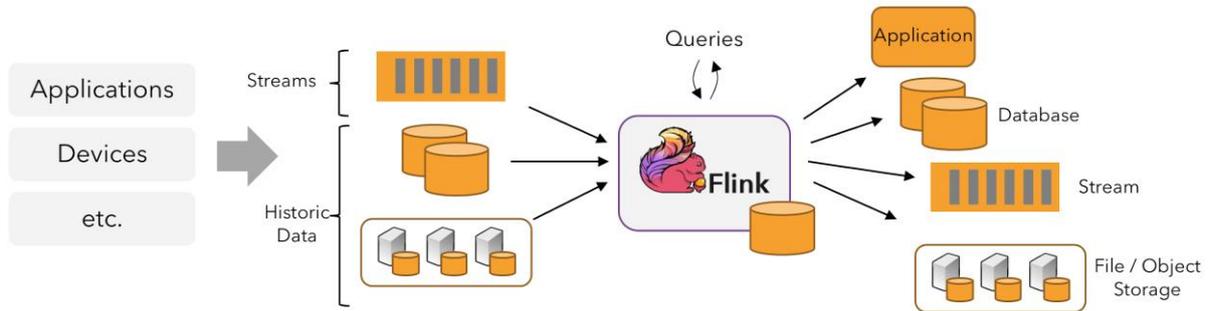


Figure 4 - Flink connectors

Programs in Flink are inherently parallel and distributed. During execution, a stream has one or more stream partitions, and each operator has one or more operator subtasks. The operator subtasks are independent of one another, and execute in different threads and possibly on different machines or containers. The number of operator subtasks is the parallelism of that particular operator. Different operators of the same program may have different levels of parallelism. For example, in Figure 5, all operators have parallelism 2, except Sink which has parallelism 1. The load of each operation is evenly balanced between its subtasks.

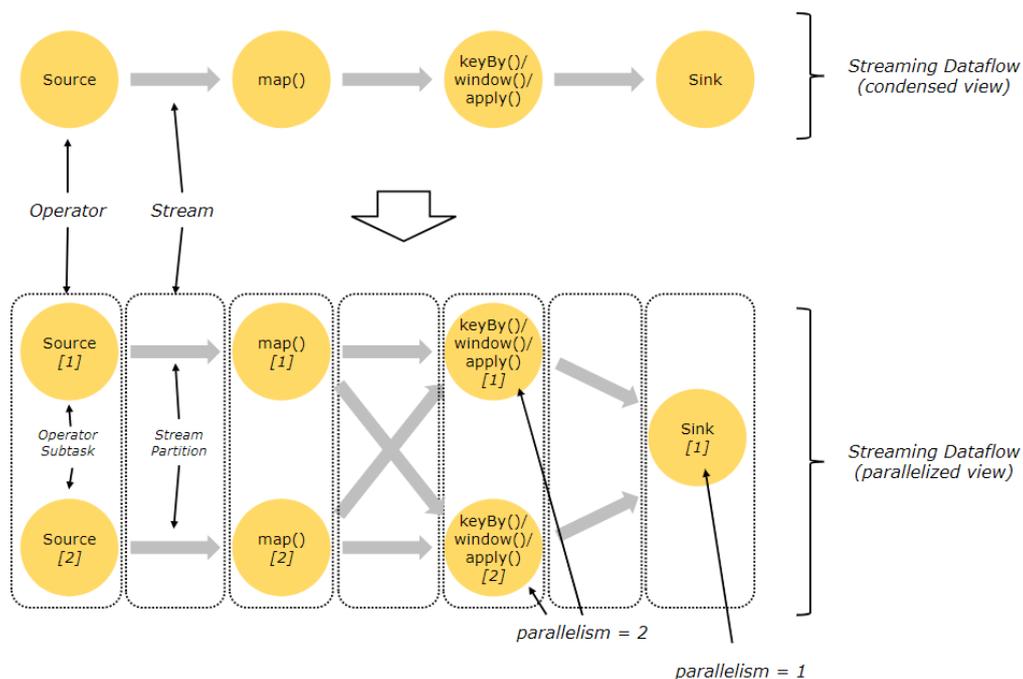


Figure 5 - Flink parallel dataflow

## Architecture

The Flink runtime consists of two types of processes: a JobManager and one or more TaskManagers. The abstract architecture is depicted in Figure 6.

### JobManager

The JobManager is responsible for the coordination of the distributed execution of Flink Applications: it decides when to schedule the next task (or set of tasks), reacts to finished tasks or execution failures, coordinates checkpoints, and coordinates recovery on failures, among others. This process consists of three different components:

- The **ResourceManager**, which is responsible for resource de-/allocation and provisioning in a Flink cluster — it manages task slots, which are the unit of resource scheduling in a Flink cluster.
- The **Dispatcher**, which provides a REST interface to submit Flink applications for execution and starts a new JobMaster for each submitted job. It also runs the Flink WebUI to provide information about job executions.
- A **JobMaster**, which is responsible for managing the execution of a single JobGraph. Multiple jobs can run simultaneously in a Flink cluster, each having its own JobMaster.

There is always at least one JobManager. A high-availability setup might have multiple JobManagers, one of which is always the leader, and the others are in standby mode.

### TaskManagers

The TaskManagers (also called workers) execute the tasks of a dataflow and exchange the data streams.

There must always be at least one TaskManager. The smallest unit of resource scheduling in a TaskManager is a task slot. The number of task slots in a TaskManager indicates the number of concurrent processing tasks. Multiple operators may execute in a task slot.

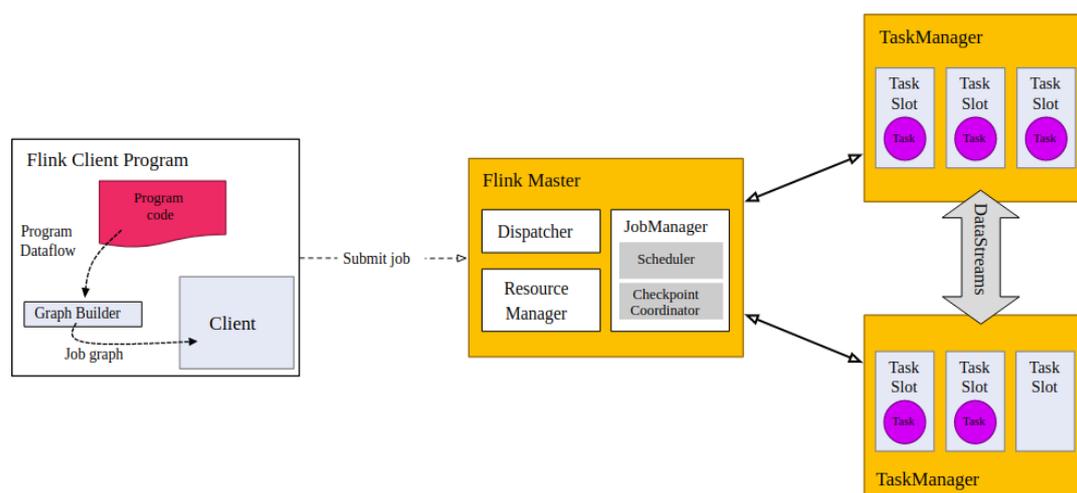


Figure 6 - Flink runtime

## Stateful Stream Processing

Flink's operations can be stateful. This means that how one event is handled can depend on the accumulated effect of all the events that came before it. State may be used for something simple, such as counting events per minute to display on a dashboard, or for something more complex, such as computing features for a fraud detection model.

### Fault Tolerance via State Snapshots

Flink is able to provide fault-tolerant, exactly-once semantics through a combination of state snapshots and stream replay. The term 'exactly-once' means there is no data loss or duplicates but each data record is processed exactly once. Data records are distinguished from each other on the basis of their offsets, which are positional numbers of the records in the data stream. Using offsets, we are able to track up to which point the data stream is consumed. As a result, lost data records can be recovered with stream replay, in which data records are re-consumed.

The snapshots capture the entire state of the distributed pipeline, recording offsets into the input queues as well as the state throughout the job graph. When a failure occurs, the sources are rewound, the state is restored, and processing is resumed. These state snapshots are captured asynchronously, without impeding the ongoing processing.

There are two main types of snapshots:

1. **Checkpoint:** a snapshot taken automatically by Flink for the purpose of being able to recover from faults. Checkpoints can be incremental, and are optimized for being restored quickly.
2. **Savepoint:** a snapshot triggered manually by a user (or an API call) for some operational purpose, such as a stateful redeploy/upgrade/rescaling operation.

## High Availability

The general idea of JobManager high availability is that there is a single leading JobManager at any time and multiple standby JobManagers to take over leadership in case the leader fails. This guarantees that there is no single point of failure and programs can make progress as soon as a standby JobManager has taken leadership. There is no explicit distinction between standby and master JobManager instances. Each JobManager can take the role of master or standby.

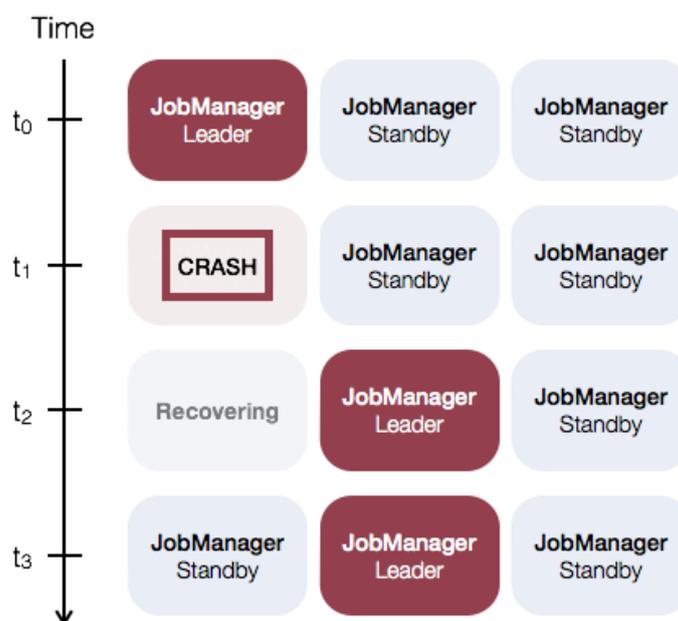


Figure 7 - Flink High Availability

An example with three JobManager instances is shown in Figure 7. The JobManager High Availability mode is enabled by using Zookeeper<sup>9</sup>. Flink utilizes Zookeeper for distributed coordination between all running JobManager instances. Zookeeper is a separate service from Flink, which provides highly reliable distributed coordination via leader election and light-weight consistent state storage. Zookeeper is described in Section 2.6.

## Cluster modes

Flink has three main modes for clustering. The Flink Cluster consists of at least one JobManager and one or more TaskManagers. The cluster mode regulates whether Flink is dedicated to only one job or can host multiple jobs. Also, each option offers different lifecycle of cluster and resource isolation guarantees.

---

<sup>9</sup> <https://zookeeper.apache.org/>

### Session Mode

Session mode assumes an already running cluster and uses the resources of that cluster to execute any submitted application. Applications executed in the same (session) cluster use, and consequently compete for, the same resources. This has the advantage that there is no resource overhead of spinning up a full cluster for every submitted job (e.g. each cluster has its one JobManager). But, if one of the jobs misbehaves or brings down a TaskManager, then all jobs running on that TaskManager will be affected by the failure. Additionally, having a single cluster running multiple jobs implies more load for the JobManager, which is responsible for the book-keeping of all the jobs in the cluster.

### Per-Job Mode

Aiming at providing better resource isolation guarantees, the Per-Job mode uses the available cluster manager framework (e.g. Kubernetes) to spin up a cluster for each submitted job. This cluster is available to that job only. When the job finishes, the cluster is shut down and any lingering resources (e.g. files) are cleared up. This provides better resource isolation, as a misbehaving job can only bring down its own TaskManagers. In addition, it spreads the load of book-keeping across multiple JobManagers, as there is one per job.

### Application Mode

In all previous modes, the application's main() method is executed on the client side. This process includes downloading the application's dependencies locally, build the job and then submitted to the cluster. This makes the client a heavy resource consumer as it may need substantial network bandwidth to download dependencies and upload binaries to the cluster. Building on this observation, the Application Mode creates a cluster per submitted application, but this time, the main() method of the application is executed on the JobManager. Creating a cluster per application can be seen as creating a session cluster shared only among the jobs of a particular application, and shut down when the application finishes. With this architecture, the Application Mode provides the same resource isolation and load balancing guarantees as the Per-Job mode, but at the granularity of a whole application. Executing the main() on the JobManager allows for saving the CPU cycles required for the build of the job, but also save the bandwidth required for downloading the dependencies locally.

## Why Apache Flink

Although other frameworks, like Spark<sup>10</sup>, Storm<sup>11</sup>, Samza<sup>12</sup> and Apex<sup>13</sup>, already provide stream processing, Flink is gaining more and more ground. Developing a stream processing engine could be quite challenging due to the nature of such data. Flink stood out from the rest of the frameworks by overcoming challenges and limitations which were left unresolved from the other platforms. Some of the major benefits of Flink are the following:

1. **Native Streaming:** Flink offers true streaming processing, which means records are consumed immediately upon their arrival. Another approach for stream processing is micro-batching, which processes records in small batches. In this case, the processing takes place after the required number of records is collected resulting to additional latency.
2. **Exactly-once guarantee:** records are processed exactly once. Even in case of a machine or software failure, there is no duplicate data or data loss. On the other hand, an at-least-once system has no data loss but could process some data records more than one times and thus affecting the final output.
3. **Performance:** a streaming engine attempts to minimize the data latency (the total time needed for the processing of each data record) and maximize the total number of processed records per second. To put it in another way, both low-latency and high-throughput define an effective framework.
4. **Stateful operations:** operators which require the ability to maintain a state. For instance, a counter updates its value by incrementing the previous one.
5. **Advanced operations:** Flink offers a variety of built-in features which make the programming easier. Such features are event time processing, aggregation and watermarks. In addition, Flink includes machine learning and graph API that developers can easily integrate within their streaming applications.
6. **Unified framework:** Flink provides different abstraction APIs for both batch and stream processing.

Therefore, Flink combines the traits of the pre-existed frameworks and extend them even further in one technology.

---

<sup>10</sup> <https://spark.apache.org/>

<sup>11</sup> <https://storm.apache.org/>

<sup>12</sup> <http://samza.apache.org/>

<sup>13</sup> <https://apex.apache.org/>

## 2.4. Prometheus

*The content of the following section is based on Prometheus documentation<sup>14</sup>.*

Prometheus<sup>15</sup> is an open-source platform used for event monitoring. The monitored targets run specified agents which export metrics (e.g. CPU load or memory usage) through HTTP endpoints. The agents can be created by using client libraries for the supported programming languages or be user custom programs which implement the required exposition formats (the format of the exported metrics). Prometheus collects data, in the form of time series<sup>16</sup>, from all targets into a centralized server by querying the HTTP endpoints on these targets at a specific polling frequency.

Every time series is uniquely identified by its metric name and optional key-value pairs called labels. Labels can include information on the data source (which server the data is coming from) and other application-specific breakdown information such as the HTTP status code, query method (GET or POST), etc. Labels enable Prometheus's dimensional data model: any given combination of labels for the same metric name identifies a particular dimensional instantiation of that metric. For example, all HTTP requests that use the method POST to the `/api/tracks` handler. In this case, the HTTP request consists the metric name while the method and handler are the labels. Fundamentally all data are stored as time series: streams of timestamped values belonging to the same metric and the same set of labeled dimensions. Besides stored time series, Prometheus may generate temporary derived time series as the result of queries (e.g. total number of GET requests of the last one hour).

In Prometheus terms, an endpoint which can be scraped (requests can be made on it), is called an instance, usually corresponding to a single process. A collection of instances with the same purpose, a process replicated for scalability or reliability for example, is called a job. For example, an API server job with four replicated instances could have the following form:

```
job: api-server
  a. instance 1: 1.2.3.4:5670
  b. instance 2: 1.2.3.4:5671
  c. instance 3: 5.6.7.8:5670
  d. instance 4: 5.6.7.8:5671
```

The monitoring targets can be configured at the beginning of the operation of Prometheus and be modified dynamically afterwards. In additions, Prometheus provides a functional query language called PromQL (Prometheus Query Language) that lets the user select and aggregate time series data, based on label dimensions in

---

<sup>14</sup> <https://prometheus.io/docs/introduction/overview/>

<sup>15</sup> <https://prometheus.io/>

<sup>16</sup> [https://en.wikipedia.org/wiki/Time\\_series\\_database](https://en.wikipedia.org/wiki/Time_series_database)

real time. Changing any label value, including adding or removing a label, will create a new time series. The result of an expression can either be shown as a graph, viewed as tabular data in Prometheus's expression browser, or consumed by external systems via the HTTP API.

## 2.5. Apache Kafka

*The content of the following section is based on Apache Kafka documentation<sup>17</sup>.*

Apache Kafka<sup>18</sup> is an open-source distributed event streaming platform used for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications.

Event streaming is the practice of capturing data in real-time from event sources like databases, sensors, mobile devices, cloud services, and software applications in the form of streams of events; storing these event streams durably for later retrieval; manipulating, processing, and reacting to the event streams in real-time as well as retrospectively; and routing the event streams to different destination technologies as needed. Event streaming thus ensures a continuous flow and interpretation of data so that the right information is at the right place, at the right time.

Kafka supports three key capabilities for event streaming end-to-end with a single battle-tested solution:

1. To publish (write) and subscribe to (read) streams of events, including continuous import/export of data from other systems.
2. To store streams of events durably and reliably.
3. To process streams of events as they occur or retrospectively.

Kafka is a distributed system consisting of servers and clients that communicate via a high-performance TCP network protocol. It can be deployed on bare-metal hardware, virtual machines, and containers in on-premise as well as cloud environments. Kafka runs as a cluster of one or more servers that can span multiple data centers or cloud regions. The clients allow the development of distributed applications and microservices that read, write, and process streams of events in parallel, at scale, and in a fault-tolerant manner even in the case of network problems or machine failures.

---

<sup>17</sup> <https://kafka.apache.org/documentation/>

<sup>18</sup> <https://kafka.apache.org/>

## Events

An event records the fact that "something happened". An event has a key, value, timestamp, and optional metadata headers.

- Event key: "Alice"
- Event value: "Made a payment of \$200 to Bob"
- Event timestamp: "Jun. 25, 2020 at 2:06 p.m."

Producers are those client applications that publish (write) events to Kafka, and consumers are those that subscribe to (read and process) these events. In Kafka, producers and consumers are fully decoupled and agnostic of each other, which is a key design element to achieve the high scalability. For example, producers never need to wait for consumers.

Events are organized and durably stored in topics. A topic is similar to a folder in a filesystem, and the events are the files in that folder. Events in a topic can be read as often as needed - unlike traditional messaging systems, events are not deleted after consumption. Instead, a policy can be defined for how long Kafka should retain events through a per-topic configuration setting, after which old events will be discarded.

## Partitions

Topics are partitioned, meaning a topic is spread over a number of "buckets" located on different Kafka brokers. The structure of a topic is depicted in Figure 8. This distributed placement of the data is very important for scalability because it allows client applications to both read and write the data from/to many brokers at the same time. When a new event is published to a topic, it is actually appended to one of the topic's partitions. Events with the same event key (e.g., a customer or vehicle ID) are written to the same partition, and Kafka guarantees that any consumer of a given topic-partition will always read that partition's events in exactly the same order as they were written.

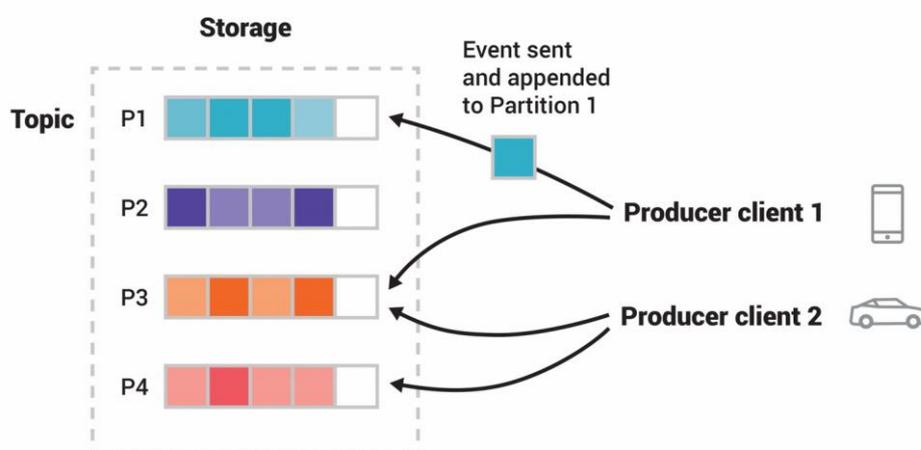


Figure 8 – Structure of Kafka topic

To make the data fault-tolerant and highly-available, every topic can be replicated, even across geo-regions or data centers, so that there are always multiple brokers that have a copy of the data just in case of failure or maintenance. Under non-failure conditions, each partition in Kafka has a single leader and zero or more followers. There is no explicit distinction between leader and followers' instances. The total number of replicas including the leader constitute the replication factor. All reads and writes go to the leader of the partition. The logs on the followers are identical to the leader's log which means that they all have the same messages in the same order. Followers consume messages from the leader just as a normal Kafka consumer would and apply them to their own log. If the leader crashes, the followers elect a new leader which continues the operation of the particular partition. The leader election functionality is provided by Zookeeper.

### **Kafka APIs**

Kafka has five core APIs:

- The Admin API to manage and inspect topics, brokers, and other Kafka objects. For example, create/delete topics or list consumers connected to the cluster.
- The Producer API to publish (write) a stream of events to one or more Kafka topics.
- The Consumer API to subscribe to (read) one or more topics and to process the stream of events produced to them.
- The Kafka Streams API is used to implement stream processing applications and microservices. It provides higher-level functions to process event streams, including transformations, stateful operations like aggregations and joins. Input is read from one or more topics in order to generate output to one or more topics, effectively transforming the input streams to output streams.
- The Kafka Connect API to build and run reusable data connectors that consume (read) or produce (write) streams of events from and to external systems and applications so they can integrate with Kafka. Compared to Streams API which read/write data from/to Kafka topics, the Connect API offers connectivity (import/export of data) between Kafka and other systems. For example, a connector to a relational database might capture every change to a table.

## 2.6. Apache Zookeeper

The content of the following section is based on Apache Zookeeper documentation<sup>19</sup>.

Apache ZooKeeper<sup>20</sup> is a distributed, open-source coordination service for distributed applications. It exposes a simple set of primitives that distributed applications can build upon to implement higher level services for synchronization, configuration service, and naming registry.

Operations like configuration management in distributed systems could be quite challenging due to the distributed nature of the system. The state of the system is composed of each individual state of the distributed parts, which increase the complexity of maintaining a valid state. A modification to one part has to be broadcasted to the rest of the system, which could be quite complex in large applications. In addition, synchronization operations (like leader election<sup>21</sup>) or maintaining a naming registry (discovery among the distributed parts), are hard to build from scratch. The aforementioned issues could be solved by introducing a centralized service which coordinates the operations. Zookeeper offers a high-performance coordination service where developers can build upon their distributed applications.

To do so, Zookeeper allows distributed processes to coordinate with each other through a shared hierarchical namespace which is organized similarly to a standard file system. The namespace consists of data registers - called znodes, which are similar to files and directories. The namespace is depicted in Figure 9. Unlike a typical file system, which is designed for storage, data is kept in-memory, achieving high throughput and low latency. Znodes maintain timestamps and version numbers for data changes. Clients can read from and write to the nodes and in this way have a shared configuration service.

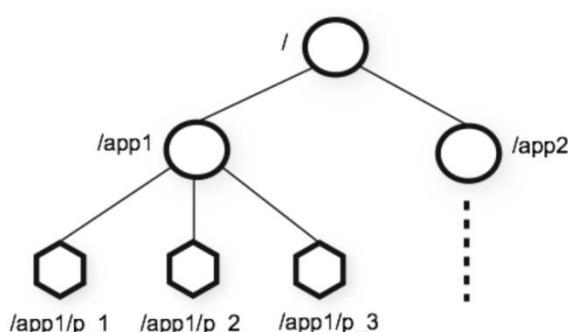


Figure 9 – Zookeeper's namespace

<sup>19</sup> <https://zookeeper.apache.org/doc/r3.6.2/index.html>

<sup>20</sup> <https://zookeeper.apache.org/>

<sup>21</sup> [https://en.wikipedia.org/wiki/Leader\\_election](https://en.wikipedia.org/wiki/Leader_election)

---

The components of distributed system connect to Zookeeper through clients and coordinate with each other through znodes. The clients can create/delete znodes or read/write data. Znodes can be used to store coordination data of the system (e.g. status information, configuration and location information). In this way, data updates are instantly available to all clients. In addition, there is the notion of ephemeral nodes, which exist as long as the client's session, that created the znode, is active. When the session ends the znode is deleted, which in turn, indicates that a client disconnected (e.g. crashed). A client can be notified of modifications in particular znodes and thus be aware of the activity of other clients. Consequently, znodes can be used to share information among a distributed system and provide notifications for the status of each component.

Last but not least, properties of znodes can be utilized to perform certain functions. For example, in a leader election, clients create child znodes under the path "/election" that represent "proposals" and Zookeeper selects as leader the client that created the znode with smallest sequence number (e.g. znode which created first).

## 2.7. Dynamic Scaling

### 2.7.1. Proactive Scaling

The mechanism for proactive scaling is based on future predictions related to targeted application and thus attempts to maintain the performance by acting in time. Typically, a proactive mechanism utilizes machine learning methods in order to generate a model which describes the behavior of the application. The model captures the relation between performance, workload and used resources. In order to do so, a dataset with historical data of the application is required for the training of the model. A proactive scaler will try to predict and then prevent a harmful event, such as SLA violation, based the incoming workload and act before it actually takes place. For example, if the application is a video streaming platform and it is known that the usage of the platform is increased at Fridays, the proactive mechanism should increase the running servers ahead of that time period. As a result, the performance is maintained to the desirable levels. In addition, resources are released as soon as the incoming workload can be processed with few resources.

The training of the model could be performed offline or online. In the first case, model fitting takes place only once using the previously stored metrics of the applications. In the second case, the training dataset is updated by the production data so as to capture new characteristics in the behavior of the application. In comparison, the offline training results to a static but simple model while online is adjustable but more sophisticated.

## Related work

Bodík et al. [1,2] describe a proactive autoscaling mechanism for Web 2.0 applications which run on virtual machines (VMs). The autoscaler is implemented using statistical machine learning and it is trained online by production data. In our work, we examine the efficiency of such autoscaler in a streaming engine like Apache Flink which is deployed in a containerized environment.

Arabnejad et al. [3] compare two different autoscaling types of Reinforcement Learning (RL), which is SARSA and Q-learning. The autoscaler dynamically resize web applications, which run on virtual machines, in order to meet Quality of Service requirements. A fuzzy controller is implemented in order to reduce the total solutions which are generated by the state-action values of the RL algorithm.

Bibal Benifa and D. Dejeu [4] propose the RLPAS algorithm, which applies Reinforcement Learning (RL) through a neural network in order to reduce the time for convergence to an optimal policy. The autoscaler learns the environment and adjusts the allocated resources for Web applications deployed on VMs.

Rossi, Nardelli and Cardellini [5] propose Reinforcement Learning (RL) solutions for controlling the horizontal and vertical elasticity of container-based applications in order to cope with varying workloads. Although, RL solutions, like [3],[4] and [5], are based on a trial and error policy. As a result, they could lead to increased SLA violations. In addition, the particular autoscalers do not apply any change point detection method for adapting the model in changes of application's behavior.

### 2.7.2. Reactive Scaling

Another mechanism for scaling is the reactive one which makes a decision when one or more metrics of the system exceeds a specified threshold. For example, a new server will be added when the CPU usage is more than 90%. Therefore, this kind of mechanism will react after an event happens. Also, the reactive scaler releases allocated resources when they remain idle for a specific period of time, showing that they are no longer needed. The implementation of the reactive scaler is much simpler than the proactive but it tolerates more SLA violations.

## Related work

Baresi et al. [6] present an autoscaling technique that allows containerized applications to scale their resources both at the virtual machine level and at the container level. The autoscaling is based on a planner, which consists of a discrete-time feedback controller. The planner computes the required resources (e.g. CPU cores) that have to be available to each tier so as to maintain the response time below a certain threshold. Afterwards, the computed resource allocations are translated into adaptation actions.

DS2 [7] is an autoscaler that enables automatic scaling to applications which consists of dataflow operators (like Apache Flink). The controller assesses the running application in operator granularity and finds the bottleneck in the dataflow (which operator slows down the whole application). In contrast to this work, DS2 adjust the

parallelism of each operator separately in order to maintain the throughput in acceptable rate. Although, the scaler works reactively rather than proactively.

Elixir [8] is an autonomous agent for enabling autoscaling on Docker Swarm. The agent monitors the targeted applications and adjusts reactively the allocated resources when a metric of the system surpasses a specified threshold (e.g. CPU usage). The resource adjustment is performed by adding or removing Nodes to the Swarm.

Autopilot<sup>22</sup> is an autoscaler which was introduced to the latest release of Ververica Platform. The autoscaler performs dynamic resource adjustment reactively on Apache Flink applications deployed on Ververica Platform. The autoscaler allocates the minimum required resources so as the application continues to keep up with all of its sources, resulting to minimum latency. Autopilot supports applications with multiple sources, while in our work, the supported applications contain only one source.

## 2.8. Faban

*The content of the following section is based on Faban documentation<sup>23</sup>.*

Faban<sup>24</sup> is a framework for developing and running benchmarks for the assessment of a targeted system, referred as System Under Test (SUT). It has two major components, the Faban Driver Framework and the Faban Harness.

### Faban Harness

Faban Harness is a tool to automate the running of server benchmarks. It also serves as a container to host benchmarks allowing them to be deployed in a rapid manner. The Faban harness provides a web interface to launch, queue, view, compare and graph run outputs. The architecture is shown in Figure 10 and includes the following components:

1. **Master:** Contains a web server which facilitates access through the web interface to Faban harness. It provides user interfaces for submitting, managing runs as well as accessing the run results. The run queue is the main engine controlling the runs. The log server receives and stores log records from the master itself, agents, and possibly the SUT. The master may or may not act as a load driving agent by itself, depending on the configuration.
2. **Agents:** There are two types of agents. The driver agents who are responsible to drive the benchmark run (e.g. making requests to the targeted SUT). Also, there are the Faban command agents who are located on the system under test (SUT). The command agents are lightweight agents which act as a proxy

---

<sup>22</sup> <https://www.ververica.com/blog/introducing-ververica-platform-2.2-with-autoscaling-for-apache-flink>

<sup>23</sup> <http://faban.org/1.3/docs/index.html>

<sup>24</sup> <http://faban.org/>

for starting/stopping server processes as well as collecting relevant statistics from the system under test. Both types of agents are optional.

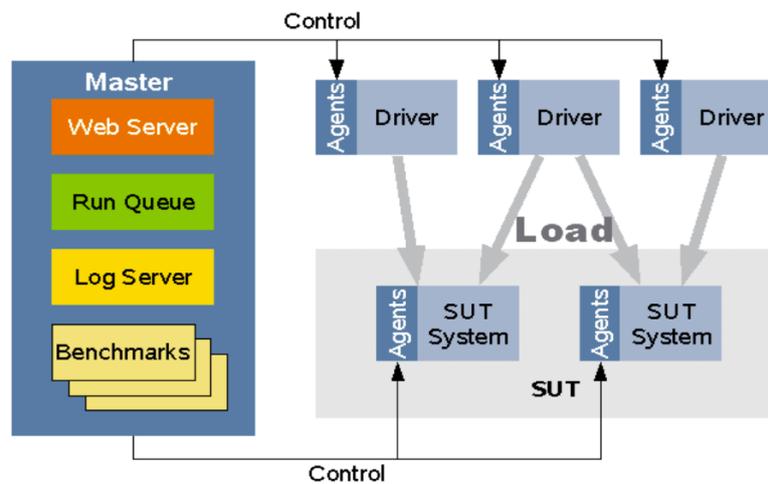


Figure 10 - Architecture of Faban Harness.

### Faban Driver Framework

Faban Driver Framework is an API-based framework that allows developers to easily define new benchmarks using the Java Programming Language. The driver framework controls the lifecycle of the benchmark run as well as the stochastic model used to simulate users. It provides built-in support for a variety of servers such as Apache httpd and a well-documented interface to add support for any other server (e.g. Apache Kafka). The Driver Frameworks architecture is depicted in Figure 11 and includes the following components:

- The **Registry** registers all the Agents so that the Master can find them and distribute the tasks to them. There is only one instance of the registry in a benchmark configuration.
- The **Master** starts, stops and collects the metrics for each of the benchmark runs. It also provides services to collect runtime metrics, do a benchmark health check, and to cancel the benchmark run prematurely. There is only one instance of the master in a benchmark configuration.
- The **Agent** is the process that actually drives the load. It will create threads as instructed by the master and drive the load for the length of time or number of iterations as instructed by the master. Each of the agent threads simulate a single client or user to the system under test (SUT). These threads will instantiate the developer-supplied driver, collect metrics, and aggregate and propagate them back to the master for final processing and reporting.
- The **Agent Thread** is created by the agent to simulate a single user executing an instance of the driver. The agent thread executes the workload and handles all timing functions and operation selections on the driver, and collects all the standard metrics.

- The **Driver** is a developer-supplied class that describes the workload and contains all the logic defining how to talk to the system under test (e.g. what operations can be made). It provides a grouping of all simulated user scenarios, each of them provided in the form of an operation. The selection of the scenarios is descriptively specified in the driver but actually controlled by the agent.
- The **Benchmark** is a grouping of one or more drivers. The benchmark result or metric is an aggregation of the driver metrics and are reported as one final metric.
- The standard **Metrics** object collects all the common statistics of a benchmark run. Faban provides an extension mechanism which allows a developer to specify custom metrics that may not be covered by the standard ones.

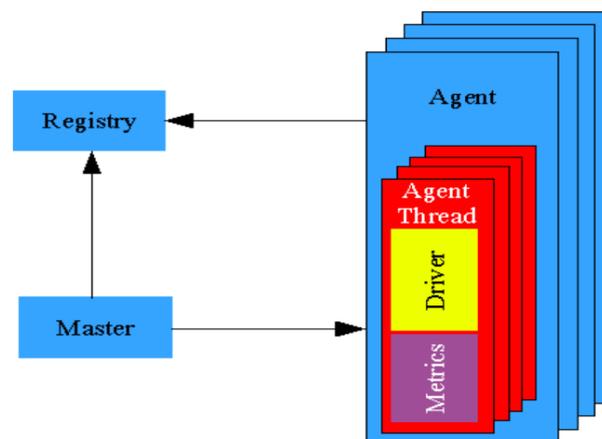


Figure 11 - Components of Faban Driver Framework

### Driver Class

The driver class represents a plain old Java object (POJO) which contains annotations defining the load model of the driver. The class is a description of the simulated user. It contains the possible operations that the user can perform on the System Under Test (e.g. visit the home page or contact page), their time characteristics (e.g. make a new request after 30 sec from the previous one) and how the operations are related to each other (e.g. visit the contact page if you currently are at the home page). The possible configuration terms are listed below:

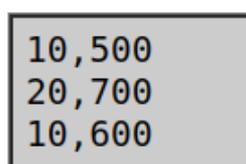
- **Operation**: a single unit of task executed by the user or the driver.
- **Operation Mix**: defines how operations are executed by a simulated user or driver thread.
- **Operation Cycle**: defines the timing characteristics from one operation to the next.
- **Metric**: is the resulting data of the operations collected over the steady state time of the benchmark operation.

- **Cycle Time:** defines the elapsed time between the begin of the previous operation to the begin of the current operation.
- **Think Time:** defines the elapsed time between the end of the previous operation to the begin of the current operation.
- **Critical Section:** defines the portion of an operation in which the response time is measured.
- **Data Preparation Section:** defines the portion of an operation before the critical section where the data gets prepared.
- **Validation Section:** defines the portion of an operation after the critical section where the results get validated.

### Configuring Runs with Varying Loads

The default workload of Faban is steady throughout the run, but it can be configured to follow a specific distribution. In this way, the system under test can be assessed in terms of elasticity, adaptivity or other limitations under varying workload. Faban provides load variation files in order to simulate such behavior and can be enabled through the configuration file of the run. In the beginning, all the necessary Agent Threads, for the entire workload, are being created. Each moment of time, only the needed Agent Threads, according the variation file, are active to drive the load. The rest of them remain idle until they are needed. Different load variation files can be defined for each agent.

The file contains load level records, one per line. Each record is a comma-separated pair of integers in the form <runtime in secs> , <thread count>. For example, in Figure 12, a load of 500 threads is applied for 10 seconds, then 700 threads start applying load for 20 seconds. Afterwards, 600 threads apply the last set of load for 10 seconds. Consequently, the variation file can describe any workload distribution including a synthetic one. The synthetic workload distribution is produced by analyzing the trace or log files of deployed services making possible to repeat the targeted real workload [9]. For example, we can measure the past workload of a system by looking into its log files and then count how many requests the system received per second. Afterwards, the workload is described in the variation file and each active thread simulates the user by making requests to the System Under Test. As a result, Faban can be used for the generation of benchmarks based on real workload distributions and thus for the evaluation of the targeted system's capabilities under realistic conditions.



```
10, 500
20, 700
10, 600
```

Figure 12 - Example file for generating varying workload in Faban.

---

## 3. Autoscaler for Flink

### 3.1. System requirements

The autoscaler can be integrated in any system which supports the following functionalities, as shown in Figure 13:

1. **Resource scaling**: The available resources of the system has to be dynamically configured. For example, in containerized environment (e.g. Docker, Kubernetes), the orchestrator creates or removes containers on the fly. Similarly, in a cloud environment (e.g. OpenStack<sup>25</sup>) the available virtual machines are allocated/deallocated dynamically. Afterwards, the available resources have to be de/allocated to a specific running application as servers.
2. **Scheduler**: The scheduler distributes the available resources to applications and provides load balancing<sup>26</sup>. The system has to be able to evenly distribute the incoming workload of the running application among its allocated servers. In other words, all servers handle the same number of requests (considering that each request demands the same amount of computational power). This is essential for the optimal utilization of resources. If the load is not evenly distributed, some servers will be stressed more than others, resulting to performance drop to requests served by the stressed servers. The even distribution of workload ensures the performance will drop only if all allocated servers are exhausted (none of the served requests satisfies the SLA).
3. **Monitoring**: The components of the system have to be constantly monitored in order to extract the required metrics for the decision making. Such metrics are:
  - a. **Number of allocated servers**: the allocated resources which serve requests.
  - b. **Workload**: The number of requests per second which has to be served by the application
  - c. **Performance**: The amount of time needed to serve a request. The performance can be measured by the average latency per request, which represent the amount of time needed by the server to generate a response. Another option is to measure the number of requests which are not yet processed by any server but remain in a queue.
4. **Endpoint**: The point which the client interacts with the running application. In stream processing such endpoint could be Kafka.

---

<sup>25</sup> <https://www.openstack.org/>

<sup>26</sup> [https://en.wikipedia.org/wiki/Load\\_balancing\\_\(computing\)](https://en.wikipedia.org/wiki/Load_balancing_(computing))

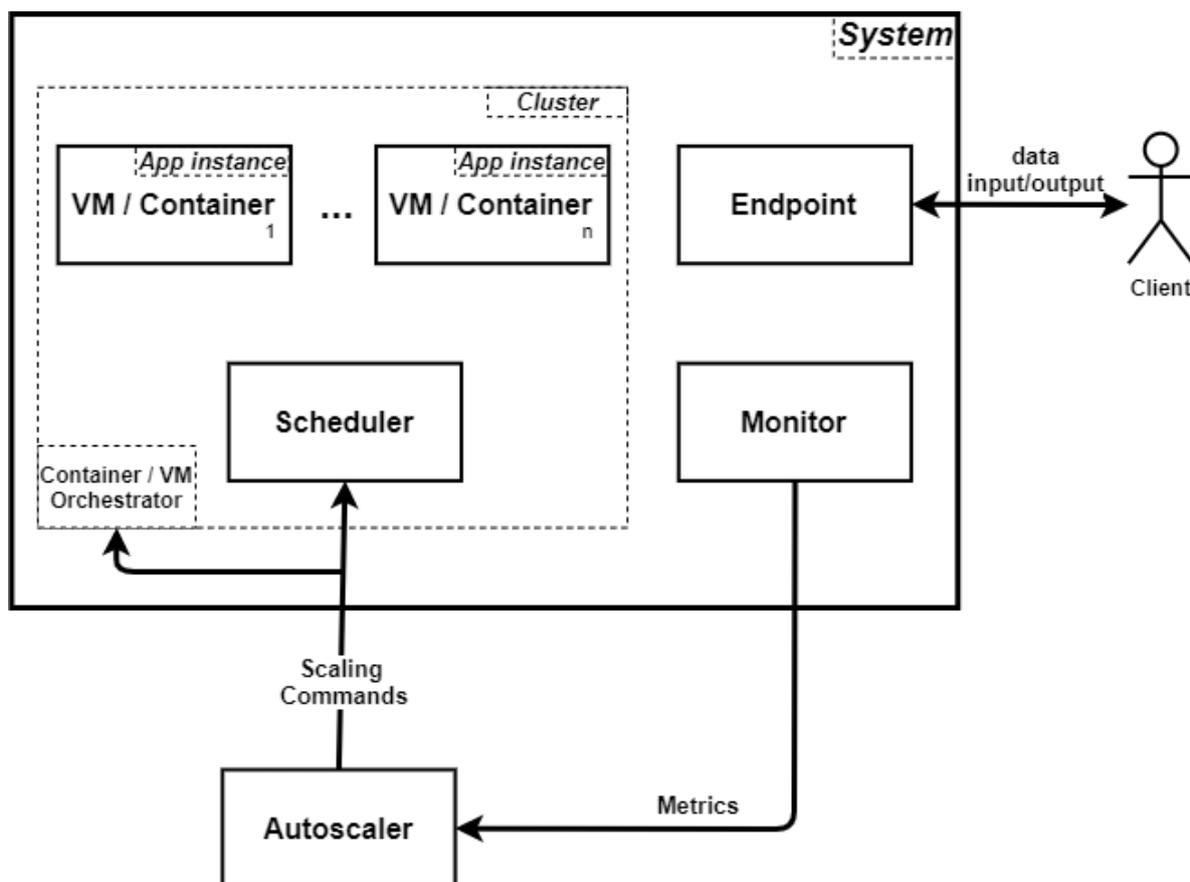


Figure 13 - General structure of system. The interconnection between the system's components is omitted.

The autoscaler can be integrated in any system which supports the functionalities referred above so as to enable dynamic resource scaling. The autoscaler automatically adjust the allocated resources of the application in order to adopt to the incoming workload. To do so, metrics related to application's state (allocated resources, workload and performance) are retrieved and based on some policy (reactive or proactive) a scaling decision (increase/decrease the allocated resources) is made.

In this thesis, the system consists of a Flink Cluster. The Flink Cluster size is adjusted by adding or removing TaskManagers which run on containers or virtual machines by the container/VM orchestrator. (e.g. Docker or OpenStack) Afterwards, the available TaskManagers can be allocated to running jobs by the JobManager (scheduler). Flink automatically load balance the workload of the running job among the allocated resources (TaskManagers). In addition, each TaskManager has exactly one task slot. Kafka plays the role of the endpoint of the system where data get in or out through Kafka topics. Prometheus is used for the monitoring of the system in terms of allocated resources, workload and performance. The form of the system is depicted in Figure 14.

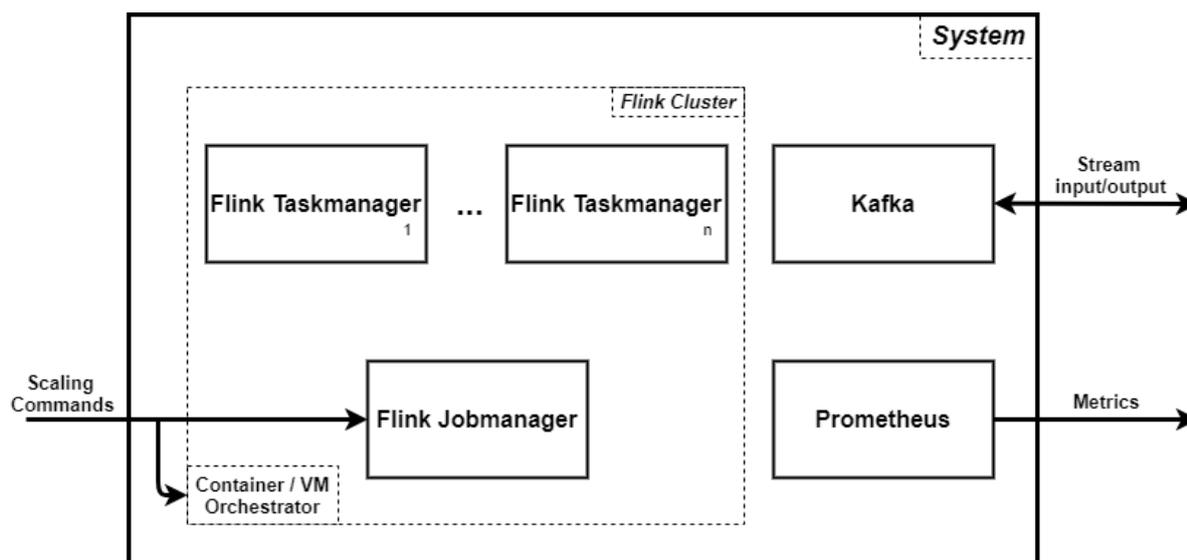


Figure 14 - Abstract system of Flink Cluster

### Defined SLA

We measure the percentage of slow records indirectly using the workload and queue metrics. The workload indicates how many records per second the system receives. The queue is the number of records which are not consumed instantly but remain in a buffer for a short period of time. The queue is an indicator which specifies in what degree the system is able to keep up with the rate of incoming records. The percentile of slow records is calculated by the formula  $\frac{queue}{workload}$ .

SLA is defined as the percentile of slow records during a period of time (e.g. per second) that the application owner or user can accept. Records are considered as slow records if they are not consumed (processed) as soon as possible but remain in a queue. If the measured percentile is above a threshold value (e.g. 90%), the system encounters a violation of the SLA. The percentile defines the worst performance that the application owner can tolerate.

### 3.2. Reactive Scaler

The autoscaler is based on a reactive mechanism which acts as soon as the SLA is violated. The scaler receives metrics from the system in order to assess the current state of the application. If the system is over-utilized or under-utilized, the autoscaler adjusts accordingly the allocated resources (adds/removes servers). The interaction between the system and the autoscaler is depicted in Figure 15.

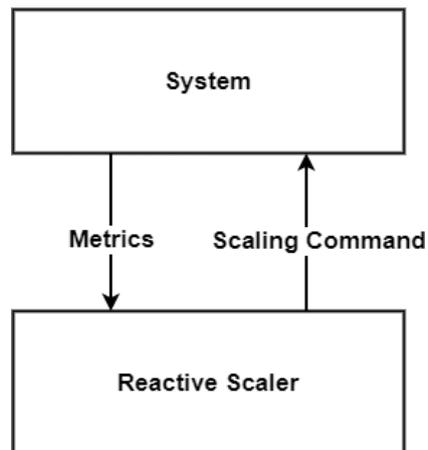


Figure 15 - Interconnection of System and Reactive Scaler

### 3.2.1. Architecture

The Reactive Scaler consists of the Capacity Calculator and the Reactive Policy. The interconnection of the components and their interconnection is depicted in Figure 16.

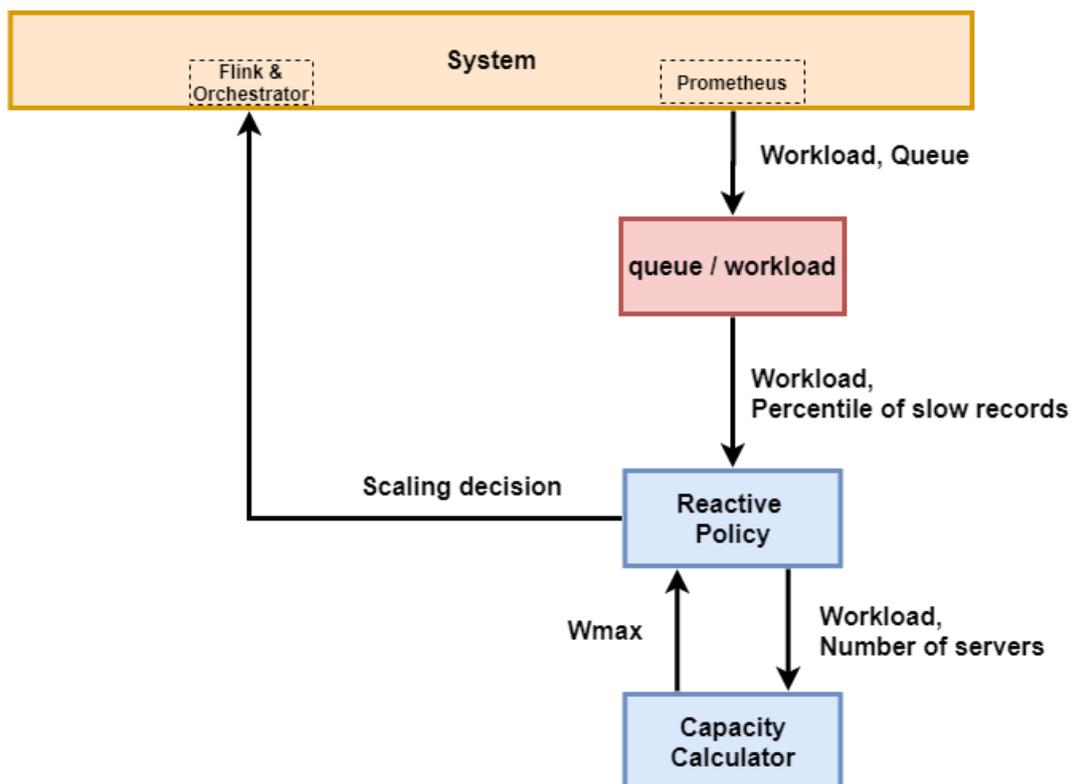


Figure 16 - Workflow of Reactive Scaler

## Capacity Calculator

The purpose of capacity is to measure how many records per second a single TaskManager is able to process. Due to the fact that every server is created from the same instance, the capacity of each running TaskManager is approximately the same. The capacity represents the maximum incoming rate of records which a TaskManager can consume while its queue length does not exceed the SLA. Consequently, we are able to calculate the maximum workload that each number of TaskManagers, running the targeted job, can consume without causing SLA violations.

In order to measure the capacity of the system the Algorithm 1 is executed, which performs the following steps:

1. Let  $w_t$  and  $n_t$  be the workload and number of servers at time  $t$ , respectively.
2. When the SLA is violated for the first time, we calculate the capacity  $w_{max}$  as  $w_t / n_t$ .
3. Each time the SLA is violated again, the capacity value is updated by taking the average of the old value, weighted with the number of the previous capacity values, and the current value  $w_t / n_t$ . This step is performed for as long as the controller is in exploration mode.

The function is executed every time a scale-up action is performed in exploration mode. This is done, every time the SLA is violated. Instead of calculating the capacity value  $w_{max}$  once, the value is constantly updated in order to get a better approximation of the actual capacity of a single server.

---

**Algorithm 1** Calculate the capacity of the application  $w_{max}$ .

---

```

1: procedure CAPACITYCALCULATOR
2:    $w_t \leftarrow \text{CurrentWorkload}$ 
3:    $n_t \leftarrow \text{CurrentNumberOfTaskmanagers}$ 
4:    $oldW_{max}, counter \leftarrow \text{GetCapacity}()$ 
5:   if  $oldW_{max} == \text{Null}$  then
6:      $w_{max} \leftarrow \frac{w_t}{n_t}$ 
7:   else
8:      $w_{max} \leftarrow \frac{counter \cdot oldW_{max} + \frac{w_t}{n_t}}{counter + 1}$ 
    $\text{setCapacity}(w_{max}, counter + 1)$ 

```

---

## Reactive Policy

The decision is based on the specified SLA. The reactive scaler checks if one of the following actions should be taken every 15 seconds.

### Scale up

The scaler increases the number of allocated TaskManagers as soon as the SLA is violated. There are two possible options for the increase rate of TaskManagers:

1. The scaler adds one new TaskManager at a time. This option is simple but it has the drawback that it is unable to keep up with a high increased workload. If more than one TaskManagers are needed, the allocation will be gradual causing over-utilization of the application.
2. The capacity  $w_{max}$  is utilized to calculate the exact number of required TaskManagers for the incoming workload  $w_t$ . Specifically, at time  $t$ , the scaler, adds  $w_t / w_{max}$  TaskManagers. The particular option is able to keep up to any workload rate. In case of a slow increase rate of workload the scaler adds one Taskmanager at a time, similarly to option 1.

### Scale down

At time  $t$ , the exploration policy will remove a TaskManager only if there are more than 90% of TaskManagers running than those are needed. In other words, the scaler removes a server if there are more than  $\lceil w_t / w_{max} / 0.9 \rceil$  servers running. Furthermore, we check if this condition is satisfied for a number of consecutive samples, referred as  $S$  (in Algorithm 2, we set  $S = 10$ ). A single sample which does not represent the real state of the system could lead to a false action. As the number of the samples is increasing, the probability of false action is decreasing.

The main idea of both scaling actions is described in Algorithm 2. The implementation of the scaling actions (ScaleUp/ScaleDown) is described in detail, in Section 4.

---

#### Algorithm 2 Scaling policy of Reactive Scaler.

---

```

1: procedure REACTIVESCALER
2:   while in ExplorationMode do
3:      $w_{max} \leftarrow \text{GetCapacity}()$ 
4:      $w_t \leftarrow \text{CurrentWorkload}$ 
5:      $n_t \leftarrow \text{CurrentNumberOfTaskmanagers}$ 
6:      $q_t \leftarrow \text{CurrentQueue}$ 
7:      $\text{slowRecords}_t \leftarrow \frac{q_t}{w_t}$ 
8:      $\text{counter} \leftarrow 0$ 
9:     if  $\text{slowRecords} \geq \text{SLA}$  then
10:        $\text{CalculateCapacity}()$ 
11:        $\text{ScaleUp}()$ 
12:     else if  $n_t > \left\lceil \frac{w_t}{w_{max} \cdot 0.9} \right\rceil$  then
13:        $\text{counter} \leftarrow \text{counter} + 1$ 
14:       if  $\text{counter} \geq 9$  then
15:          $\text{counter} \leftarrow 0$ 
16:          $\text{ScaleDown}()$ 

```

---

### 3.3. Proactive Scaler

The autoscaler implements a controller which is in a constant switch between two states, the exploration mode and the optimal control as shown in the Figure 17. When the controller is initialized for first time, it enters exploration mode in which the behavior of the system is discovered while working reactively. When this phase is successfully completed, the controller switches to optimal control. In this phase, the gathered information from the previous step is used for the training of a statistical machine learning model. Then, the proactive scaler takes over the control for scaling actions by predicting the near future behavior of the application. Although, changes in the system (e.g. updates or failures) or in the running application itself (e.g. introducing a new feature) could affect the overall performance of the application (e.g. increase the latency per request). As a result, the optimal control is no longer able to take effective scaling actions and the proactive scaler has to be re-trained. In this case, the controller returns again to exploration mode in order to discover the new behavior of the application.

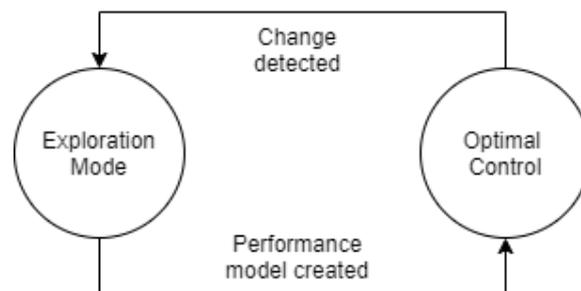


Figure 17 - States of controller

#### 3.3.1. Exploration mode

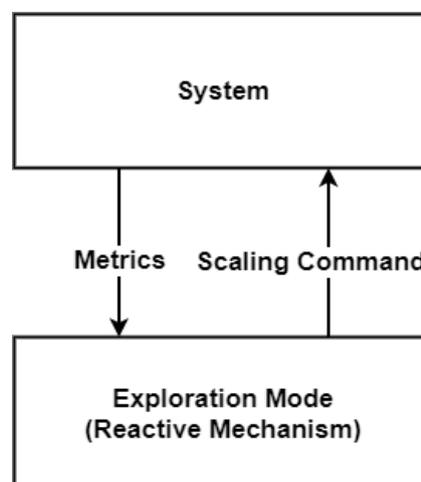


Figure 18 - Interconnection of System and Exploration Mode

The controller explores the behavior of the application under varying workload and different parallelism (number of servers). The outcome of this phase is a machine learning model which is able to predict the performance of the running application. During exploration, the controller uses a reactive mechanism, as shown in Figure 18, in order to ensure the normal operation of the application in terms of performance and resource utilization. The components of the exploration mode and the way they communicate each other is depicted in Figure 19.

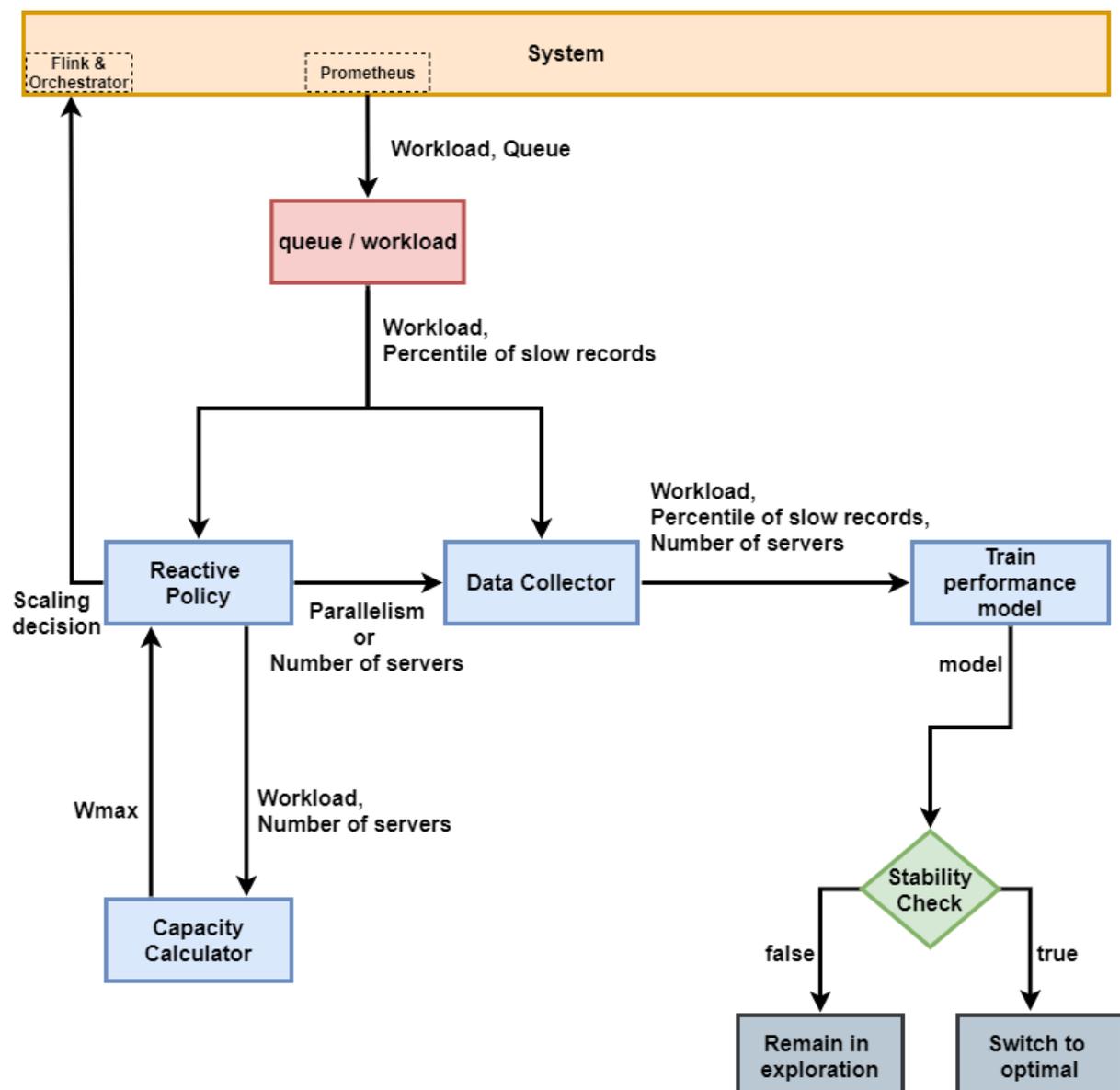


Figure 19 - Workflow of Exploration Mode

## Reactive Scaler

During the exploration mode, there are not enough data to extract a pattern about the behavior for the running application. For this reason, a reactive mechanism is used to decide whether the system should make a scaling action. The decision is based on the specified SLA. The Reactive Scaler operates as described in section 3.2. and is consists of the Reactive Policy and the Capacity Calculator.

## Data Collector

This component collects and stores data throughout the operation of the system both in exploration mode and optimal control. In exploration mode the following functions are supported:

- **Metrics insertion:** Each time the autoscaler retrieves metrics from the system in order to examine if a scaling action should be made (every 15 seconds in our case), the metrics are also appended in data structures. Specifically, at a time  $t$ , we append the metrics  $w_t, n_t, performance_t$  where  $w$  the workload,  $n$  the number of allocated servers and  $performance$  the percentile of slow records computed over the total number of records generated during a prespecified time interval (e.g. 15 seconds in our case). The stored metrics form the dataset of the exploration mode. The metrics are stored in memory and periodically are moved into files in order to ensure free space in memory and durability of data. For example, after 1000 insertions the data are moved from memory to disk. The data are not stored directly into disk since that would significantly increase the overall number of disk accesses. Considering that operations into disk are slower than those into memory<sup>27</sup>, more disk accesses are resulting into higher latency per metric's insertion.
- **Metrics retrieval:** Each time the controller needs to create a performance model, it retrieves the stored metrics. Note that the metrics could be divided both in disk and memory (as explained in Metrics insertion), so the Data Collector retrieve metrics from both data storages.

## Performance Model Training

The performance model captures the relation between workload, number of TaskManagers, and the percentage of slow records. The model is trained using non-linear regression<sup>28</sup> and it includes two independent variables and one dependent

---

<sup>27</sup> [https://en.wikipedia.org/wiki/Computer\\_data\\_storage](https://en.wikipedia.org/wiki/Computer_data_storage)

<sup>28</sup> [https://en.wikipedia.org/wiki/Nonlinear\\_regression](https://en.wikipedia.org/wiki/Nonlinear_regression)

variable. In other words, the model is described by the following form:  $f(x_1, x_2) = y$  where  $x_1$  is the workload,  $x_2$  is the number of TaskManagers and  $y$  is the percentage of slow records. The independent variables can be also represented in one variable  $X$  for simplicity:  $X = (x_1, x_2)$ .

The degree of the polynomial is selected by applying brute-force search<sup>29</sup>, in which the solution exhaustively searched. The procedure is depicted in Algorithm 3. The following steps are performed:

1. We define the set  $P = \{1 \dots n\}$  which contains the degrees that will be tested. The  $n^{\text{th}}$  degree is the maximum degree that will be applied. The computational complexity is significantly increasing as the degree takes higher values.
2. For each degree, a model is created using non-linear regression using the collected dataset.
3. The Route-Mean-Square-Error (RMSE) of each of model is measured using the original dataset. The RMSE is calculated by comparing the actual value and the predicted value of the model. RMSE is a factor that describes how concentrated the dataset is around the regression line.
4. The final degree of the model is the one with me minimum RMSE. Therefore, the one which is the best fit for the dataset.

The training dataset is derived from the production data which are collected during the exploration mode. Periodically, a performance model is trained and then evaluated according to its accuracy to predict the behavior of the application. The accuracy of the model determines whether the system will switch to optimal control or not.

---

**Algorithm 3** Find which degree is the best fit for the dataset (X,y) using brute-force search.

---

```

1: procedure FINDDEGREE(X,y,maxDegree)
2:    $minRMSE \leftarrow \infty$ 
3:    $bestDegree \leftarrow 0$ 
4:    $degree \leftarrow 0$ 
5:   while  $degree < maxDegree$  do
6:      $model \leftarrow NonLinearRegression(X,y,degree)$ 
7:      $y_{actual} \leftarrow y$ 
8:      $y_{predicted} \leftarrow model.predict(X)$ 
9:      $RMSE \leftarrow MeasureRMSE(y_{predicted}, y_{predicted})$ 
10:    if  $RMSE < minRMSE$  then
11:       $minRMSE \leftarrow RMSE$ 
12:       $bestDegree \leftarrow degree$ 
13:     $degree \leftarrow degree + 1$ 
return  $bestDegree$ 

```

---

<sup>29</sup> [https://en.wikipedia.org/wiki/Brute-force\\_search](https://en.wikipedia.org/wiki/Brute-force_search)

## Stability Check

Each time we train a performance model, we check if we can rely on its predictions. To do so, we use bootstrap sampling<sup>30</sup> to measure the standard deviation of the model. Bootstrapping is a technique which measures the properties of an estimator (such as its variance).

The method is described in Algorithm 4. Bootstrap sampling works as follows:

1. Let  $D$  be the original dataset and let  $N$  be its size, which is used to train the model. In addition, let  $k$  be the number of the bootstrap samples that will be created. As the  $k$  increasing, the approximation of the standard deviation is closer to the real value but also increases the computational complexity. In our experiments, we set  $k = 250$ .
2. Create  $k$  random samples with replacement with sample size  $N$ . The term “with replacement” refers to the way that a sample is created. Each sample is filled by randomly selecting  $N$  elements from the original dataset  $D$ . The choices are independent from each other, which means that some elements could be selected multiple times while others are not selected at all. In the end of this step, there are  $D_1 \dots D_k$  samples.
3. For each  $D_i$  sample, the standard deviation is calculated.
4. The standard deviation of the model is measured as the mean of the  $k$  standard deviations measured by the  $D_i$  models.

---

**Algorithm 4** Standard Deviation measurement of Performance model using Bootstrap sampling.

---

```

1: procedure MEASURESTANDARDDEVIATION( $X$ , PerformanceModel,  $k$ )
2:    $D \leftarrow X$ 
3:    $N \leftarrow \text{Length}(D)$ 
4:    $\text{StandardDeviationList}[] \leftarrow \text{List}(k)$ 
5:    $i \leftarrow 0$ 
6:   while  $i < k$  do
7:      $X_i \leftarrow \text{RandomChoices}(X, N)$ 
8:      $D_i \leftarrow \text{PerformanceModel.predictList}(X_i)$ 
9:      $\text{value} \leftarrow \text{StandardDeviation}(D_i)$ 
10:     $\text{StandardDeviationList}[i] \leftarrow \text{value}$ 
11:     $i \leftarrow i + 1$ 
12:   $\text{standardDeviation} \leftarrow \text{mean}(\text{StandardDeviationList})$ 
  return  $\text{standardDeviation}$ 

```

---

The stability check is performed in two phases and is depicted in Algorithm 5:

---

<sup>30</sup> [https://en.wikipedia.org/wiki/Bootstrapping\\_\(statistics\)](https://en.wikipedia.org/wiki/Bootstrapping_(statistics))

### Phase 1

In this phase, we check if the dataset consists of enough data to create an accurate model. We retrieve from the Data Collector the metrics which consists the independent variables of the performance model  $x_1, x_2$ , which are the workload and the number of Taskmanagers respectively. We calculate the standard deviation of the predicted values  $y$  by applying bootstrap sampling. If the standard deviation of the predicted values is less than the *model stability threshold*  $\lambda$  then we continue to Phase 2. Otherwise, we ignore the trained model and we remain in exploration mode in order to collect more data.

### Phase 2

In this step, we check if the created model is accurate to predict the performance of the application for the current number of TaskManagers  $n_{current}$ . The phase 1 checks the accuracy of the model at least up to the point  $w_{current}$ , which is the current workload (the workload value when the stability check began). In Phase 2, we check if the model is stable for the rest of the workload that  $n_{current}$  Taskmanagers can process without causing SLA violations. The rest of that workload consists of the points  $w_{current}$  through  $n_{current} \cdot w_{max}$  which gives us the maximum workload that  $n_{current}$  can process with good performance according the capacity value.

The procedure is similar to Phase 1. We perform bootstrap sampling at the points  $(w_{current}, n_{current})$  through  $(n_{current} \cdot w_{max}, n_{current})$ . If the standard deviation of the predicted values  $y$  is less than the *model stability threshold*  $\lambda$  then we switch to Optimal Control. Otherwise, we ignore the model and we remain in exploration mode in order to collect more data.

Note that in this phase, the capacity of the application  $w_{max}$  must be available, which means that the controller has scaled up at least one time. If the value  $w_{max}$  has not been calculated yet, the stability check does perform phase 1 (since phase 2 cannot be executed at all) and the controller remains in exploration mode for data collection.

**Algorithm 5** Stability Check phases.

---

```

1: procedure STABILITYCHECK(PerformanceModel, X,  $\lambda$ , k )
2:    $w_{max} \leftarrow GetCapacity()$ 
3:   if  $w_{max} == Null$  then return false ▷ Phase 1
4:    $StandardDeviation_{phase1} \leftarrow MeasureStandardDeviation(X, PerformanceModel, k)$ 
5:    $stability_{phase1} \leftarrow StandardDeviation \leq \lambda$ 
6:   if  $stability_{phase1}$  then ▷ Phase 2
7:      $w_{current} \leftarrow GetCurrentWorkload()$ 
8:      $n_{current} \leftarrow GetCurrentNumberOfServers()$ 
9:      $w_{list} \leftarrow [w_{current} \dots n_{current} \cdot w_{max}]$ 
10:     $X_2 \leftarrow (w_{list}, n_{current})$ 
11:     $StandardDeviation_{phase2} \leftarrow MeasureStandardDeviation(X_2, PerformanceModel, k)$ 
12:     $stability_{phase2} \leftarrow StandardDeviation_{phase2} \leq \lambda$ 
13:    if  $stability_{phase2}$  then return true
    return false

```

---

The *stability threshold*  $\lambda$  is the maximum acceptable error of the model and it is user defined. The *stability threshold*  $\lambda$  is common for both phases. By selecting a small value, the trained model will be more accurate in term of its prediction capability. In addition, the exploration policy could last longer, since it continues the data collection until the standard deviation of the trained model is less than  $\lambda$ . A good value of  $\lambda$  is defined as the balance between of the acceptable error of the model and the duration of the exploration mode. In this work, we set  $\lambda = 0.05$

### 3.3.2. Optimal Control

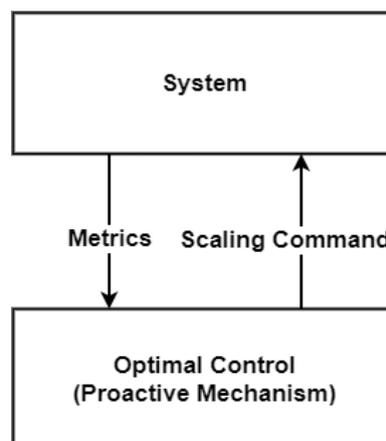


Figure 20 - Interconnection of System and Optimal Control

In this mode, the behavior of the application is considered known and it is described by the performance model. The system switches from reactive mechanism to proactive, as shown in Figure 20, and no longer uses instant metrics to make a decision about parallelism (number of servers). Instead, near future predictions are used to determine the optimal parallelism of the system. The model attempts to predict and prevent possible SLA violations before they occur. The Figure 21 describes the workflow of Optimal Control and its components.

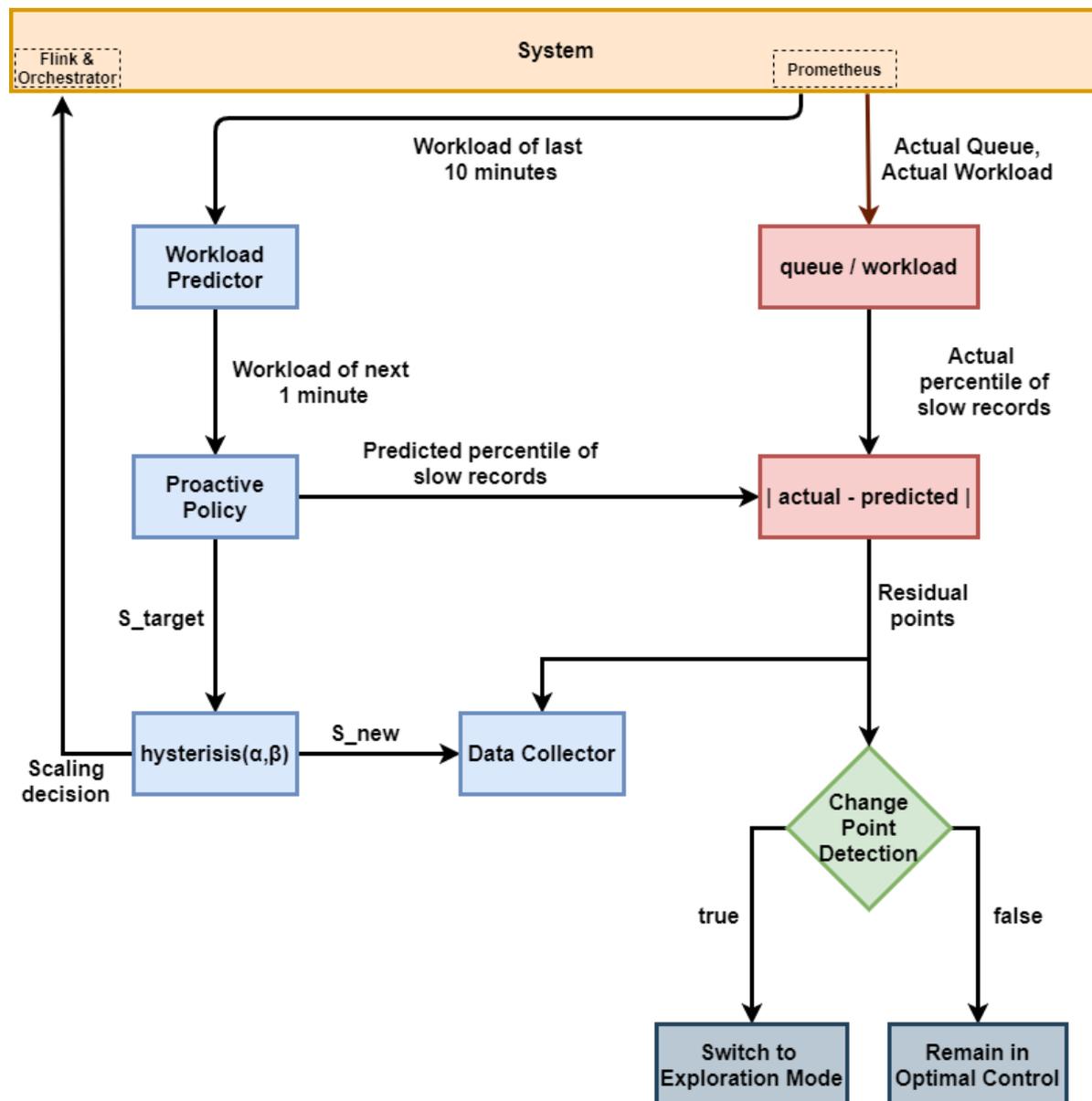


Figure 21 - Workflow of Optimal Control

## Workload Predictor

The Workload Predictor captures the slope of the curve which the workload distribution followed in the near past. Considering that the slope will still apply in the near future, we are able to predict the upcoming workload. The slope of the curve is extracted using linear regression<sup>31</sup> which is trained each time a new prediction is required. A new prediction is required each time the Proactive Scaler needs to check whether a scaling action should be made. The Workload Predictor does not attempt to capture the entire distribution of the workload but just the recent workload rate (if the workload increases, decreases or remains steady). In this way the predictor is able to adjust to any distribution.

During the operation of optimal control, we predict the next 1 minute of workload using the most recent 10 minutes. The recent workload values could be retrieved by the monitor's storage (e.g. Prometheus) or by the Data Collector. In Figure 21, the data are retrieved from the monitor. The Workload Predictor outputs an array of the predicted workload values for the desired time series with some step called  $w_{step}$ . Specifically, for future predictions for the next 60 seconds with step of 5 seconds ( $w_{step} = 5$ ), we get the following result:  $w_{future} [w_1, w_{60}, 5]$ . The length of the array is equal to  $\frac{60}{w_{step}} = \frac{60}{5} = 12$  points.

## Proactive Policy

The performance model, which was trained during the exploration mode, is used to predict the behavior of the application for future workload. The overall procedure for scaling is described by Algorithm 6. The following steps are performed:

1. The next 1 minute of the future workload  $w_{future}$  is predicted using the Workload Predictor.
2. We define the set  $[1 \dots n_{max}]$  which represents the possible parallelisms that the application can take. The value  $n_{max}$  is the maximum number of server that the system can allocate to the application (e.g. due to resource limitation). For each possible parallelism  $n_i \in [1 \dots n_{max}]$  and the future workload, we predict the percentile of slow records using the performance model. In other words, the following function is executed for point of  $w_{future}$  and each parallelism:  $f(n_i, w_{future}) = slowRecords_{future}$ . The output  $slowRecords_{future}$  is an array which has the same length with  $w_{future}$  and each predicted performance point has  $w_{step}$  time distance from the next one.

---

<sup>31</sup> [https://en.wikipedia.org/wiki/Linear\\_regression](https://en.wikipedia.org/wiki/Linear_regression)

3. The predictions  $slowRecords_{future}$  associated with the parallelism  $n_i$  are evaluated according the given SLA. In detail, we reject the parallelism  $n_i$  if at least one of the predicted performance points in  $slowRecords_{future}$  violates the SLA.
4. The optimal parallelism  $S_{target}$  is the minimum parallelism  $n_i$  which satisfies the SLA.

To prevent rapid oscillations in the controller, we apply hysteresis with gains  $\alpha$  and  $\beta$ . Both values are defined in the set  $[0,1]$ . The final parallelism  $S_{new}$  is calculated as follows:

$$\rightarrow \text{if } S_{target} > S_{old} \text{ then } S_{new} = S_{old} + \alpha(S_{target} - S_{old})$$

$$\rightarrow \text{if } S_{target} < S_{old} \text{ then } S_{new} = S_{old} + \beta(S_{target} - S_{old})$$

The parameter  $\alpha$  specifies how quickly the system will make the transition from  $S_{old}$  to  $S_{target}$ . For a low value of  $\alpha$  targeted number of TaskManagers, will be added gradually to the running application. On the other hand, a high value will lead to the desirable number of TaskManagers in a short period of time. This parameter has a significant effect on the performance since the lack of adequate resources will over-utilize the application causing SLA violations. For this reason, a value close to one is preferred so as the  $S_{target}$  be reached quickly.

On the other hand, the parameter  $\beta$  defines how quickly the system will scale down from  $S_{old}$  to  $S_{target}$ . Similarly, to  $\alpha$ , a high value will remove TaskManagers with a fast rate while a lower value with a slower one. This parameter does not cause any SLA violation but could cause under-utilization of the application if resources remain idle for long time. Eventually, the  $S_{target}$  will be reached even if the parameter  $\beta$  has a low value.

In this work, the gain parameters are defined by the user. In our experiments we set  $\alpha = 0,9$  and  $\beta = 0,4$ . Although, the implementation of a Control Policy Simulator [2] could provide a solution for finding dynamically which gain parameters are optimal according the incoming workload distribution.

**Algorithm 6** Scaling policy during optimal control

---

```

1: procedure PROACTIVESCALER
2:    $w_{future} \leftarrow \text{WorkloadPredictor}()$ 
3:    $parallelismSet \leftarrow [1 \dots n_{max}]$ 
4:    $S_{target} \leftarrow n_{max}$ 
5:   for  $n_i \in parallelismSet$  do ▷ select optimal parallelism
6:      $performancePoints \leftarrow \text{Predict}(n_i, w_{future})$ 
7:      $evaluation \leftarrow \text{CheckViolation}(performancePoints)$ 
8:     if  $evaluation$  then
9:        $S_{target} \leftarrow n_i$ 
10:      break
11:    $S_{new} \leftarrow \text{Hysterisis}(S_{target})$ 
12:    $scale(S_{new})$ 
13: procedure WORKLOADPREDICTOR
14:    $w_{past} \leftarrow \text{GetPastWorkload}(10mins)$ 
15:    $slope \leftarrow \text{LinearRegression}(w_{past})$ 
16:    $w_{future} \leftarrow slope.predict(1min)$ 
17:   return  $w_{future}$ 
17: procedure CHECKVIOLATION(performancePoints)
18:   for each  $point_i \in performancePoints$  do
19:     if  $point_i \geq SLA$  then return false
20:   return true
20: procedure HYSTERISIS( $S_{target}$ )
21:    $S_{old} \leftarrow \text{CurrentNumberOfTaskmanagers}$ 
22:   if  $S_{target} > S_{old}$  then
23:      $S_{new} \leftarrow S_{old} + \alpha \cdot (S_{target} - S_{old})$ 
24:   else if  $S_{target} < S_{old}$  then
25:      $S_{new} \leftarrow S_{old} + \beta \cdot (S_{target} - S_{old})$ 
26:   else
27:      $S_{new} \leftarrow S_{old}$ 
28:   return  $S_{new}$ 

```

---

**Change Point Detection**

Changes in the environment such as system updates or hardware failures could lead the application to behave in a different way. In case the performance model is no longer able to accurately predict the behavior of the application, it should be discarded and train a new one.

We use the residuals of the percentile of slow records to assess the accuracy of the model. Under steady time, a residual is calculated from the form:  $|actual\ performance - predicted\ performance|$  which gives us the difference of the predicted percentile of slow records and the actual one. The residuals of an accurate model should be almost equal to zero or in other words the predictions of the model should be verified by the actual performance of the application.

Residual points are constantly generated while the controller is in optimal control, hence, they form a live-streaming time series. The frequency of residual generation is regulated by the step of Workload Predictor ( $w_{step}$ ), which our case is 5 seconds. Since such data are unbounded, a value has to be assessed as soon as it arrives. Online change detection [10] which captures abrupt changes in the streaming data, can be used to evaluate the generated residual points. The function returns a score which represents the degree of prediction failure. We consider that a change occurred when the score exceeds a specified threshold related to configuration of the chosen detection method. This threshold has to be set appropriately so as to captures the acceptable range of prediction deviations. The value of the threshold can be found experimentally.

In case of very low threshold the controller will mark the performance model as inaccurate for very small prediction deviations. On the contrast, a quite high threshold could lead to SLA violations in case of significant prediction failure of the performance model. If a change point occurs the controller switches to exploration mode. When the controller returns to exploration mode, it starts the dataset collection from the beginning. Since a change point occurred, the previous dataset is considered inaccurate.

## Data Collector

In optimal control, the Data Collector supports functions for metrics and performance residuals insertion. Although the scaler examines if an action every 1 minute, metrics retrieval takes place more frequently.

- Metrics insertion: Every  $w_{step}$  seconds the controller retrieves the metrics from the systems and appends them in data structures. Specifically, at a time  $t$ , we append the metrics  $w_t, n_t, performance_t$  where  $w$  the actual workload,  $n$  the number of allocated servers and  $performance$  the actual percentile of slow records. This function is the same as the described one in exploration mode but it is called in a different frequency.
- Performance Model residuals insertion: We use the retrieved value  $performance_t$  from the previous function, let's call it  $ActualPerformance_t$ . We also get the corresponding predicted value for time  $t$  from the array  $slowRecords_{future}$ , let's call it  $PredictedPerformancePoint_t$ . The array has  $\frac{60}{w_{step}} = \frac{60}{5} = 12$  elements, so we get the 12 corresponding actual performance points. For example, the first element of the array  $slowRecords_{future}$  is the prediction of performance in the next 5 seconds. So, we wait 5 seconds and we measure the real performance. The residual is calculated as described in "Change Point Detection" module, by the form  $|actual\ performance - predicted\ performance|$ . Finally, the residual value is appended into data

structure in memory and periodically moved into disk (similarly to metric insertion function).

The workflow of both functions is depicted in Algorithm 7. The aforementioned data, are not retrieved by the controller but are used for the assessment of the optimal control (e.g. in experiments).

---

**Algorithm 7** Metrics and Residuals insertion to Data Collector.

---

```

1: procedure DATAINSERTION( $PredictedPerformancePoints_{future}, w_{step}$ )
2:   for each  $PredictedPerformance_t \in PredictedPerformancePoints_{future}$  do
3:      $w_t, n_t, performance_t \leftarrow Prometheus.GetMetrics()$  ▷ Metrics Insertion
4:      $DataCollector.MetricsInsertion(w_t, n_t, performance_t)$ 
5:      $ActualPerformance_t \leftarrow performance_t$  ▷ Residual Insertion
6:      $PerformanceResidual_t \leftarrow |ActualPerformance_t - PredictedPerformance_t|$ 
7:      $DataCollector.ResidualInsertion(PerformanceResidual_t)$ 
8:     wait  $w_{step}$  seconds

```

---

### 3.3.3. Comparison to related work

In comparison to the publications [1] and [2], the autoscaler of the work has the following differences:

1. Targeted applications: Bodík et al. examine the dynamic resource allocation to Web 2.0<sup>32</sup> applications while in this work we implement resource scaling on streaming applications. Web 2.0 applications receive requests by end users and generate a response back to them. On the other hand, stream processing receives high speed unbounded data in the form of records which could be generated by users, software agents (e.g. web scrapping agents<sup>33</sup>) or devices (e.g. sensors).
  - Performance measuring: In publications [1,2] the performance of the application is measured using the latency per request (total required time to response back to the user). Since streaming applications does not necessary generate a result per record, we assess their performance by measuring the number of records which are not processed immediately but remain in a buffer (queue).
  - Application deployment environment: In our work, we perform resource scaling on a Flink Cluster by adding or removing allocated TaskManagers (in the form of containers) to a running job. On the contrary, Bodík et al. adjust the allocated resources by adding or removing server instances as virtual machines.

---

<sup>32</sup> [https://en.wikipedia.org/wiki/Web\\_2.0](https://en.wikipedia.org/wiki/Web_2.0)

<sup>33</sup> [https://en.wikipedia.org/wiki/Web\\_scraping](https://en.wikipedia.org/wiki/Web_scraping)

- Exploration Policy: The publication [1] adds/removes servers with the goal to reach a specified latency (which takes a random value between 0 and an upper safety threshold) in order to explore the system's behavior under different values of latency. We did not follow the same approach since in the current Flink version, rescaling of a running job can be performed by restarting the job resulting to increased recovery delay. In this way, multiple scale actions could significantly increase the total time where the application is in recovery and thus the performance is dropped. Instead, we initialize the application to a user-defined parallelism and let the controller to adjust the resources according the workload.
- Change Point Detection: In publication [2] the implemented change point detection algorithm is able to capture both abrupt and gradual changes. In this work, the used algorithm is not able to capture gradual changes in the accuracy of the Performance Model.
- Hysteresis gains definition: In the current work, the gains  $\alpha, \beta$  are user-defined, while in publication [2] the optimal gain values are defined using a policy search algorithm.

## 4. Implementation

### 4.1. Flink deployment options

Flink can be installed directly on bare-metal machines or be deployed on virtual machines or containers. Using cloud technologies, the resources can be easily extended by adding new virtual machines. There are several options to select for deployment in the cloud technologies, in which the scaling action can be performed manually or using a software agent.

#### Virtual machines infrastructure

Flink runs as a cluster on multiple virtual machines. At the beginning, the cluster is initialized with a number of virtual machines but it can be easily expanded by adding new machines to the cluster. Likewise, the cluster may shrink with the removal of unnecessary machines. The architecture is depicted in Figure 22.

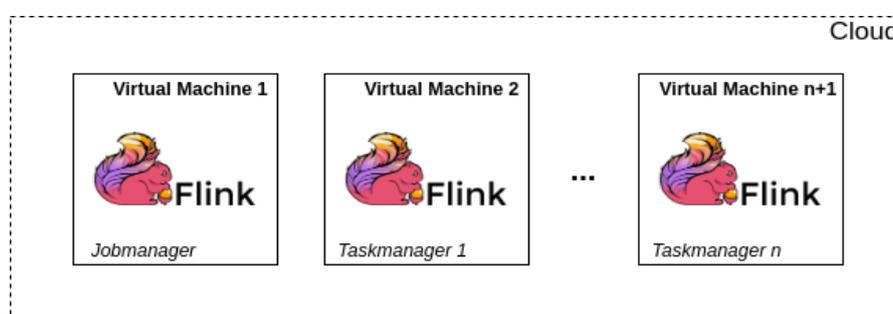


Figure 22 - Flink cluster deployed on virtual machines

#### Containers infrastructure

The Flink cluster is deployed in a containerized environment. Such solutions are Mesos, Docker and Kubernetes. In this case, Flink instances run inside to containers, as shown in Figure 23. Containers are isolated processes by reserving resources from bare metal or virtual machines. Resources are not shared among containers but used by only one at a time. Therefore, the number of deployed containers is limited by the resources of the machine. Containers by default reserve as much resources are required to perform their operations. The maximum allocation of resources to each one can be restrained. The size of the cluster is modified, using vertical scaling, by adding or removing containers which run Flink processes.

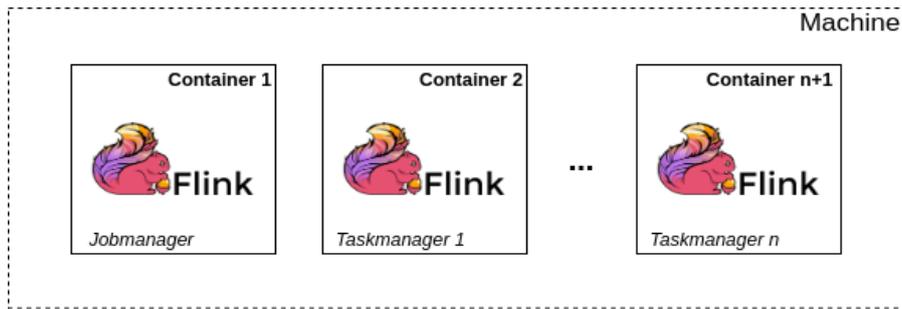


Figure 23 - Flink cluster deployed on containers

**Hybrid infrastructure**

This type of architecture is a combination of the previous ones. Each virtual machine hosts a number of containers forming the Flink cluster. The cluster can be expanded by added new virtual machines and then deploy more containers inside them. In this way, further isolation between processes, running on the same machine, is achieved. The architecture, as depicted in Figure 24, supports both horizontal and vertical scaling.

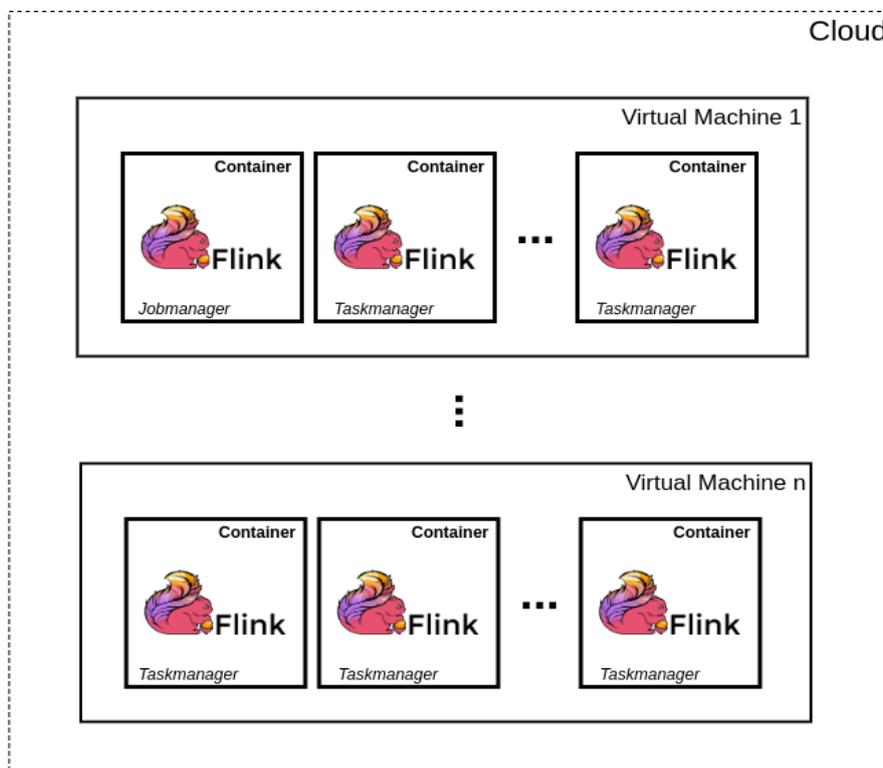


Figure 24 - Flink cluster deployed on containers which run on virtual machines

## 4.2. Architecture

The implemented infrastructure uses Flink in order to run stream processing applications. Kafka plays the role of the system’s endpoint, so stream gets in or out through Kafka topics. The Autoscaler adjusts the allocated resources of the targeted application and Faban simulates the workload. The Autoscaler and Faban are deployed directly on virtual machines in OpenStack. The rest of the components are deployed as Docker containers which are distributed among virtual machines. Specifically, the Flink Cluster and Prometheus hosted in the same machine, while Zookeeper, Kafka and JMX exporter are placed together in another one. The virtual machines which host Docker containers, form a Docker Swarm cluster. In this way, the management of all containers can be performed by a single machine, the Docker Swarm Manager. The architecture of the implemented infrastructure is depicted in Figure 25.

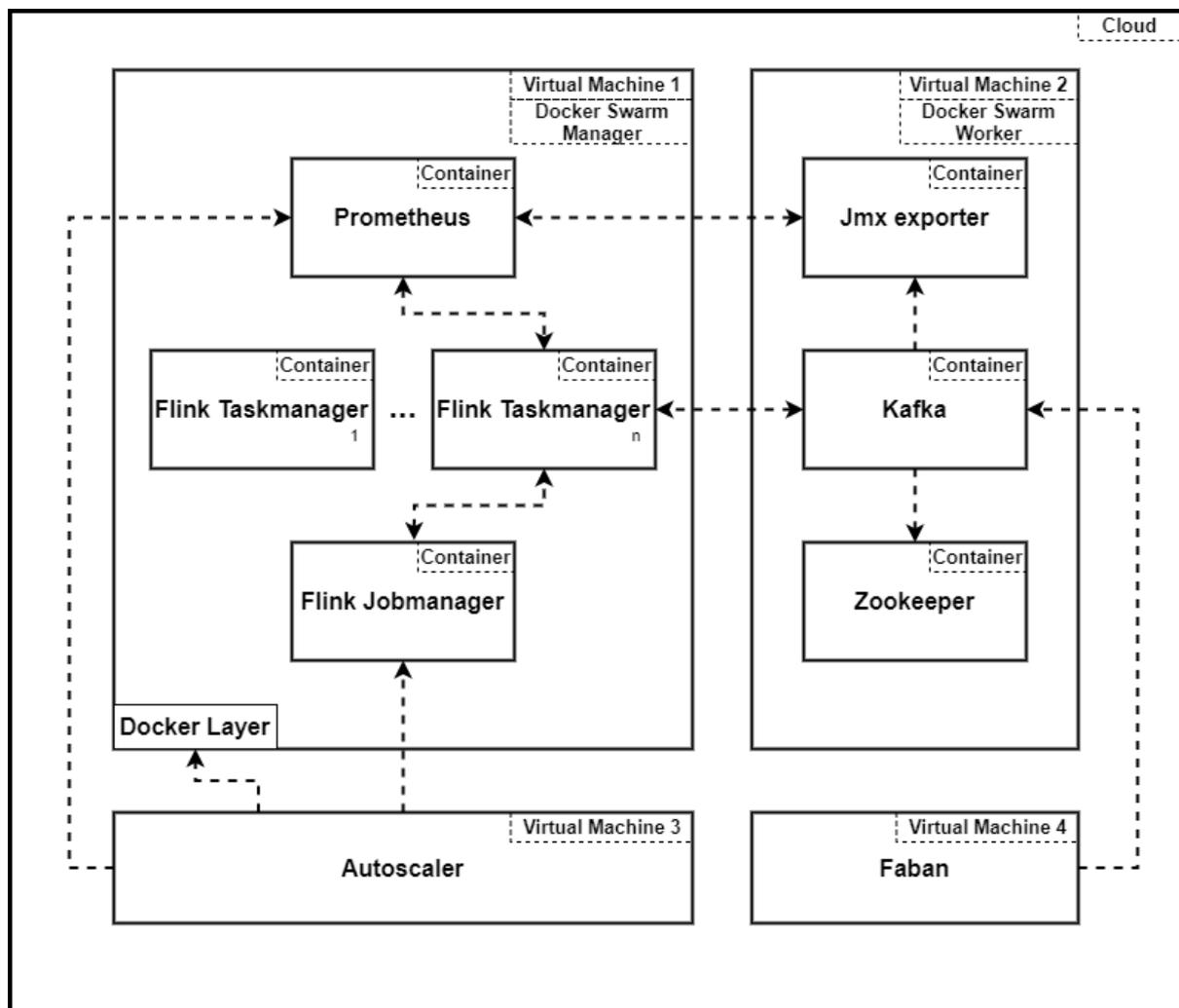


Figure 25 - Architecture of System Infrastructure

## Infrastructure Initialization

The Autoscaler and Faban, which run directly on virtual machines are installed manually. The other two virtual machines (VM1 and VM2), are created and configured by Docker Machine<sup>34</sup>. Docker Machine is a tool which allows the installation of Docker Engine on virtual hosts, referred as 'machines'. It also provides API to perform operations on the created machines, like running commands. The virtual machines form a Docker Swarm<sup>35</sup> cluster (an alternative could be Kubernetes<sup>36</sup>). In this way we are able to manage all containers from one machine, the Docker Swarm manager.

The static components of the infrastructure's architecture, that is all components except the TaskManagers, is described in a docker compose YAML file. The number of allocated TaskManagers varies over time and it is regulated by the Autoscaler. Docker Compose<sup>37</sup> is a tool for defining and running multi-container Docker applications. With Compose, a YAML file is used to configure the application's services. Then, with a single command, the services can be created and started from the configuration file. This file is built locally and its containers are pushed to Docker Hub. In this way, the virtual machines just download the required containers without building anything.

Afterwards, the docker compose file is copied to Docker Swarm manager which will deploy the infrastructure. The file also specifies the resources that each container can use and in which virtual machine will be placed.

## Resources

- Virtual Machine 1 (8 CPU and 16GB RAM)
  - Flink JobManager (1 CPU and 2 GB RAM)
  - Flink TaskManagers (1 CPU and 2 GB RAM each)
  - Prometheus (1 CPU and 1 GB RAM)
- Virtual Machine 2 (4 CPU and 8 GB RAM)
  - Kafka (2.5 CPU and 6 GB RAM)
  - Zookeeper (1 CPU and 1 GB RAM)
  - JMX exporter (0.1 CPU and 0.5 GB RAM)
- Virtual Machine 3 (4 CPU and 8 GB RAM)
  - Autoscaler
- Virtual Machine 4 (2 CPU and 4 GB RAM)
  - Faban

---

<sup>34</sup> <https://docs.docker.com/machine/overview/>

<sup>35</sup> <https://docs.docker.com/engine/swarm/>

<sup>36</sup> <https://kubernetes.io/>

<sup>37</sup> <https://docs.docker.com/compose/>

## 4.3. Components

### 4.3.1. Flink

The Flink cluster is formed by deploying Flink instances inside Docker containers. The instances can be configured to be JobManagers or TaskManagers. In our implementation we have only one JobManager and at least one running TaskManager. Although, the containers of the Flink cluster could be extended to more than one machines, in the current implementation the instances are constrained into one single virtual machine. This is due to container's communication issue among the Nodes of Docker Swarm which did not made the distribution of the Flink instance possible. To be more specific, TaskManagers were not able to connect to JobManager preventing the cluster formation.

#### Scaling

Currently, Flink (version 1.11 or older) doesn't support dynamic scaling of a running job. As a result, the only way to change the parallelism of a job is the following:

- Stop the job and take savepoint.
- Re-run the job with the new parallelism.

However, the community of Flink plans to add the re-scale functionality in a future version<sup>38</sup>.

#### Cluster Configuration

- The cluster is deployed as Flink Session Cluster<sup>39</sup>. In this way, the lifecycle of the running job is independent from the lifetime of the cluster. In other case, the cluster will shut down during the job's rescaling due to the need for restart.
- Each TaskManager has exactly one task slot. Which means that the number of allocated TaskManagers is identical to the parallelism of the job. As a result, we scale a job by modifying the number of TaskManager which are allocated to that particular job.
- For monitoring purposes, the metrics are exposed to Prometheus<sup>40</sup>.

#### Job configuration

We observed that the rescaling request via the Flink REST API could not overwrite the parallelism specified in source code of the job. In order to change the parallelism of the job by passing the value as argument, the operators' parallelism has not to be strictly specified in its source code. Otherwise, the passed parallelism is ignored and the scale is not performed.

---

<sup>38</sup> <https://issues.apache.org/jira/browse/FLINK-12312>

<sup>39</sup> <https://ci.apache.org/projects/flink/flink-docs-stable/ops/deployment/#session-mode>

<sup>40</sup> <https://prometheus.io/>

### Performance metrics

Flink offers a several metrics for the monitoring of running jobs. We found that three of them could provide us data that could effectively represent the performance of a job. The selected metrics are listed as follows:

1. **Latency tracking:** The sources of the job will periodically emit a special record, called a LatencyMarker. The latency markers are not accounting for the time user records spend in operators as they are bypassing them (they are not processed by the operators). If operators are not able to accept new records, thus they are queuing up, the latency measured using the markers will reflect that. This method requires clock synchronization in order to avoid false latency results. In addition, enabling latency metrics can significantly impact the performance of the cluster. Its use is recommended for debugging purposes only.
2. **BackPressure:** Backpressure refers to the situation where a job is receiving data at a higher rate than it can process during an increase of load. A back-pressure warning (e.g. High) for a task, means that it is producing data faster than the downstream operators can consume. Back pressure monitoring works by repeatedly taking back pressure samples of the running tasks. The sampling may affect the running job's performance
3. **Kafka Consumer records-lag-max:** Records lag is the calculated difference between a consumer's current log offset and a producer's current log offset. In other words, records lag shows how many records the consumer is behind from the latest produced one. Records lag max is the maximum observed value of records lag. An increasing value over time is an indication that the consumer group is not keeping up with the producers.

By comparing the aforementioned metrics, the records-lag-max is considered the best choice to monitor the performance of the job. It offers good enough accuracy and does not add further burden to cluster's load.

### 4.3.2. Prometheus

Prometheus is responsible for the monitoring of running applications. We measure the incoming workload by monitoring the Kafka topics which are used as a source by the running job. In order to retrieve metrics for a running job, every allocated TaskManager has to be monitored. Since the active TaskManagers change over time, the monitored targets have to be adjusted too. We perform such adjustment by updating a JSON file which contains the hostnames of the targeted TaskManagers. Each time the file is updated, Prometheus applies automatically the changes and the new targets are discovered. The length of the job's queue can be measured by monitoring the

---

TaskManagers' consumers. Specifically, we measure the total job's queue as the sum of the individual queues of each Taskmanagers allocated to the particular job.

### 4.3.3. Autoscaler

In Chapter 3 we analyzed the operation of the autoscaler and focused on the decision mechanism for scaling actions, abstracting from the interconnection to other components of the Infrastructure. In this section, we describe in detail how the agent's commands are actually implemented in the infrastructure. The autoscaler is implemented in Python 2.7 and it is responsible for the scaling of one running job.

The autoscaler has three main endpoints:

1. Prometheus REST API, for metrics retrieval.
2. Flink REST API, for job management.
3. An internal HTTP server with REST API, for resource adjustment on the Docker layer.

#### **Safety mechanisms**

The autoscaler uses the following safety mechanism to improve its operation:

##### **1. TaskManagers in hot standby**

The instantiation of a new TaskManager and its full connectivity to JobManager is a slow process. This kind of delay could prevent the controller to scale up instantly. We solve this issue by maintaining a small number of TaskManagers on standby. These instances are connected to Flink but they are not used by any job. When a standby TaskManager is allocated to a running job, a new one will be created to replace it.

##### **2. Discarding data during rescaling**

Due to the necessity of job's restart in the case of rescaling, the job needs several minutes to recover. In this case, the job stops consuming records. As a result, the incoming records remains to Kafka until the recovery of the job. When the job is running again, there is an accumulation of new records in Kafka which need to be consumed. In our experiments, the controller discards the metrics of the job until it is fully recovered. We observed that the job returns to its normal operation approximately after four minutes of delay from the rescaling action. Consequently, the controller waits four minutes in order the job to catch up with the incoming workload, and then continues its operation.

## HTTP server

Operations in docker layer, such as the creation or the removal of containers, are performed via an HTTP server. In this way, functions like container creation are performed in an asynchronous way and there is no need for delays in the controller's workflow. The HTTP server responses with success code as soon as a request is received and then attempts to execute the command. The server supports the following functions:

### 1. TaskManager creation

*Request:* POST /taskmanager

*Body:*

```
{
  "target":"integer"
}
```

*Function:* Creates  $n_{target}$  TaskManager containers. If the creation of the requested quantity exceeds the maximum number of total TaskManagers  $n_{max}$ , the HTTP server creates as much containers as the remaining space for new TaskManagers. If the maximum number of total TaskManagers containers is already reached, the request is ignored. The server takes care that IDs of the running containers are always in the range  $[1 \dots n_{max}]$ . As a result, the IDs are recycled instead of getting random or arbitrary values. The procedure is shown in Algorithm 8.

---

#### **Algorithm 8** Creation of Taskmanagers

---

```
1: procedure ADDTASKMANAGER( $n_{target}$ )
2:    $n_{max} \leftarrow \text{MaxNumberOfTaskmanagers}$ 
3:    $IDS_{allocated} \leftarrow \text{GetTotalTaskmanagers}()$ 
4:    $IDS_{all} \leftarrow [1 \dots n_{max}]$ 
5:    $IDS_{free} \leftarrow IDS_{all} \setminus IDS_{allocated}$ 
6:    $n_{free} \leftarrow \text{Length}(IDS_{free})$ 
7:   if  $n_{target} > n_{free}$  then
8:      $n_{target} \leftarrow n_{free}$ 
9:    $i \leftarrow 0$ 
10:  while  $i < n_{target}$  do
11:     $ID \leftarrow \text{Pop}(IDS_{free})$ 
12:     $\text{CreateTaskmanagerContainer}(ID)$ 
13:     $i \leftarrow i + 1$ 
```

---

## 2. TaskManager removal

*Request:* DELETE /taskmanager

*Body:*

```
{
  "used": "array",
  "free": "integer"
}
```

*Function:* Removes unused TaskManagers and leaves a number of TaskManagers idle. Idle TaskManagers are not allocated to a particular job, but remain available for future use. An array with the hostnames of the used TaskManagers is passed as argument in order to ensure that the removed TaskManagers are not allocated to the running job. The procedure is shown in Algorithm 9.

---

### Algorithm 9 Removal of Taskmanagers

---

```
1: procedure REMOVE_TASKMANAGER( $IDS_{used}, n_{free}$ )
2:    $IDS_{allocated} \leftarrow GetTotalTaskmanagers()$ 
3:    $IDS_{idle} \leftarrow IDS_{allocated} \setminus IDS_{used}$ 
4:    $n_{idle} \leftarrow Length(IDS_{idle})$ 
5:   if  $n_{free} > n_{idle}$  then
6:      $n_{free} \leftarrow n_{idle}$ 
7:    $i \leftarrow 0$ 
8:   while  $i < n_{idle} - n_{free}$  do
9:      $DeleteTaskmanagerContainer(IDS_{idle}[i])$ 
10:     $i \leftarrow i + 1$ 
```

---

## Scaling actions

The final form of the scaling actions includes resource management on both Docker layer and Flink Cluster. Note that the resources which are allocated to a job are only known to Flink Cluster. Docker Engine deploys TaskManagers instances but it has no knowledge whether an instance is used or it is idle. As a result, we first interact with the Flink Cluster in order to track which TaskManagers are active (e.g. they are receiving records for processing). A TaskManager is distinguished by its hostname (or container name). In addition, each running job is identified by a unique ID. Both TaskManager hostnames and job IDs are available to Prometheus. In this way, it is possible to get the allocated TaskManager to a specific job by querying Prometheus.

Afterwards we continue hierarchically to the next layer, which is the Docker one, so as to remove or add containers. Let's assume that there are five active TaskManagers to the cluster with hostnames TM1, TM2, TM3, TM4 and TM5, and we have a running job which has two allocated TaskManagers. We identify that TaskManagers with hostnames TM1 and TM4 are used by our job. If we want to remove a TaskManager instance, we can command the Docker Engine to remove the

container with hostname TM2 or TM3 (which we know that they are both idle). In this way, we ensure that we do not remove a used TaskManager and thus interfere to the normal operation of the job. On the other hand, the creation of a new TaskManager instance does not affect the operation of the job since the job's resources are already allocated.

In case of a Flink Cluster distributed among virtual machines, an extra resource layer is added, which manages the allocation/deallocation of virtual machines. In this case, we have to ensure that the swarm node to be removed, does not host any TaskManager containers used by a running job. Similarly, the cloud provider does not know in which machine, the used TaskManager containers are located. The Docker Engine is able to identify which nodes have idle Taskmanagers and thus be removed. The resources layers are depicted in Figure 26.

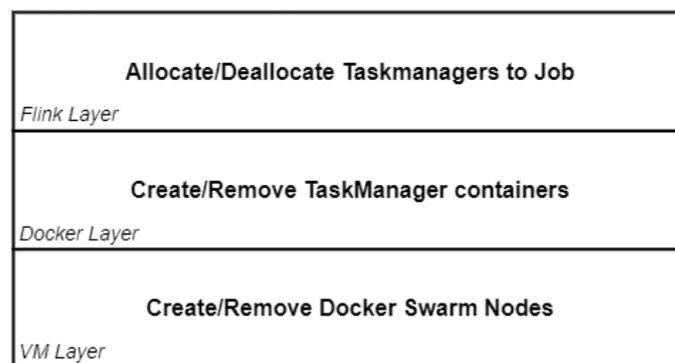


Figure 26 - Resource adjustment layers

The Algorithm 10 describes the function of the scaling actions. To be more specific, modification to Flink Cluster are performed through requests to Flink REST API, while changes to the Docker layer are done using the HTTP server. Both scaling actions change the parallelism by one level (adding one TaskManager or removing one TaskManager), but they could easily be extended to change more levels.

1. **Scale Up:** The first step is to check if the operation can be performed, which means that job has not reached the maximum parallelism yet. In other words, we check if the requested number of TaskManagers is available (e.g. due to resource exhaustion). Considering that  $n_{backup}$  TaskManagers are already deployed from a previous call, the required resources already exist to Flink Cluster. We scale up the running job, so the cluster remains with one less idle TaskManager than before the scaling action. Afterwards, the operation ensures that  $n_{backup}$  TaskManager remain idle by sending a request to HTTP server. The HTTP server will implement the function on the Docker layer by creating the required containers.
2. **Scale down:** Similarly, first we check if the job can be scaled down. If the job is already to the minimum parallelism, the operation is aborted. Afterwards, the job is scaled down, thus, the number of idle TaskManagers is increased. The

unnecessary TaskManagers are removed from the Flink Cluster by sending a request to the HTTP server. The server will let  $n_{backup}$  TaskManagers running and remove the rest.

---

**Algorithm 10** Flink job scaling
 

---

```

1: procedure SCALEUP
2:    $n_{max} \leftarrow \text{MaxNumberOfTaskmanagers}$ 
3:    $n_{current} \leftarrow \text{CurrentParallelism}$ 
4:    $n_{backup} \leftarrow 2$ 
5:   if  $n_{current} \geq n_{max}$  then return
6:    $\text{FlinkJobScaleUp}()$ 
7:    $\text{AddTaskmanager}(n_{backup})$ 
8: procedure SCALEDOWN
9:    $\text{IDs}_{used} \leftarrow \text{GetUsedTaskmanagers}()$ 
10:   $n_{current} \leftarrow \text{CurrentParallelism}$ 
11:   $n_{backup} \leftarrow 2$ 
12:  if  $n_{current} \leq 1$  then return
13:   $\text{FlinkJobScaleDown}()$ 
14:   $\text{RemoveTaskmanager}(\text{IDs}_{used}, n_{backup})$ 

```

---

#### 4.3.4. Kafka

Kafka is the source of records of the running job. The job consumes records from a specified topic. The producers send records to the job by targeting that particular topic. Each TaskManager that runs the job, has its own Kafka Consumer that receives a part of the workload.

In Flink, partitions are assigned to parallel task instances<sup>41</sup>. As a result, we have the following cases:

- When there are more Flink tasks than Kafka partitions, some of the Flink consumers will just remain idle, not reading any data. As a result, some TaskManagers does not process records and the workload of the job is not load balanced.
- When there are more Kafka partitions than Flink tasks, Flink consumer instances will subscribe to multiple partitions at the same time. In this case, all TaskManagers receive and process records.

Kafka does not support the dynamic decrease of the partition number of an existing topic. For this reason, we set a static number of partitions during the creation of the topic. We set the number of partitions to be at least equal with the maximum number

---

<sup>41</sup> <https://www.ververica.com/blog/kafka-flink-a-practical-how-to>

---

of Flink parallel tasks  $n_{max}$ , hence, the maximum parallelism of the Flink Cluster. In this way, all of the allocated TaskManagers to a job receive records for processing.

Generally, a Flink job can get input from multiple Kafka topics. Each topic could have different distribution of workload (the rate that new records enter the topic) from the others. In addition, the records of each topic could affect the utilization of TaskManagers in a different way. For example, the processing of records from one topic could require more computational power than the processing of another topic. In this way, multiple topics could require a more complex model, than the implemented one, to extract a pattern about the behavior of the application. As a result, creating a relation between workload per topic, allocated resources and performance could be more challenging. For this reason, the autoscaler of this work, can adjust only running jobs which use as a source only one topic. We could consider that the workload of the application is the sum of the individual workloads representing one single source. Although, we have not tested the efficiency of the autoscaler in practice for such a case.

### **JMX Exporter**

In order to monitor Kafka, a JMX agent is used. The direct connection between Kafka and Prometheus was not possible, so we use a JMX agent to retrieve the metrics. Java Management Extensions<sup>42</sup> (JMX) is a technology which enables managing and monitoring applications, system objects, devices, and service-oriented networks. The agent retrieves metrics from Kafka and makes them available by exposing a HTTP server. Prometheus retrieves the Kafka metrics by querying the HTTP endpoint of JMX exporter. In this way, we are able to monitor the workload of Kafka topics and thus the workload of the running job.

### **Zookeeper**

Zookeeper is essential for the function of Kafka<sup>43</sup>. It is also required by Flink in the High Availability mode. Although, in our implementation the High Availability mode is not enabled.

---

<sup>42</sup> [https://en.wikipedia.org/wiki/Java\\_Management\\_Extensions](https://en.wikipedia.org/wiki/Java_Management_Extensions)

<sup>43</sup> <https://medium.com/@logeesan/zookeeper-in-kafka-ce31b3dd55b1>

#### 4.3.5. Faban

Faban produces the workload of the System Under Test (SUT) by scheduling benchmarks. The benchmark is running a driver class which specifies where the workload should be sent and what operations should be performed. The benchmark creates Kafka producers which send records to the targeted application. The format of the records is dependent on the Flink's job. The records are filled with random data and then sent to the targeted Kafka topic. The distribution of the produced rate is controlled by the variation file which specifies the number of Faban active threads. Each thread creates a Kafka producer. By adding/removing Kafka producers it is possible to generate any desirable distribution.

Faban is deployed in a separate machine from the other components since it is not part of the system but simulates its client. In addition, the separation is essential for the right assessment of the system, Faban should not burden with its operation the rest of the components during the run of experiments.

At the beginning of the operation of benchmark with varying workload, Faban performs some checks according the given workload distribution. During those checks, a spike is produced which last a few minutes. Since this spike is not part of the workload distribution, the autoscaler is configured to ignore it.

## 5. Experiments

The experiments in this chapter assess the ability of the proactive Autoscaler (as described in Chapter 3) to dynamically adjust the resources of the running application in the Infrastructure described in Section 4.2.

### 5.1. Application: Clicks Fraud Detection

In the next experiments we evaluate the implemented Autoscaler by running a click fraud detection application<sup>44</sup>. Click fraud<sup>45</sup> is a type of fraud that occurs on the Internet in pay-per-click (PPC) online advertising. The owners of websites that post the ads are paid an amount of money determined by how many visitors to their sites click on the ads. Fraud occurs when a person, automated script, or computer program imitates a legitimate user of a web browser, by clicking on such an ad without having an actual interest in the target of the ad's link.

The application receives records from a Kafka topic which has the following JSON format:

```
{
  "ip": "205.0.44.187",
  "userID": "e61b8f7a-5029-433a-9e44-79d5f514d309",
  "timestamp": 1595431611,
  "eventType": "display/click"
}
```

Consequently, the production rate of these records represents the workload of our application.

The application attempts to capture these kinds of fraud by searching for three patterns:

- Pattern 1: Count the User IDs per unique IP address in a window of 60 seconds.
- Pattern 2: Count the IP addresses per unique User ID in a window of 60 seconds.
- Pattern 3: Calculate Click-Through Rate (CTR) per UID. In other words, is measures how many times an ad is clicked on, as compared to the number of times the ad is shown.

For each pattern the application stores the results into files.

---

<sup>44</sup> [https://github.com/Nada-S/Clicks\\_fraud\\_detection\\_with\\_Kafka\\_and\\_Flink](https://github.com/Nada-S/Clicks_fraud_detection_with_Kafka_and_Flink)

<sup>45</sup> [https://en.wikipedia.org/wiki/Click\\_fraud](https://en.wikipedia.org/wiki/Click_fraud)

## 5.2. Experiment: Exploration Mode

In this experiment we test the exploration mode of the proactive Autoscaler under a 5-hour workload which follows a gaussian distribution. The particular distribution begins from zero workload, smoothly increasing to its peak and then gradually returns to zero. In this way, the controller is able to explore all the desired parallelisms of the application on both scaling up and scaling down actions. The autoscaler was configured with the following parameters:  $\lambda = 0.05$  and  $k=250$  which are both parameters of Stability Check. Also, the controller created a Performance Model and performed Stability Check every 450 samples (~ 2 hours). The autoscaler switched to Optimal Control at the end of the particular experiment, so no observations are made for this mode.

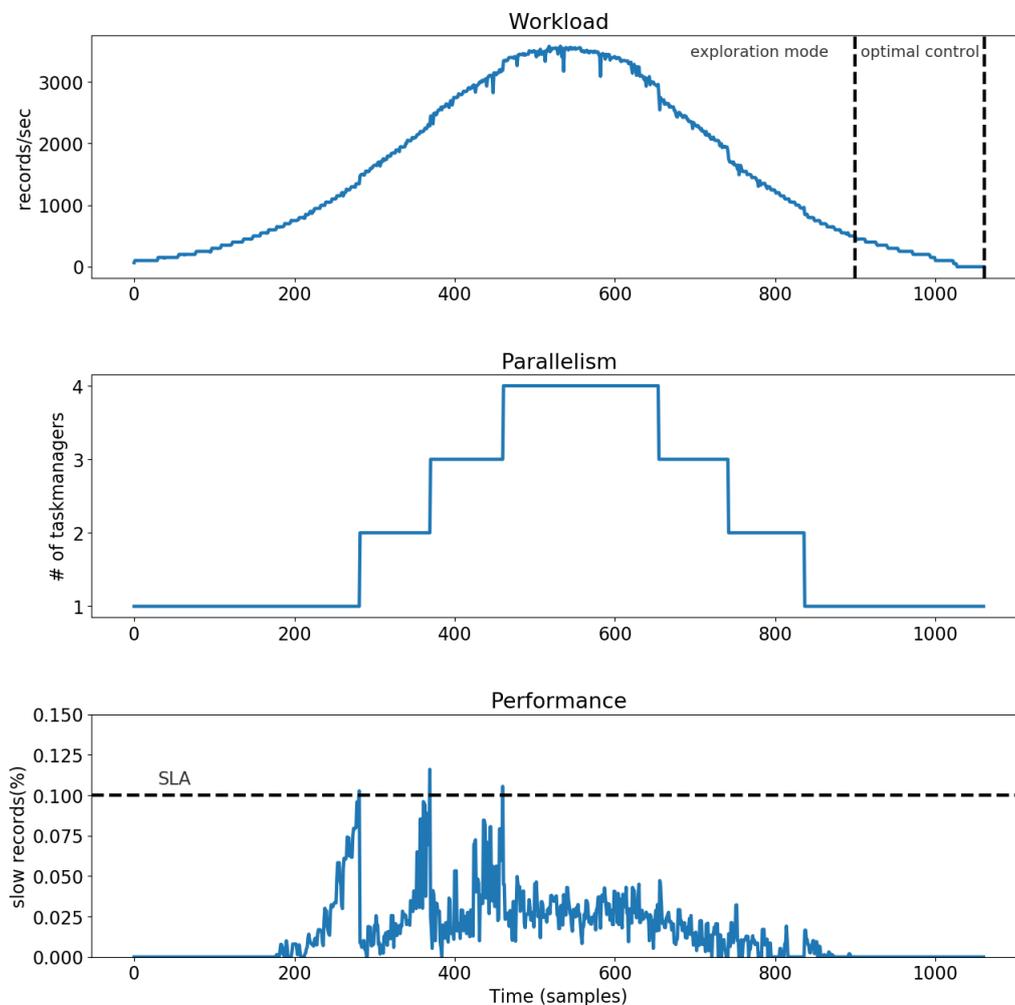


Figure 27 - Performance under gaussian distribution

---

In Figure 27 there are three diagrams that depict the state of the running application. All three diagrams have the same x-axis which is the number of samples. In exploration mode, a sample is taken every 15 seconds, which means that we retrieve metrics (workload and performance) every 15 seconds. The first diagram shows how the workload varies in time (how many records/second the application receives). The first diagram shows some abrupt changes in the distribution, since the autoscaler discards data after rescaling (in our case, data of the next 4 minute after rescaling occurred). This also applies to the next experiments. The second diagram shows the parallelism, which is the number of used TaskManagers, of the running application. The parallelism of the job is decided by the reactive scaler of the exploration mode according the SLA. The last diagram, depicts the performance of the application and whether the SLA is satisfied or not. We measure the performance as the percent of the incoming records that are not instantly served but remain in the queue. We consider that the SLA is violated when more than 10% of the incoming records remain in the queue. More formally, the SLA is defined as the 10<sup>th</sup> percentile of slow records. In the third diagram, SLA violation occur when the percentage of slow records is above the value 0.1 (10%) or the blue signal surpasses the black horizontal line.

We can observe that the controller under increasing workload is scaling up after SLA violation. This is in line with the addition of new TaskManagers in the second diagram (a new TaskManager in the second diagram is added each time an SLA violation occurs in the third diagram). That behavior was expected since the exploration mode works reactively. When the workload is decreasing, the controller gradually scales down. In this phase no SLA violation is observed. At the end of the experiment the controller switches to optimal.

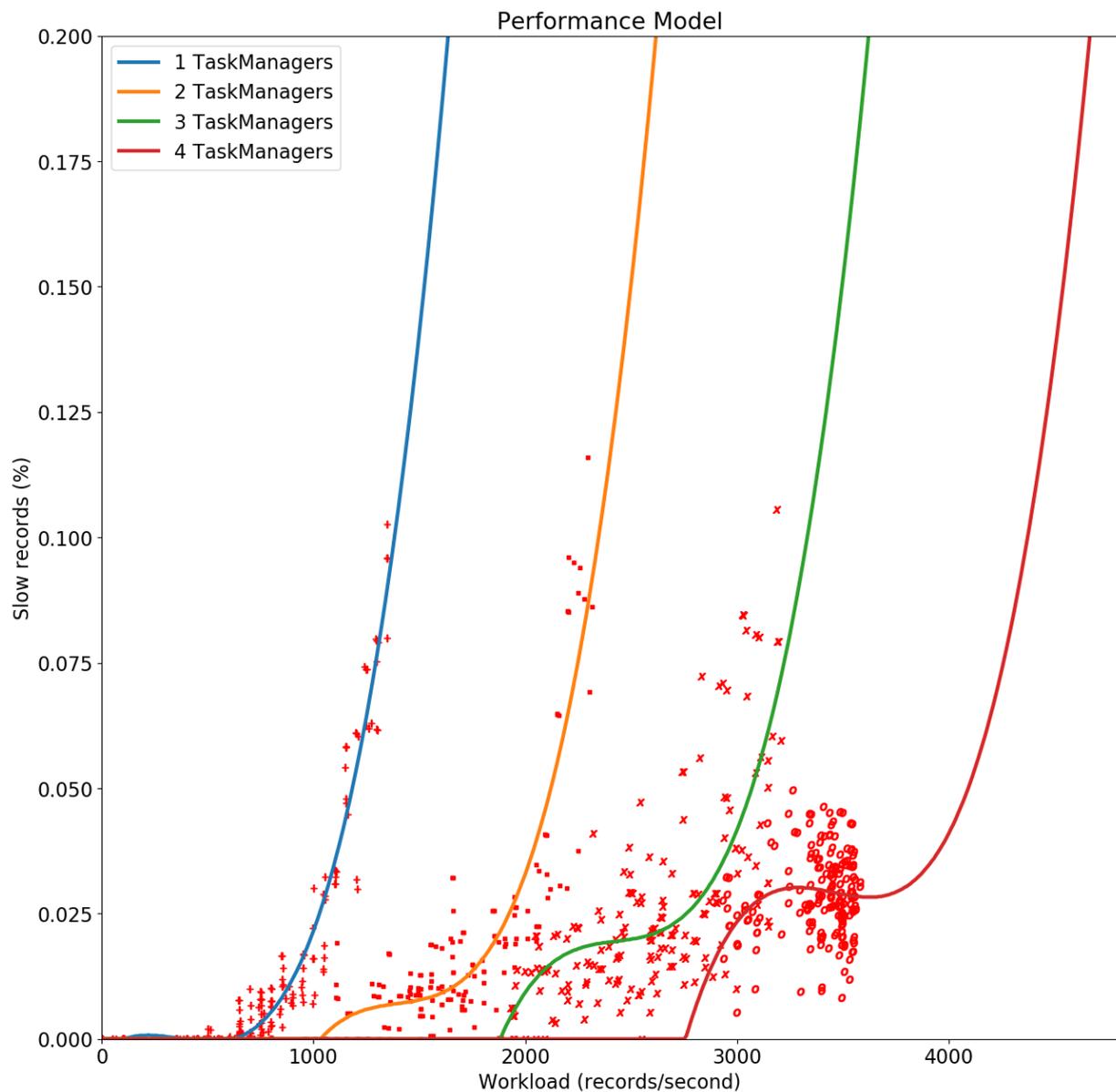


Figure 28 - Performance model created by gaussian distribution

Figure 28 illustrates the generated performance model. The performance model is the result of the exploration mode and utilize the collected data in order to describe the behavior of the application under different circumstances. Specifically, the performance model maps workload, number of Taskmanager and percentage of slow records. Generally, the representation of the performance model requires a three-dimensional space, since we have three variables. We simplify the representation in two dimensions by showing each parallelism separately with a different line. The x-axis depicts the incoming workload and the y-axis the percentage of slow records. The Performance Model is a function which gives the percentage of slow requests depending on workload and parallelism. For any incoming workload (in the range of 0 records/second up to 4500), we can select which parallelism (which line) is optimal (does not violate the SLA).

The points in the diagram represent the actual measurements of the controller during the exploration mode while the line of each parallelism show us the predicted performance given the workload. In other words, each line is a function which takes as input the workload and output the performance. Consequently, we have accomplished to create an accurate model which is able to describe and thus predict the behavior of the application

We can also observe that the model presents some fluctuations, mainly to the last two parallelisms. We consider that phenomenon occur for the following reasons:

- Flink consume rate: The consume rate of Flink does not keep up with the workload instantly but after a few seconds. In case of increase in the workload rate, we may observe a slight increase of the queue. Afterwards, when the consume rate keeps up with workload's rate, the queue returns to a decreased value again. As a result, we can observe temporal decrease of the slow records even when the workload is higher than before. This behavior is characteristic of Flink.
- Infrastructure's bug: As the workload increased and the controller allocated more Taskmanager to the application, we observed that the performance fall more quickly than expected. We consider this behavior is caused by a bug to our infrastructure or issue related to the running application.

### 5.3. Experiment: Optimal Control

In this experiment we study the operation of the optimal control of the Proactive Scaler under a rapidly changing workload. The particular workload lasts 1.5 hours. The controller is pre-trained with the collected metrics of the previous experiment. As result, the controller begins its operation in optimal control and uses the performance model of Figure 21. At the end, we compare the proactive mechanism with the reactive one. The autoscaler was configured with  $\alpha = 0.9$ ,  $\beta = 0.4$  (see page 44).

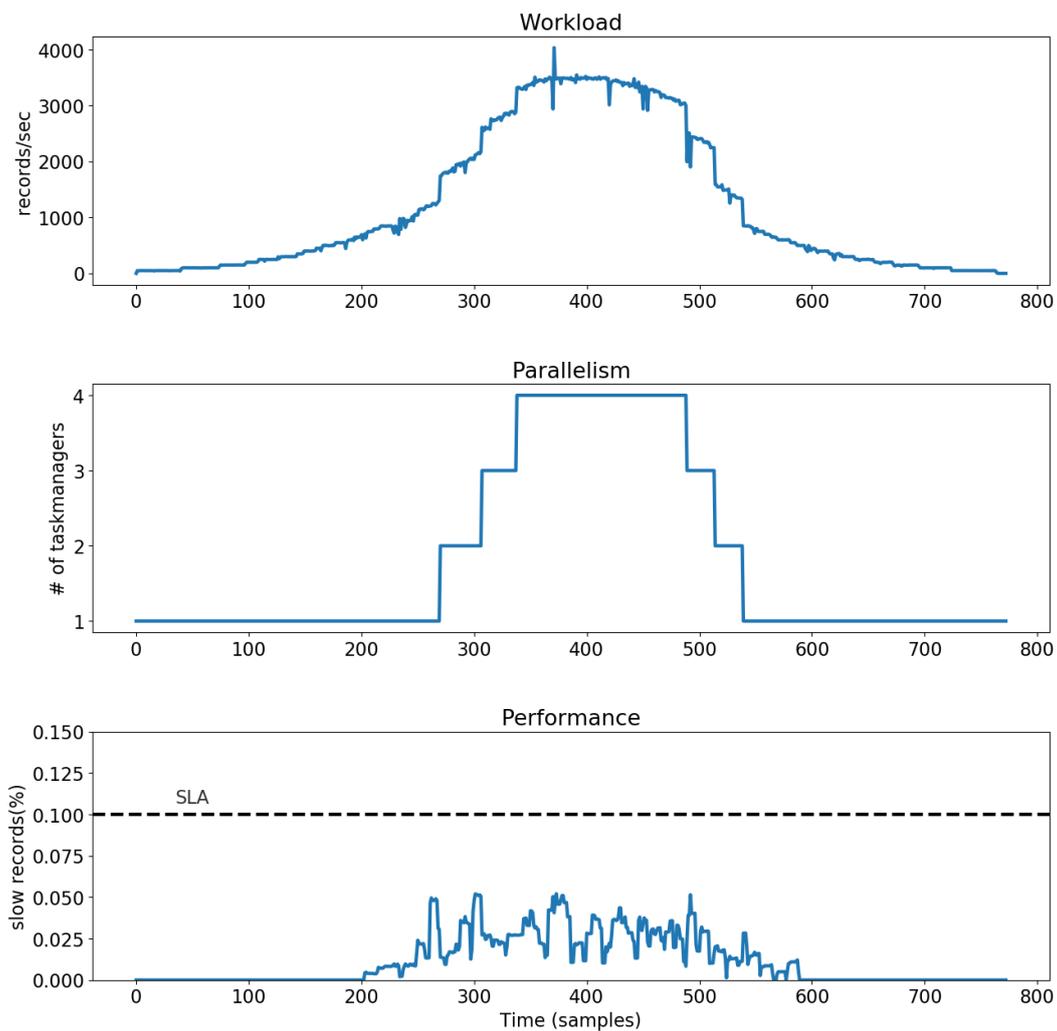


Figure 29 - Performance of optimal control under gaussian distribution with low variance

As we can see in Figure 29, the controller adds TaskManager in time and no SLA violations occurred.

The Figure 30 depicts the accuracy of the predictions during optimal control. The x-axis of all three diagrams correspond to the workload (number of samples). In optimal control, the controller takes samples of the system every 5 seconds. The first diagram compares the actual performance to the predicted one. The actual

performance is the percentage of slow requests that the client of the system observes. The predicted performance is the output of the Performance Model for the selected parallelism. In the second diagram, the residuals of the application's performance are depicted. The particular diagram shows how close the actual performance values are to the predicted ones. Each residual point is the difference between the predicted value and the corresponding actual one shown in the first diagram and it is the output of the formula:  $|actual\ performance - predicted\ performance|$ . When the residual points are close to zero, the Performance Model is considered accurate while increasing residual values indicate lack of accuracy. Consequently, we observe that the performance mode predicts the percentage of slow records accurately enough. The autoscaler was configured to switch to exploration mode if the residual points surpass the value 0.08 (or the prediction error is higher than 0.08). Although, the maximum difference of actual and predicted value is around 0.04 and thus no change point occurred.

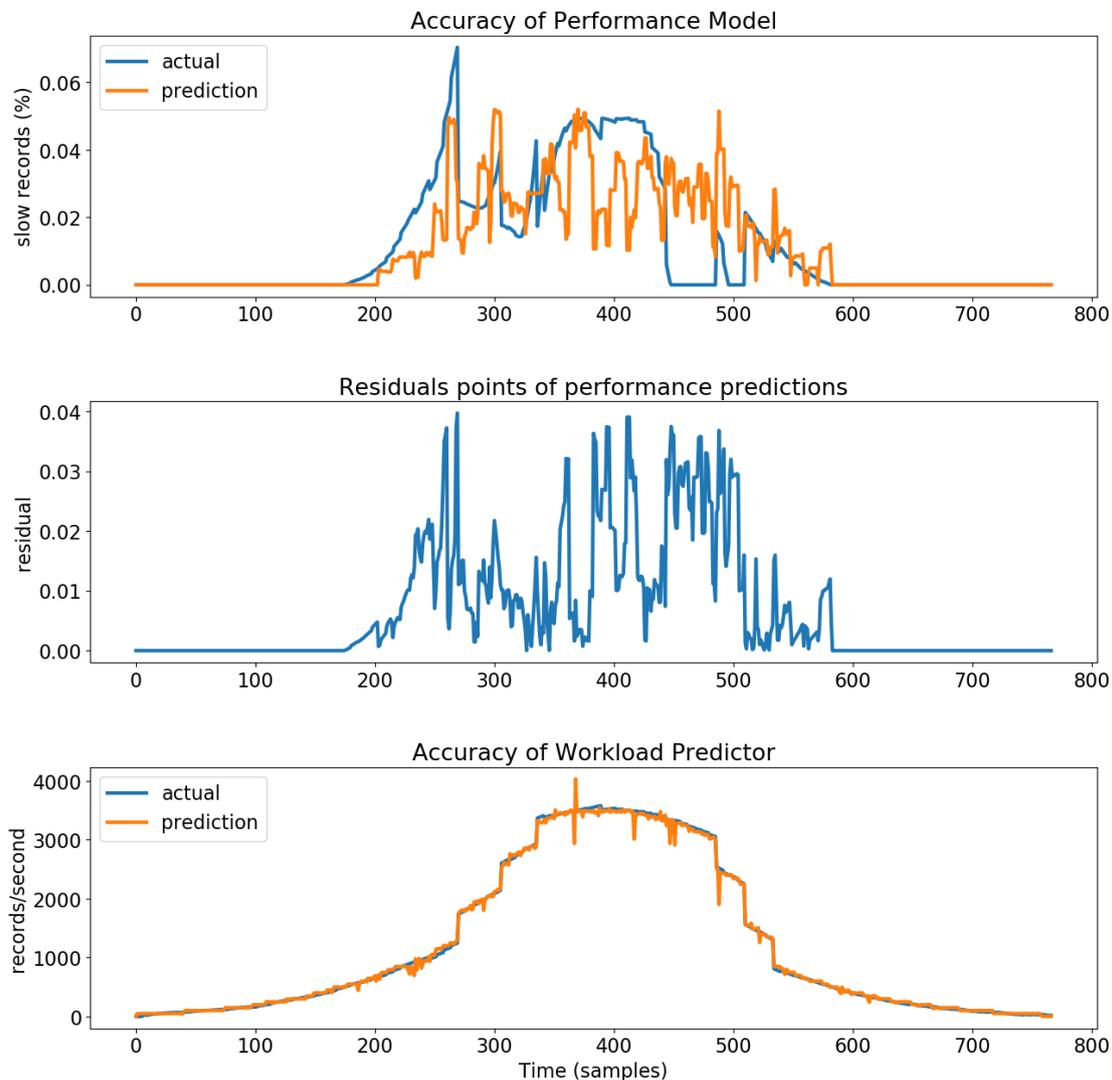


Figure 30 - Predictions of performance model under gaussian distribution with low variance

The Figure 30 depicts the predictions of the Workload Predictor. In this case, the predicted workload and the actual one is identical. The increased accuracy of the predictions of Workload Predictor was expected since the predictor is constantly adapted to the incoming workload (every 60 seconds). Since the predictions of Workload Predictor are input to the Performance Model, the accuracy of the first module affect the accuracy of the second one. Specifically, if the predicted workload is distant from the actual one, we would observe an increased difference to the actual and predicted performance. Although, we have not observed lack of accuracy to Workload Predictor in our experiments.

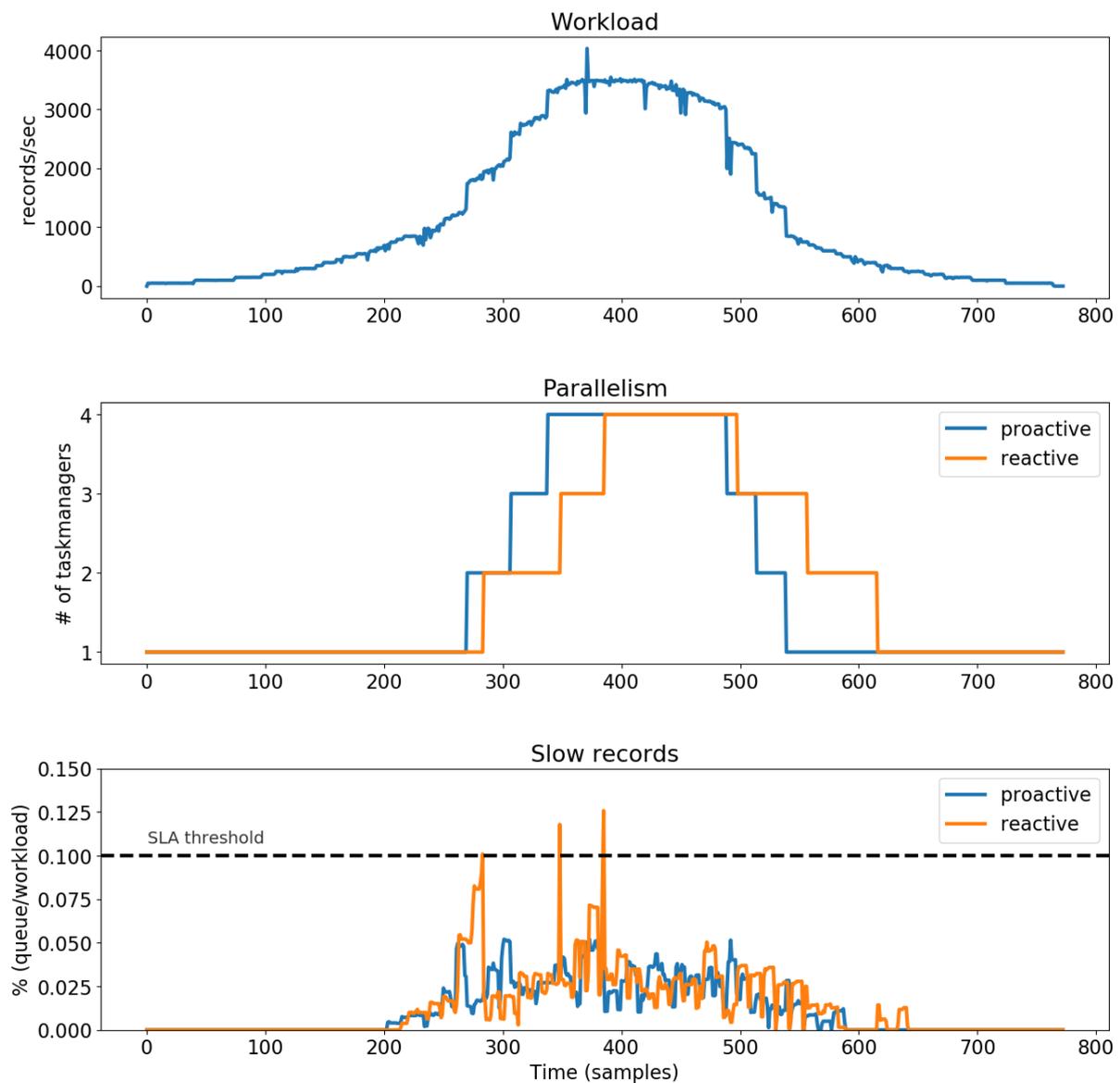


Figure 31 - Comparison of proactive and reactive scaler under gaussian distribution with low variance

In Figure 31 we compare the optimal control with the reactive mechanism in terms of performance and utilization. The first diagram of the figure represents the

workload distribution. The other two diagrams show the parallelism and the performance for each mechanism respectively. In the last two diagrams, we observe that the reactive scaler increases the parallelism of the application as soon as the SLA is violated, which is not the case of the proactive scaler. In this experiment, the reactive scaler allocates 1.84 Taskmanagers per sample (interval of 5 seconds) while the proactive scaler allocates 1.81 TaskManagers per sample. Consequently, in the current experiment, both mechanisms allocate approximately the same number of TaskManagers.

Although, the proactive scaler is more effective since no SLA violation occurred in the third diagram. In the second diagram we observe that the proactive scaler increases the parallelism sooner than the reactive one. In other words, the proactive scaler allocates the required number of TaskManagers, needed for the near future workload, beforehand. Similarly, in decreasing workload the proactive scaler removes idle resources sooner than the reactive. In this case, both mechanisms have the same performance but with different resources. The reactive scaler maintains more resources than those are needed, ending up under-utilized. In summary, the proactive scaler outperforms the reactive one and shows increased utilization of resources.

#### 5.4. Experiment: Synthetic Workload Distribution

In this experiment, we generated a 3-hour synthetic workload by analyzing a real workload from World Cup 98 Trace<sup>46</sup>. Although, only a part of the original workload distribution is shown to this experiment. The whole experiment could not be performed due to bug in the infrastructure. The purpose of the experiment is to analyze the whole operation of the controller which includes both exploration mode and optimal control. Also, we compare the proactive scaler with the reactive one. The autoscaler was configured with the following parameters:  $\lambda = 0.05$ ,  $\kappa = 250$ ,  $\alpha = 0.9$ ,  $\beta = 0.4$  and to perform stability check every 300 samples (40 minutes). The change point detection captures changes when the residual points are higher than 0.08. In this experiment the autoscaler performs a check (if a scaling action should be made) every 5 seconds in both exploration and optimal control. The same applies for the reactive scaler. Otherwise each scaler would generate different number of samples and the comparison sample to sample would not be feasible.

---

<sup>46</sup> <ftp://ita.ee.lbl.gov/html/contrib/WorldCup.html>

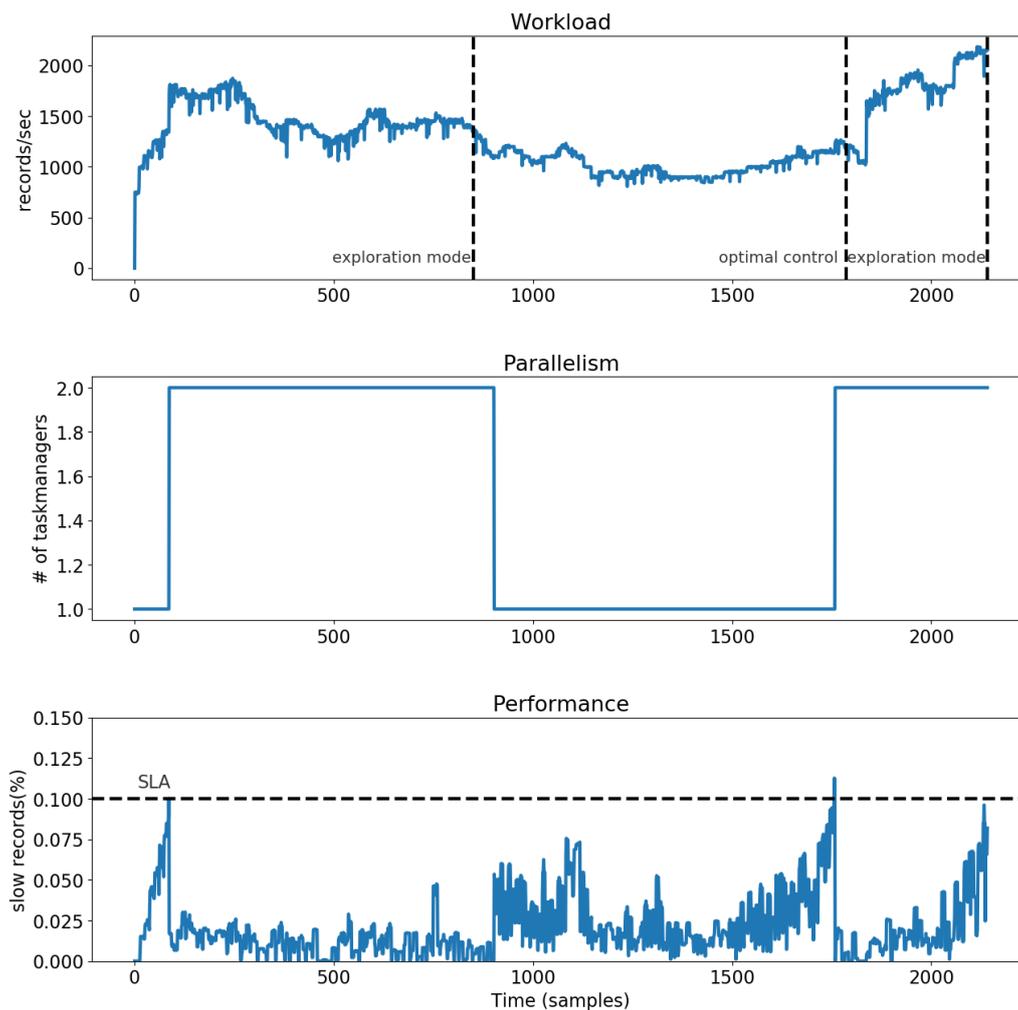


Figure 32 – Performance under synthetic workload distribution

In Figure 32, we observe the behavior of the actions and the performance of the application. The controller operates in exploration mode until the 800<sup>th</sup> sample. Afterwards, a Performance Model is created and the controller switches to optimal control until the 1700<sup>th</sup> sample. Finally, a change point was detected which switches the controller to exploration mode again.

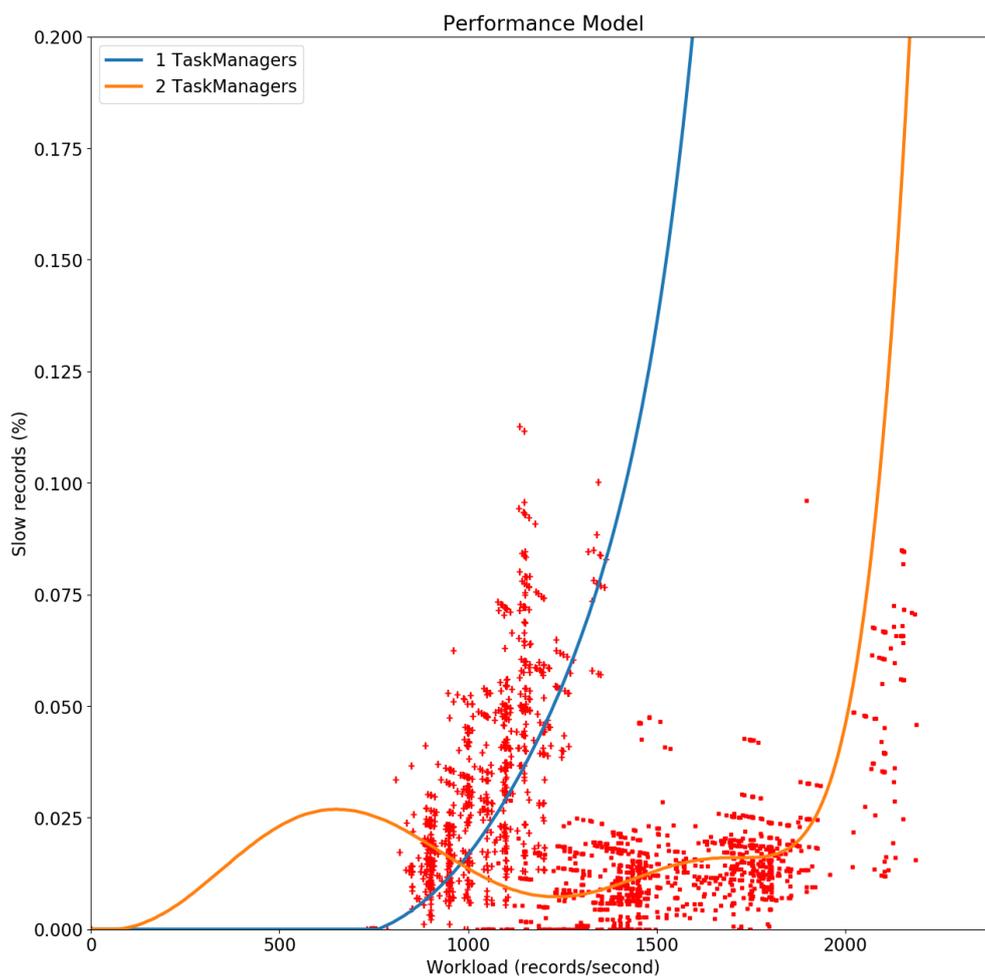


Figure 33 - Performance model created by synthetic workload

In Figure 33, the performance model of the controller is depicted. During the exploration mode, the controller collected data about the performance of the application (percentile of slow records) for the parallelisms 1 and 2.

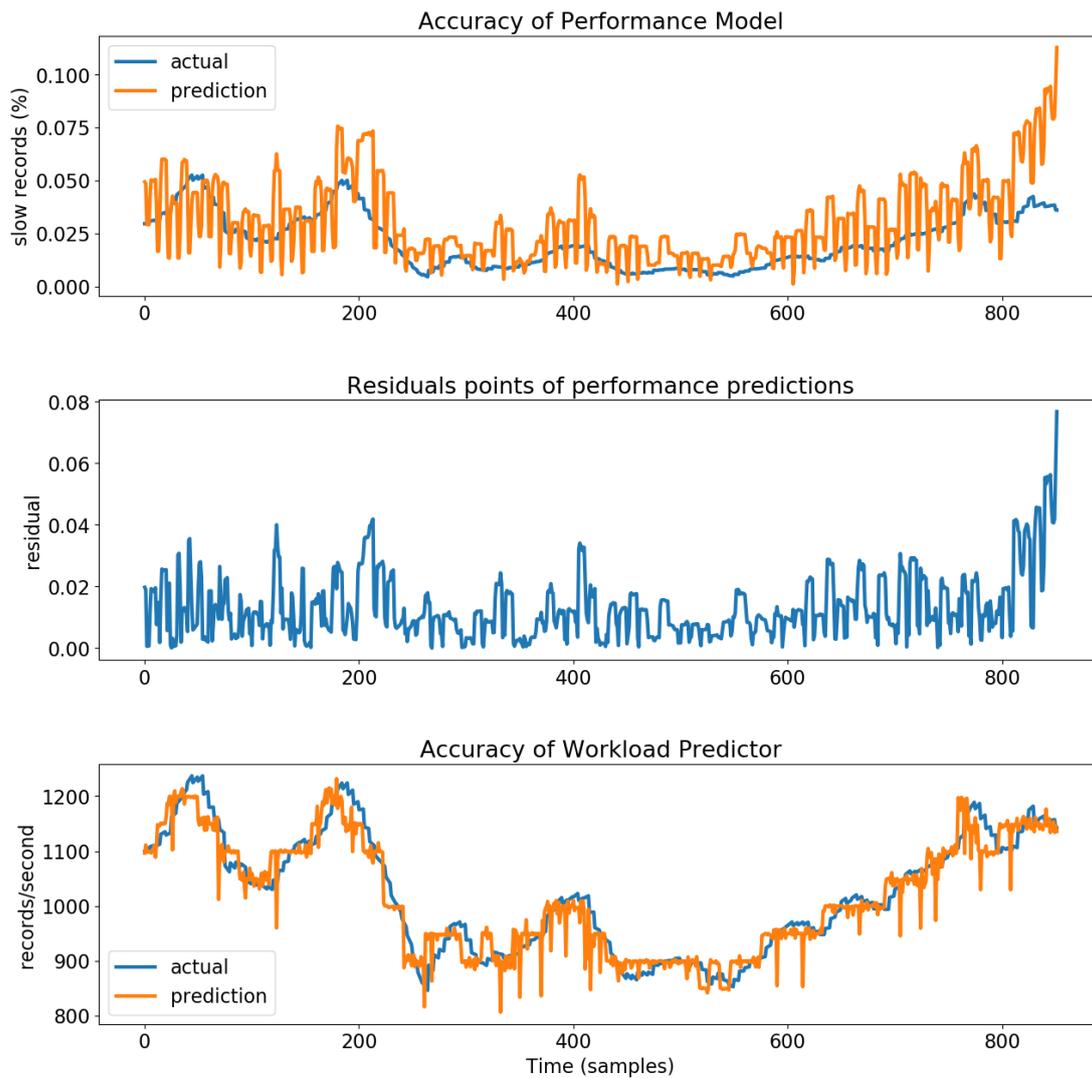


Figure 34 - Predictions of performance model under synthetic workload

In the first diagram of Figure 34, the predictions of the performance model are depicted. We observe that the model predicts the percentage of slow records accurately enough. The accuracy of the predictions falls at the last points (around 800<sup>th</sup> sample), since the predicted and actual values diverges in higher degree. This is also depicted in the residuals points which are more increased at the last points and finally reach the value 0.08. In this case, the online change point captures the lack of accuracy in the predictions and switches the controller to exploration mode. In the last diagram of the figure, we observe the accuracy of the Workload Predictor where the predicted workload is close to the actual one.

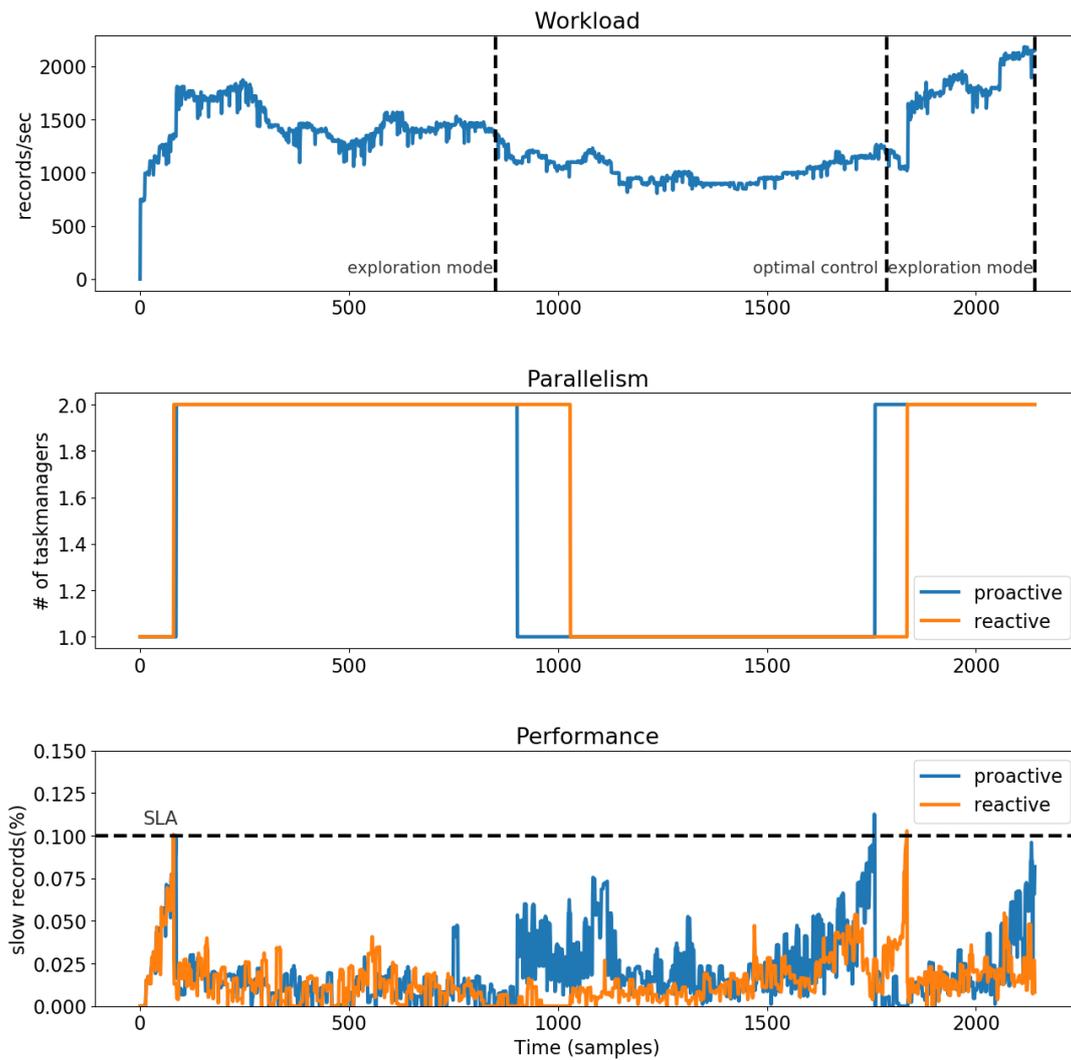


Figure 35 - Comparison of proactive and reactive scaler under synthetic workload in terms of application's parallelism and performance.

In Figure 35 we compare the proactive mechanism with the reactive one. At the beginning, both scalers behave in a similar way. This is expected because the controller is in exploration until the 800<sup>th</sup> sample and then switches to optimal. Afterwards, we observe that around the 900<sup>th</sup> sample the proactive scaler reduces to parallelism while the reactive performs the scale down later at the 1000<sup>th</sup> sample. We observe that at the 1700<sup>th</sup> sample during the operation of the proactive scaler, the performance gradually falls and violates the SLA. This phenomenon is not observed in the case of the reactive scaler. In addition, we observe that the workload around the sample 1700<sup>th</sup> is relatively steady and thus the performance fall was not expected. We consider that this phenomenon is an anomaly caused by an internal issue (e.g. crash) in the operation of the infrastructure. Finally, the proactive scaler returns to exploration mode until the end of the experiment. At this stage, both scalers behave similarly. During this experiment, the proactive scaler uses 1.56 TaskManagers per sample while the reactive scaler allocates 1.59 TaskManagers per sample.

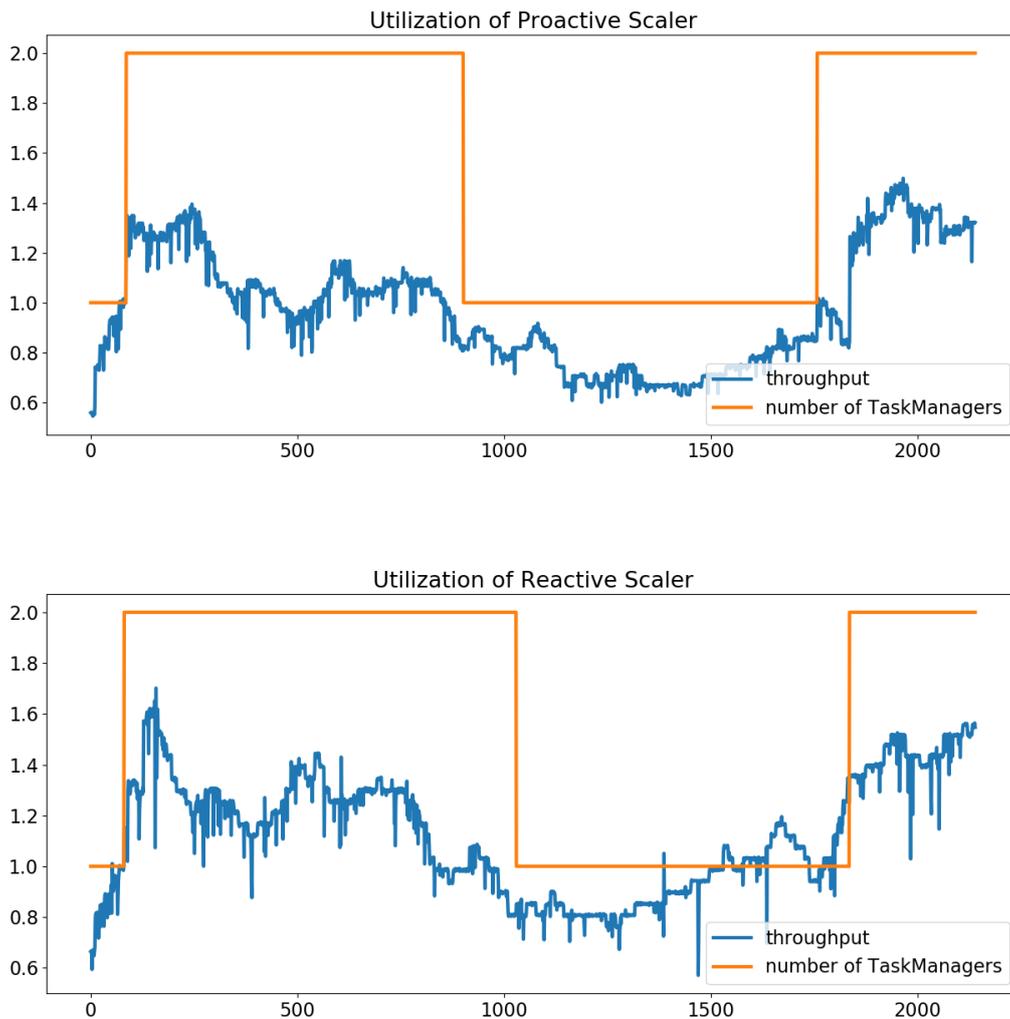


Figure 36 - Utilization of resources under synthetic workload

The Figure 36 depicts the throughput and the allocated TaskManagers for each scaler (proactive and reactive). The throughput of each scaler is calculated by the formula  $w_t / w_{max}$  where  $w_t$  the workload at time  $t$  and  $w_{max}$  the capacity of a single server (as described in Section 3.2). The throughput calculates approximately how many TaskManagers are required in order to handle the incoming workload with good performance.

We observe that in beginning of both diagrams (samples 0 to 100) the throughput gradually increases and then coincides with the number of allocated TaskManagers. In this case, both scalers (since the proactive operates in exploration mode) work at the maximum capacity of the allocated resources, and at some point, they surpass it. Afterwards, the system is over-utilized causing SLA violation in both cases (as shown in the same samples at Figure 35) and each scaler increases the parallelism. We also observe that in the reactive scaler's diagram of the Figure 36,

around the 1800<sup>th</sup> sample, the system is overutilized again which lead to another SLA violation (as depicted in the third diagram of Figure 35 around the 1800<sup>th</sup> sample).

Finally, at the second point where the proactive scaler violated the SLA, which is depicted at the third diagram in Figure 35 around the 1700<sup>th</sup> sample, the application does not show over-utilization in the corresponding sample of the Figure 36. This is another indicator that an anomaly to the system occurred at that moment.

## 5.5. Experiment: Autoscaler Load

The purpose of this experiment is to measure the resources which the Autoscaler uses in order to perform proactive scaling in the target system. We examine the resource usage in both exploration mode and optimal control. The measurement is performed by the Linux command “htop”<sup>47</sup>. The autoscaler runs on a machine with 4 CPU cores and 8 GB RAM.

### Exploration mode

We examine the exploration mode in two phases depending on the frequency of the operations. In the first phase, in which operations are executed every 15 seconds, the controller retrieves metrics and performs scaling actions. The second phase, which takes place every couple of hours, consists of the performance model training and the stability check.

#### Phase 1: Reactive Scaler

Typically, in this phase the scaler gets measurement by Prometheus and checks if an action should be made. The request rate to Prometheus is associated to the check frequency performed by the scaler. For example, in the first experiment the scaler checks if the SLA is violated every 15 seconds. As result, the scaler makes queries to Prometheus every 15 seconds. If a condition for scaling is satisfied (e.g. SLA violation), the scaler makes a request to FLINK REST API in order to change the parallelism. Although, such request does not follow a specific period but it is workload dependent. During the described part of operation, we observe the resource usage depicted in Figure 37:

- The CPU usage ranges between 0% and 2%.
- The allocated Memory is 141MB. This results from the RES (resident size) column, which is an accurate representation of how much actual physical memory a process is consuming.
- Disk read/writes occur rarely with the rate 2.58 KB/s. This behavior was expected since the collected data remain in memory and move to disk every couple of hours (e.g. every 8 hours).

Note that in this phase, the controller works reactively, so the required resources for the whole operation of a reactive scaler is the same.

---

<sup>47</sup> <https://htop.dev/>

```

1  [||||]                3.9%]   Tasks: 164, 679 thr; 2 running
2  [||||]                2.6%]   Load average: 1.01 1.08 1.22
3  [||||]                5.2%]   Uptime: 11:08:20
4  [||||]                1.3%]
Mem[|||||||||||||||||||||5.28G/7.71G]
Swp[|||||]              77.8M/7.63G]

```

PID	USER	PRI	NI	VIRT	RES	SHR	MEM%	CPU	CPUR	DISK WRITE	DISK READ	DISK R/W	TIME+	Command
12503	panagiotti	20	0	873M	141M	42104	1.8	1	1.3	2.58 K/s	0.00 B/s	2.58 K/s	0:03.45	python main

Figure 37 - General resource usage during exploration mode.

Phase 2: Performance Model training and Stability check

Periodically (e.g. every 1 hour) the controller examines if there are enough collected data to generate an accurate Performance Model. This step included the performance model training and the Stability Check.

- **Performance Model training:** This function searches which polynomial degree is the best fit for the collected data (in our case from degree 1 to 8) and creates a Performance Model. This step lasts about 5 seconds and we observe that the CPU usage is 118% while the allocated memory is 144MB out of 8GB. When the CPU usage is above 100% the process uses more than one core. The results are also depicted in Figure 38.
- **Stability Check:** In this case, we use bootstrap sampling to measure the standard deviation of the trained model. The parameter k, which regulates the number of the created samples, significantly affect the total duration of bootstrap’s execution and the resource usage. In our experiments we set k = 250 samples and thus the Stability check last about 60 seconds. We observe that the allocated memory is 144MB, the CPU usage is 104% and no disk IO operations are performed. The results are also depicted in Figure 39.

```

1  [|||||||||||||||||||||100.0%]   Tasks: 135, 493 thr; 2 running
2  [||||]                9.2%]   Load average: 1.02 0.96 1.03
3  [|||||]               13.5%]   Uptime: 01:24:26
4  [||||]                9.7%]
Mem[|||||||||||||||||||||3.45G/7.71G]
Swp[|||||]              0K/7.63G]

```

PID	USER	PRI	NI	VIRT	RES	SHR	MEM%	S	CPU%	CPUR	DISK READ	DISK WRITE	DISK R/W	TIME+	Command
10660	panagiotti	20	0	960M	144M	45404	1.8	R	118.	1	0.00 B/s	0.00 B/s	0.00 B/s	0:05.42	python mal

Figure 38 - Resource usage in exploration mode during Performance Model training.

```

1  [||||]                4.1%]   Tasks: 164, 688 thr; 2 running
2  [||]                 0.7%]   Load average: 2.94 3.58 3.19
3  [||]                 0.7%]   Uptime: 09:32:05
4  [|||||||||||||||||||||100.0%]
Mem[|||||||||||||||||||||5.69G/7.71G]
Swp[|||||]              79.8M/7.63G]

```

PID	USER	PRI	NI	VIRT	RES	SHR	MEM%	CPU	CPUR	DISK WRITE	DISK READ	DISK R/W	TIME+	Command
25934	panagiotti	20	0	952M	144M	44288	1.8	1	104.	0.00 B/s	0.00 B/s	0.00 B/s	1:04.03	py

Figure 39 - Resource usage in exploration mode during Stability Check.

## Optimal Control

The controller, in this stage uses the Workload Predictor to estimate the near future workload and the Performance Model to determine the optimal parallelism of the running job. These operations take place every 60 seconds and require 1 query to Prometheus. Although, the controller performs further computations during those 60 seconds in order to specify the accuracy of the performance model. Specifically, every 5 seconds we get the actual performance by querying Prometheus and then perform change point detection.

During the whole operation of Optimal Control, we observe the resource usage shown in Figure 40. Specifically, the CPU usage ranges between 0% and 3%, the allocated memory is 143MB and rare disk IO operations take place.

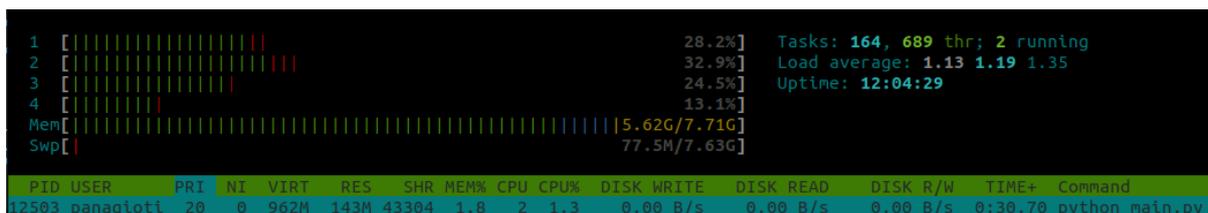


Figure 40 - Resource usage during Optimal Control.

Consequently, the frequent operations of the Autoscaler require minimal resources while the rare operations could use more than one CPU cores. As a result, we consider that an environment with 2 CPU cores and 2 GB RAM will be enough for the normal operation of the Autoscaler. Although, different configuration of the controller (e.g. different value for the parameter  $k$ ) would affect the overall resource usage.

---

## 6. Conclusions

In this thesis, we designed and implemented a proactive autoscaler for Apache Flink applications. Records can get in or out of Flink jobs through Apache Kafka topics. Also, Prometheus is used for monitoring of Flink jobs. The deployment took place on cloud technologies which provide us an easy way to handle the allocated resources. In this way, we are able to adjust the size of the Flink Cluster by adding or removing TaskManager instances. The proposed Autoscaler extends the capabilities of a Flink Cluster by providing dynamic scaling of resources to a running job according the workload. In this way, the running job uses as few resources as possible while preserving the performance in acceptable limits. Hence, the application provider is not charged for idle resources while the client does not experience high delays in the operation of the particular application.

The main idea of the autoscaler is based on publications [1] and [2] but is adjusted to be compatible to Flink and streaming applications. The autoscaler begins its operation by collecting metrics of the targeted application while preserving the performance using a reactive mechanism. The targeted application is analyzed in terms of performance, workload and allocated resources, so as to export a pattern about its behavior. This is done by training a statistical machine learning model which captures the relation between workload, number of TaskManagers and performance. Afterwards, the generated machine learning model is utilized in order to act proactively and made optimal decisions. The autoscaler is also capable to detect and adopt to changes in application itself or its environment. The scaling actions, decided by the autoscaler, are performed in two stages. The number of allocated TaskManagers is determined according the incoming workload and then implemented by the Docker Engine in the form of containers.

Last but not least, we run a variety of experiments in order to assess the efficiency of the autoscaler using both known (i.e. gaussian) and real distribution. The implemented controller proved to be capable for predicting the performance of a running Flink with good accuracy and thus act proactively. We also compared the proactive autoscaler to a reactive one in order to prove the superiority of such proactive mechanism. To be more specific, the proactive mechanism shown decreased number of SLA violations and increased utilization of resources. At the end, we measured the resource usage of the autoscaler and concluded that a medium flavor machine could be enough to handle its requirements.

## 7. Future work

The current work was been done within specific time limits, as a result, there is room for some issues as future work. The operation of the controller could be improved by addressing the following issues:

1. **Detecting gradual changes:** Currently, the used change point detection is unable to capture changes that take place gradually. If the difference between the predictions of performance model and the actual values is increased slowly, the controller will remain in optimal control. This issue could be addressed by introducing a method which captures such changes like twin comparison.
2. **Data selection:** During the operation of the application, a temporary change in the system could create outliers<sup>48</sup>. For instance, a slight crash of component, which occur rarely, could cause an instant but temporary performance drop. An outlier diverges from similar observations and can cause serious problems in statistical analysis. As a result, a few outliers in the dataset could dominate the training of the performance model resulting to lack of accuracy. Excluding data points, which does not describe the overall behavior of the application, could significantly improve the predictions of the performance model. Such a method is Outlier Detection.
3. **Defining optimal hysteresis gains:** In the current implementation of the autoscaler, the gains  $a, \beta$  are static and user-defined. The efficiency of the optimal control could be increased by defining the optimal parameters according the incoming workload. A method which provided solution to this kind of problem is a policy search algorithm [2].
4. **Rescaling on operator granularity:** Currently, a rescaling action modifies the parallelism of all operators of the job. For example, in case of SLA violation, every operator will be scaled up. We could improve the overall resource utilization of the job, by adjusting each operator separately according its needs. Such a rescaling methodology, which works reactively, is described in DS2 [7].
5. **Supporting jobs with multiple Kafka topics:** The current form of the autoscaler is designed to monitor jobs which use a single Kafka topic as a source. Although, a job could consume data from multiple Kafka topics. This could be achieved by introducing a more sophisticated performance model.

---

<sup>48</sup> <https://en.wikipedia.org/wiki/Outlier>

---

## 8. References

- [1] Bodík, P., R. Griffith, Charles Sutton, A. Fox, Michael I. Jordan and D. Patterson. "Automatic exploration of datacenter performance regimes." *ACDC '09* (2009).
- [2] Bodík, P., R. Griffith, Charles Sutton, A. Fox, Michael I. Jordan and D. Patterson. "Statistical Machine Learning Makes Automatic Control Practical for Internet Datacenters." *HotCloud* (2009).
- [3] Arabnejad, Hamid, C. Pahl, Pooyan Jamshidi and G. Estrada. "A Comparison of Reinforcement Learning Techniques for Fuzzy Cloud Auto-Scaling." *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)* (2017): 64-73.
- [4] Benifa, J. V. Bibal and Dharma Deje. "RLPAS: Reinforcement Learning-based Proactive Auto-Scaler for Resource Provisioning in Cloud Environment." *Mobile Networks and Applications* (2018): 1-16.
- [5] Rossi, Fabiana, M. Nardelli and V. Cardellini. "Horizontal and Vertical Scaling of Container-Based Applications Using Reinforcement Learning." *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)* (2019): 329-338.
- [6] Baresi, L., S. Guinea, A. Leva and G. Quattrocchi. "A discrete-time feedback controller for containerized cloud applications." *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2016): n. pag.
- [7] Kalavri, Vasiliki, 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." *OSDI* (2018).
- [8] Alexiou, Michail S. and Euripides G. M. Petrakis. "Elixir: An Agent for Supporting Elasticity in Docker Swarm." *AINA* (2020).
- [9] Pelluri, Sudha and Keerti Bangari. "SYNTHETIC WORKLOAD GENERATION IN CLOUD." *International Journal of Research in Engineering and Technology* 04 (2015): 56-66.
- [10] Yamanishi, Kenji and Jun'ichi Takeuchi. "A unifying framework for detecting outliers and change points from non-stationary time series data." *KDD '02* (2002).