Cost-efficient enactment of stream processing topologies in public clouds utilizing container technologies

Christoph Hochreiner Corresp., [1] , **Michael Vögler** [2] , **Stefan Schulte** [1] , **Schahram Dustdar** [1]

[1] Distributed Systems Group, TU Wien, Vienna, Austria
[2] TU Wien, Vienna, Austria

Corresponding Author: Christoph Hochreiner
Email address: c.hochreiner@infosys.tuwien.ac.at

The continuous increase of unbound streaming data poses several challenges to established data stream processing engines. One of the most important challenges is the cost-efficient enactment of stream processing topologies under changing data volume. These data volume pose different loads to stream processing systems whose resource provisioning needs to be continuously updated at runtime. First approaches already allow for resource provisioning on a virtual machine based level, but this only allows for coarse resource provisioning strategies. Based on current advances and benefits for containerized software systems, we have designed a cost-efficient resource provisioning approach and integrated it into the runtime of the Vienna Ecosystem for Elastic Stream Processing. Our resource provisioning approach maximizes the resource usage for virtual machines obtained from cloud providers while at the same time minimizing the number of reconfigurations for the enacted topology. The evaluation shows that our approach leads to a cost reduction of 12% compared to techniques presented in our previous publication while maintaining the same level of quality of service.

Cost-efficient enactment of stream processing topologies in public clouds utilizing container technologies

Christoph Hochreiner[1], Michael Vögler[2], Stefan Schulte[1], and Schahram Dustdar[1]

[1]**Distributed Systems Group, TU Wien, Vienna, Austria**
[2]**TU Wien, Vienna, Austria**

ABSTRACT

The continuous increase of unbound streaming data poses several challenges to established data stream processing engines. One of the most important challenges is the cost-efficient enactment of stream processing topologies under changing data volume. These data volume pose different loads to stream processing systems whose resource provisioning needs to be continuously updated at runtime. First approaches already allow for resource provisioning on a virtual machine based level, but this only allows for coarse resource provisioning strategies. Based on current advances and benefits for containerized software systems, we have designed a cost-efficient resource provisioning approach and integrated it into the runtime of the Vienna Ecosystem for Elastic Stream Processing. Our resource provisioning approach maximizes the resource usage for virtual machines obtained from cloud providers while at the same time minimizing the number of reconfigurations for the enacted topology. The evaluation shows that our approach leads to a cost reduction of 12% compared to techniques presented in our previous publication while maintaining the same level of quality of service.

Keywords: Cloud Computing, Data Stream Processing, Resource Elasticity, Resource Optimization

1 INTRODUCTION

Due to the transition towards a data-centric society, today's stream processing engines (SPEs) need to deal with a continuous increase of unbound streaming data regarding volume, variety, and velocity (McAfee et al., 2012). Currently, this growth in data is mainly driven by the advent of the Internet of Things (IoT)[1]. Sensors, which represent a vital part of the IoT, emit a huge volume of streaming data that needs to be processed to provide additional value to users or to trigger actions for IoT devices or other services, e.g., handling user notifications. Furthermore, many scenarios call for data processing in near real-time, which requires the application of SPEs like System S (Gedik et al., 2008), Apache Storm (Toshniwal et al., 2014), Heron (Kulkarni et al., 2015), or Apache Spark (Zaharia et al., 2010). State-of-the-art SPEs provide the user with an extensive set of APIs to design and enact stream processing topologies. These topologies represent a choreography of different stream processing operators, like filters, transformations, or other operations, which are required to process data (Gedik et al., 2008).

Although SPEs are highly efficient regarding data processing, they struggle with varying volumes of data over time (Hochreiner et al., 2015). Because most SPEs operate on a fixed amount of computational resources, e.g., on clusters, they cannot adapt to changes of the data volume at runtime (Hochreiner et al., 2016a). One solution for this issue is the over-provisioning of computational resources so that the SPE can process any amount of incoming data while complying with given Service Level Agreements (SLAs). While this approach ensures a high level of SLA compliance, it is not cost-efficient because the provisioned computational resources are not used most of the time. The more economically feasible approach to this challenge is under-provisioning, where an SPE is equipped with computational resources to cover most of the incoming data scenarios. However, in the case of underprovisioning, the SPE may cover most scenarios, but it may also violate SLAs in some high load scenarios, due to a delay in the data processing.

[1]http://www.gartner.com/newsroom/id/3165317

Based on the Cloud Computing paradigm (Armbrust et al., 2010), a more promising provisioning approach, namely elastic provisioning for stream processing systems, emerged in recent years (Satzger et al., 2011; Gedik et al., 2014; Heinze et al., 2015; Lohrmann et al., 2015; Xu et al., 2016). This approach allows the SPE to lease computational resources on-demand whenever they are required. Resources can be released again as soon as they are not needed anymore. This approach allows for the cost-efficient enactment of stream processing topologies while maintaining high SLA compliance (Hochreiner et al., 2016a). Up to now, most elastic provisioning approaches only consider virtual machines (VMs) as the smallest entity for leasing and releasing of computational resources. This approach is perfect applicable for private clouds, where the only objective of resource provisioning algorithms is resource-efficiency, and there is no need to consider any billing aspects or Billing Time Units (BTUs). A BTU defines the minimum leasing duration for computational resources, e.g., VMs, and often amounts to one hour like on Amazon EC2[2]. The concept of the BTU means that the user has to pay for each started hour, regardless of how many minutes the VM is used. Because of the BTU, the repeated leasing and releasing of VMs may result in even higher cost than an over-provisioning scenario (Genaud and Gossa, 2011), because releasing a VM before the end of the BTU results in a waste of resources.

To address this shortcoming, this paper considers an additional resource abstraction layer on top of the VMs, to allow for more fine-grained elastic provisioning strategies with the goal to ensure the most cost-efficient usage of the leased resources while respecting given SLAs. This additional layer is realized by applying the recent trend towards containerized software components, i.e., containerized stream processing operators. The containerization provides several advantages regarding deployment and management of computational resources. Besides the smaller granularity compared to VMs, containerized stream processing operators also allow for a faster adoption of the stream processing topology on already running computational resources. An additional layer of containers also enables reusing already paid computational resources, i.e., resources can be utilized for the full BTU. Today, frameworks like Apache Mesos (Hindman et al., 2011) or Docker Swarm[3] provide the functionality to deploy containerized applications on computational resources. However, these frameworks rely on simple principles like random deployment, bin-packing, or equal distribution to deploy containers across multiple hosts. QoS aspects are not taken into account. Furthermore, the frameworks are optimized to operate on static computational resource configurations and do not consider the resource elasticity aspect of the cloud computing paradigm.

In this paper, we leverage containerized stream processing operators and propose an elastic resource provisioning approach which ensures an SLA-compliant enactment of stream processing topologies while maximizing the resource usage of computational resources and thus minimizing the operational cost for the topology enactment. To demonstrate the feasibility of our solution, we integrate our proposed approach in the Vienna Ecosystem for Elastic Stream Processing (VISP) (Hochreiner et al., 2016b) and evaluate it based on a real world scenario from the manufacturing domain. The results of our evaluation show that our approach achieves a cost reduction of about 12% compared to already existing approaches while maintaining the same level of quality of service.

The remainder of this paper is structured as follows: First, we provide a motivational scenario, discuss the system architecture and present the derived requirements in Section 2. Based on these requirements we then provide the problem definition for the optimization problem in Section 3, which leads to our optimization approach presented in Section 4. In Section 5, we present our evaluation setup and in Section 6 we present the evaluation results and their discussion. Section 7 provides an overview on the related work, before we conclude the paper in Section 8.

2 MOTIVATION

2.1 Motivational Scenario

In the following paragraphs, we describe a data stream processing scenario from our EU H2020 project *Cloud-based Rapid Elastic Manufacturing* (CREMA) (Schulte et al., 2014). Figure 1 shows a stream processing topology, which is composed of nine different stream processing operator types (O1 – O9) that process the data originating from three different sources (S1, S2, S3). Each of the operator types performs a dedicated operation to transform the raw data from manufacturing machines into value-added and human-

[2]`https://aws.amazon.com/ec2/pricing/`
[3]`https://docs.docker.com/swarm/`

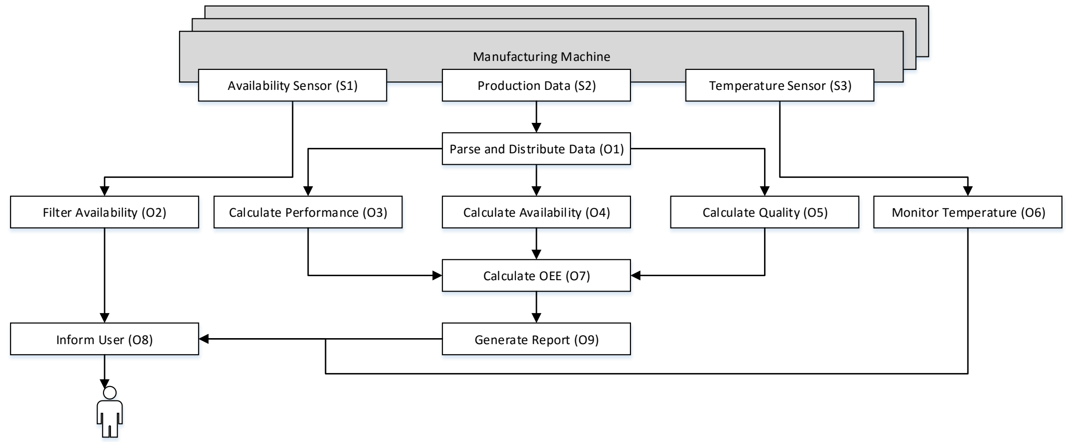


Figure 1. Stream Processing Topology from the Manufacturing Domain

readable information. The information from the data sources is used to monitor three different aspects, like the availability of the manufacturing machines or the machine temperature to avoid overheating of the machines and assess their Overall Equipment Effectiveness (OEE). In this scenario, we have two different types of data sources. The first type of data source are sensors, i.e., S1 and S3, which emit machine-readable data and can be directly accessed via an API. The second type of data, e.g., S2, is a video feed, which scans a display of the manufacturing machines because some information is not directly accessible via an API. This information needs additional preprocessing to transform the data into machine-readable data.

The *Availability Sensor* (S1) emits the current status, i.e., available, defect or planned downtime, of the manufacturing machine every two seconds. This information is then filtered by the *Filter Availability* (O2) operator, which generates warnings for each new downtime incident of a specific manufacturing machine. The warning is then forwarded to the *Inform User* (O8) operator, which informs a human supervisor of the machines.

The second data source is the *Production Data* (S2), which is obtained by a video stream, i.e., an image taken every ten seconds. This image contains different production related information, such as the amount of produced goods and needs further processing, e.g., by Optical Character Recognition (OCR), to extract machine-readable information. The *Parse and Distribute Data* (O1) operator distributes the information to the three operators O3, O4, O5 that calculate the different components of the OEE value. These individual components are then united by the *Calculate OEE* (O7) operator and then forwarded to the *Generate Report* (O9) operator, which generates a PDF-report every minute. This report aggregates the information of all monitored machines and is forwarded once every minute to the *Inform User* (O8) operator.

The *Temperature Sensor* (S3) emits the temperature twice every second. This information is processed by the *Monitor Temperature* (O6) operator, which triggers a warning whenever the temperature exceeds a predefined threshold. This warning is then also forwarded to the *Inform User* (O8) operator to inform the human supervisor.

Due to the different levels of complexity of the operations, each of these operator types has different computational resource requirements, e.g., CPU or memory. Some of the operators, e.g., the Parse and Distribute Data operator type, require more resources for processing one data item than others, like the Filter Availability. Besides the computational requirements, each operator type is also assigned with specific Service Level Objectives (SLOs), like the maximal processing duration of one single data item. These SLOs are monitored, and whenever one operator type threatens to violate the imposed SLA, the system needs to provide more computational resources for data processing.

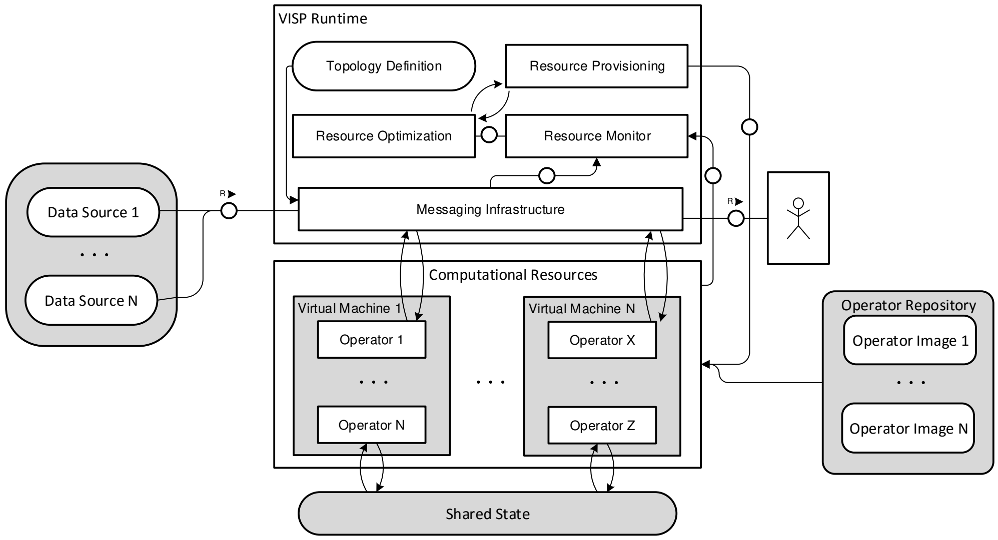


Figure 2. VISP Stream Architecture

2.2 System Architecture

To enact the stream processing topology from the motivational scenario, it is required to instantiate it on an SPE. For our work at hand, we are extending the VISP ecosystem[4], which was introduced in our previous work (Hochreiner et al., 2016b). VISP represents an SPE, which is capable of provisioning computational resources on demand to adapt to the incoming load from data sources. VISP is composed of different components, to cover the whole lifecycle of the stream processing topology enactment. Figure 2 shows a subset of these components, which are relevant for enacting the topology. For a detailed description of the components, please refer to our previous work (Hochreiner et al., 2016b).

The primary task of the SPE, i.e., VISP Runtime, is to process data originating from data sources (on the left side of the figure) to obtain value added data for users (on the right side of the figure). The data sources push the data to the Messaging Infrastructure of VISP, which routes the data based on the Topology Definition. The actual data processing is conducted by Operators, which are deployed on computational resources, e.g., VMs, provided by an Infrastructure as a Service (IaaS) environment. Each operator type is instantiated from dedicated operator images, which are hosted on an external operator repository. To instantiate a specific operator instance on any host for the first time, the operator image needs to be downloaded from the registry, which takes a certain amount of time, depending on the size of the operator image. After the first instantiation of the operator type, the operator image is cached locally on the host to speed up the instantiation of future instances. Each operator type is also assigned with individual SLAs whereas each SLAs consists of different SLOs. The first SLO is the maximum processing duration for one data item and ensures the near real-time processing capabilities of the stream processing topology. The second SLO describes the minimal resource requirements that are needed to instantiate the stream processing operator. These requirements are represented by the minimum amount of memory, i.e., Memory in MegaByte (MB), and the number of CPU shares.

For the enactment of a stream processing topology, each Operator from the topology is represented by at least one, but up to arbitrarily many Operators. These Operators fetch the data from the Messaging Infrastructure, process it and push it back for further processing steps. The remaining components of the VISP Runtime are in charge of monitoring the load on the Messaging Infrastructure as well as on the Operators. This monitoring information is then used by the Resource Optimization component to evaluate whether operator types need to be replicated to deal with the incoming load. The last component, the Resource Provisioning component is in charge of deploying and un-deploying Operators on computational resources.

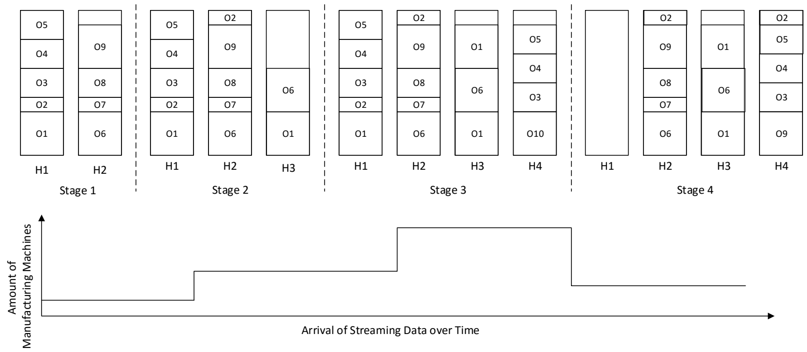


Figure 3. Deployment Stages

2.3 Enactment Scenario

During the enactment, the stream processing operators need to deal with streaming data from a varying amount of manufacturing machines, as shown in Figure 3 at the bottom. This varying data volume requires the SPE to adapt its processing capabilities, i.e., the number of operator instances for specific operator types, which are hosted on an arbitrary amount of hosts, e.g., H1 – H4 in Figure 3, on demand to comply with the SLAs. Nevertheless, the SPE aims at minimizing the needed number of hosts, since each host amounts for additional cost, by using an optimal deployment.

The enactment of our motivational scenario is partitioned into different stages, with a varying number of running manufacturing machines in each stage. At the beginning of Stage 1, each operator is deployed once across the two hosts. Since the volume of streaming data increases after some time, the SPE needs to adapt the processing capabilities by deploying replicas of the operator types O1, O2 and O6 in Stage 2. These operator instances are hosted on a new host H3 because the two already existing hosts cannot cope with the additional operator instances. Because the amount of data increases again in Stage 3, the SPE needs to replicate further operators to comply with the SLAs. Although the second replication of the operator type O1 is feasible on the currently available resources, the SPE is required to lease a new host for the additional operator instances of types O3, O4, O5, and O9.

At the end of Stage 3, the first two hosts meet the end of their BTU. Therefore, the SPE evaluates whether some of the replicated operators can be removed again without violating the SLAs. Because the amount of data is decreasing after Stage 3, the system can remove (O1, O3, O4, and O5) or migrate (O2) some of the operator instances to other hosts. The result is that no operator instances are running on host H1 at the end of its BTU and the SPE can release the host H1 at the end of its BTU, while the host H2 needs to be leased for another BTU.

2.4 Requirements

Based on our motivational scenario, we have identified several requirements, which need to be addressed by the optimization approach.

SLA Compliance The first requirement is SLA compliance, i.e., maximum processing duration, for data that is processed by the stream processing topology. This compliance is the overall goal that needs to be met, regardless of the actual incoming data rate.

Cost Efficiency The second requirement is the cost efficiency for the enactment. This requirement asks for a high system usage of leased resources and an efficient usage of cloud resources, especially regarding their BTU.

Optimization Efficiency The optimization efficiency requirement can be split into two different aspects. The first aspect is the solution of the optimization problem presented in Section 3. Because this optimization problem is NP-complete (see Section 3.2), it is required to devise heuristics to achieve a time and resource efficient optimization approach. The second aspect is that the optimization needs to minimize the number of reconfigurations, e.g., scaling operations, for the stream processing topology because each reconfiguration activity has a negative performance impact on the data processing capabilities.

[4] `https://visp-streaming.github.io`

3 PROBLEM DEFINITION

3.1 System Model and Notation

The system model is used to describe the system state of the individual operator types that form the stream processing topology as well as the used computational resources. The individual operator types are represented by $O = \{1, \ldots, o^{\#}\}$, where $o \in O$ represents a specific operator type. Each operator type o is assigned with minimal resource requirements o_{cpu} and o_{memory}, which need to be met, to instantiate an operator on any host. At runtime, each operator type is represented by at least one, but up to arbitrary many operator instances, which are described by the set $I = \{1, \ldots, i^{\#}\}$, whereas each i_{type} is assigned to a particular operator type $o \in O$.

This set of operator instances I is running on arbitrarily many hosts that are represented by the set $H = \{1, \ldots, h^{\#}\}$, whereas each host hosts a subset of I. Each of these hosts is furthermore assigned with a set of attributes. The attributes h_{cpu} and h_{memory} represent the overall computational resources of the host, and the attributes h_{cpu*} and $h_{memory*}$ represent the remaining computational resources at runtime. The attributes h_{cpu*} and $h_{memory*}$ are decreased for every operator instance i on the specific host h and can be used to determine if it is possible to deploy an additional operator instance on this particular host h. The attribute h_{cost} represents the cost for the host, which needs to be paid for each BTU. The attribute h_{BTU*} represents the remaining, already paid, BTU time. To represent the different startup times between cached and non-cached operator images, each host furthermore denotes a set of images h_{img}. This set contains all operator images $o \in O$, which are cached on this particular host. Each operator type is assigned a specific image, whose identifier is identical to the name of the operator type.

Besides the fundamental operator type attributes for instantiating operators, there is also a set of attributes, which is used to ensure the SLA compliance for data processing. Each operator type is assigned with an estimated data processing duration o_{slo}, that represents the time to process one data item and pass it on to the following operator type according to the stream processing topology. The o_{slo} value is recorded in an optimal processing scenario, where no data item needs to be queued for data processing. Since the SLO o_{slo} only presents the expected processing duration, we also denote the actual processing duration for each operator o_d and the amount of data items o_{queue} that are queued for a particular operator type for processing.

Besides the current o_d, the system model also considers previous processing durations. Here we consider for each operator type o, the last N processing durations o_d denoted as o_{d_1} to o_{d_N}, whereas each of the values gets updated after a new recording of the o_d, i.e., o_{d_1} obtains the value of o_d and o_{d_2} obtains the value of o_{d_1}, etc. If the actual processing duration o_d takes longer than the SLO o_{slo}, penalty cost P accrue to compensate for the violated SLAs each time a violation $v \in V$ occurs.

Furthermore, we denote two operational attributes for each operator type. The attribute $o_{\#}$ represents all current instances, i.e., the sum of all instances of the operator type o and the attribute o_s represents all already executed scaling operations, both upscaling and downscaling, for a specific operator type. Last, we also denote the current incoming amount of data items as DR.

3.2 Optimization Problem

Based on the identified requirements in Section 2.4, we can formulate an optimization problem as shown in Equation 1. The goal of this optimization problem is to minimize the cost for the topology enactment while maintaining given SLOs. This equation is composed of four different terms, which are designed to cover the different requirements. The first term represents the cost for all currently leased hosts by multiplying the number of all currently leased hosts with the cost for a single host. The second and third term are designed to maximize the resource usage on all currently leased hosts regarding the CPU and memory. The last term ensures the SLA compliance of the deployment, due to the penalty cost, which accrue for each SLO violation.

Although the solution of this optimization problem provides an optimal solution for a cost-efficient deployment, it is not feasible to rely on the solution of this problem due to its complexity. To define the complex nature of this problem, we are going to provide a reduction to an unbounded knapsack problem (Andonov et al., 2000), which is known to be NP-hard.

$$\begin{aligned}\text{Min} \quad & h^{\#} \cdot h_{cost} \\ & + \frac{\sum_{h\in H} h_{cpu} - \sum_{i\in I \cap i_{type}=o} o_{cpu}}{\sum_{h\in H} h_{cpu}} \\ & + \frac{\sum_{h\in H} h_{memory} - \sum_{i\in I \cap i_{type}=o} h_{memory}}{\sum_{h\in H} h_{memory}} \\ & + \sum_{v\in V} v \cdot P\end{aligned} \tag{1}$$

Definition of Knapsack Problem The unbounded knapsack problem assumes a knapsack, whose weight capacity is bounded by a maximum capacity of C and a set of artifacts A. Each of these artifacts a is assigned with a specific weight $a_w > 0$ as well as a specific value $a_v > 0$ and can be placed an arbitrary amount of times in the knapsack. The goal is to find a set $A1$ of items, where $\sum_{a\in A} a_w \leq C$ and $\sum_{a\in A} a_v$ is maximized.

NP-Hardness of the Optimization Problem For our reduction, we assume a specific instance of our optimization problem. For this specific instance, we assume that the number of hosts is fixed and that each of the operators has the same memory requirements o_{memory}. Furthermore, we define the value of a specific operator by the amount of data items o_{queue} that are queued for a specific operator type, i.e., the more items need to be processed, the higher is the value for instantiating a specific operator.

Based on this specific instance of the optimization problem, we can build an instance of the unbounded knapsack problem, where the maximum capacity C is defined by the maximum amount of CPU resources on all available hosts $\sum_{h\in H} h_{cpu}$, the weight a_w of the artifacts a is defined by the CPU requirements o_{cpu} of one operator and the value a_v of the artifact is defined by the number of items waiting on the operator type-specific queue o_{queue}.

Because a specific instance of our optimization problem can be formulated as a knapsack problem, we can conclude that our optimization problem is also NP-hard. This concludes that there is no known solution which can obtain an optimal solution in polynomial time. Since this conclusion conflicts with the third requirement given in Section 2.4, we decided to realize a heuristic-based optimization approach, which can be solved in polynomial time.

4 OPTIMIZATION APPROACH

The overall goal our optimization approach is to minimize the cost for computational resources and maximize the usage of already leased VMs while maintaining a high quality of service. Therefore, we apply an on-demand approach to reduce the deployment and configuration overhead, i.e., instantiating and removing additional operator instances, and minimize the computation resources required for finding an optimal deployment configuration. Due to our emphasis on the BTUs of VMs, we call our approach BTU-based approach in the remainder of this paper.

4.1 Ensure Sufficient Processing Capabilities

To avoid penalty cost, our approach continuously evaluates the SLA compliance of the stream processing topology. Whenever the individual processing duration o_d of a particular operator type o exceeds or threatens to exceed the maximum allowed processing duration o_{slo} according to the *Upscaling Algorithm* as shown in Algorithm 1, the upscaling procedure for the specific operator type is triggered.

This upscaling procedure consists of several steps, as depicted in Figure 4. The first task is to evaluate if any of the currently running hosts offers enough computational resources to host the additional instance of the specific operator. Therefore, we apply the *Host Selection Algorithm*, as described in Algorithm 2, for every currently running host to obtain a utility value for the host. Assuming that there is at least one host with a positive utility value, the host with the best utility value is selected to deploy the new operator instance, and the upscaling procedure is finished.

When no host with a positive utility value is available, i.e., no hosts offers enough computational resources to instantiate a new instance for the required operator type, there are two possibilities to obtain the required computational resources. The first possibility is to scale down existing operators when they are not required anymore. We therefore apply the *Operator Selection Algorithm*, as described in

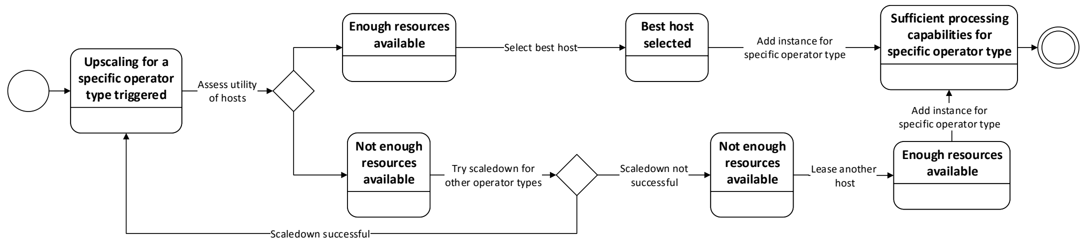


Figure 4. Upscaling procedure for a specific operator type

Algorithm 3. This algorithm assigns each operator with a utility value, which describes the suitability to scale down one instance of a particular operator type. Whenever there is an operator type with a positive utility value, one operator instance for the operator type with the highest utility value will be scaled down. After the scale down operation is finished, the procedure starts the upscaling procedure for the particular operator type again.

If there is no operator with a positive utility value, i.e., all operators are needed for SLA-compliant data stream processing, the SPE needs to start a new host, deploys the operator instance on the new host and finishes the upscaling procedure. To eliminate any unnecessary system reconfigurations, we apply a preliminary simulation phase for the downscaling operations. This simulation step ensures that enough operator instances can be scaled down to host the new operator instance. If the simulation fails, a new host is spawned immediately without any further system reconfigurations.

4.2 Optimize Resource Usage

To minimize the cost of computational resources, the optimization approach aims at using the leased resources as efficient as possible. This means that the SPE uses all paid resources until the end of their BTUs and evaluate shortly before, i.e., within the last 5% of the BTU, whether a host needs to be leased for another BTU, i.e., the resources are still required, or if the host can be released again.

To achieve this releasing procedure, as shown in Figure 5, all operator instances running on the designated host, which is targeted to be shut down, need to be either migrated to other hosts or can be released as well. Therefore, the optimization approach applies the *Operator Selection Algorithm* for all operator types, which have running instances on this host, and obtains their utility value. If any of the operator types has a positive utility value, the operator instances of this type running on this host are released.

When any operator instances are remaining whose operator types cannot be scaled down, the optimization approach tries to migrate the operators to other, currently running hosts. Here we apply the upscaling procedure for operator types, as described in Section 4.1. The only difference is that the host, which is targeted to be released, is omitted as a suitable host. If all operator instances can be successfully migrated, i.e., instantiated on other hosts, the operator instances are removed from this host and the host can be released again. When the migrations are not feasible, the host is leased for another BTU.

4.3 Algorithms

To realize our resource provisioning approach, we have devised three algorithms, which are discussed in detail in this section.

The first algorithm, the *Upscaling Algorithm* as listed in Algorithm 1, is used to evaluate whether any operator needs to be scaled up. This algorithm is executed on a regular basis, e.g., every 10 seconds for each operator type o and either returns 0, if the current stream processing capabilities are enough to comply with the SLAs, or 1 if the operator type needs to be scaled up. Therefore, this algorithm considers, on the one hand, the current processing duration of the operator (Line 2) and, on the other hand, the trend of the previous processing durations. For the trend prediction, we apply a simple linear regression for the last N observations, based on the linear least squares estimator (Lines 5 – 9). If the current duration o_d or the predicted duration is higher than the SLO o_{slo}, we consider the operator type to be scaled up (Line 10). Before we trigger the upscaling operation, we apply an additional check if the upscaling operation is required.

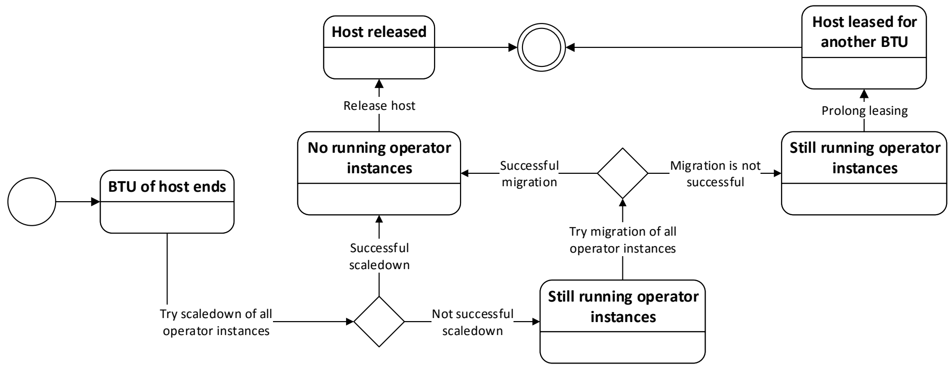


Figure 5. Downscaling procedure for a host

The stream processing topology may retrieve short-term data peaks whose volume can be so high that it can be only processed with a short delay. This results in a short time in high processing durations that disappear without any further activity already after a short time. Nevertheless, the upscaling algorithm would trigger the upscaling procedure, although the additional resources may not be required anymore. Therefore, the algorithm also considers the current load of data items o_{queue} before scaling up by checking whether the amount of queued items for processing exceeds a configurable $scalingThreshold$ (Lines 13 – 16).

Algorithm 1 Upscaling Algorithm

1: **function** UPTRIGGER(o, N)
2:     **if** $o_d > o_{slo}$ **then**
3:         upscaling = 1
4:     **end if**
5:     observationsMean = $\frac{1}{N} * \sum_{i=1}^{N} i$
6:     durationMean = $\frac{1}{N} * \sum_{i=1}^{N} o_{d_i}$
7:     $\beta = \frac{\sum_{i=1}^{N}(i - \text{observationsMean}) * (o_{d_i} * \text{durationMean})}{\sum_{i=1}^{N}(i - \text{observationsMean})^2}$
8:     $\alpha = \text{durationMean} - \beta * \text{observationsMean}$
9:     predictedDuration = $\alpha + \beta * (N + 1)$
10:     **if** predictedDuration $> o_{slo}$ **then**
11:         upscaling = 1
12:     **end if**
13:     **if** upscaling = 0 **then**
14:         **return** 0
15:     **end if**
16:     **if** $o_{queue} >$ scalingThreshold **then**
17:         **return** 1
18:     **end if**
19:     **return** 0
20: **end function**

The second algorithm, the *Host Selection Algorithm* (see Algorithm 2), is used to rank all currently leased hosts according to their suitability to host a new operator instance of a particular operator type. Therefore, the algorithm evaluates for each host h whether a new instance of the required operator type o could be hosted on that specific host at all. Here, the algorithm considers both, the CPU and memory requirements, and derives the maximum amount of instances that can be hosted. If this value is less than 1, i.e., there are no resources left for a single additional operator instance, the function returns a negative value. The first check evaluates the feasibility of deploying a new operator instance on the host (Lines 2 – 5). In a second stage, this algorithm evaluates the suitability of this host. Here the algorithm simulates the resource usage of the host, assuming the operator instance would be deployed on the host. The overall goal is an equal distribution of CPU and memory usage across all hosts, to avoid situations where hosts maximize their CPU usage, but hardly use any memory and vice versa. Therefore, the algorithm calculates the difference between the normalized CPU usage and memory usage, whereas a lower value represents a better ratio between CPU and memory and therefore a better fit (Lines 6 – 9). Besides the equal distribution of memory and CPU on the individual hosts, we also want to distribute the operators equally among all currently leased hosts. The assigned CPU o_{cpu} and memory o_{memory} attributes only represent the resources which are guaranteed for the operators. This allows operators to use currently unused resources of the hosts based on a first come first service principle. To maximize the usage, we aim for an equal distribution of the unassigned resources, i.e., h_{cpu*} and $h_{memory*}$, which can be used by the operators to cover short-term data peaks without any reconfigurations required. This aspect is covered by dividing the $difference$ value by the $feasibility$ value to prefer those hosts which are least used (Line 9). Last, we also consider the deployment time aspect for a particular operator type. Here, we prefer those hosts, which have already the operator image cached. Therefore, we multiply the $suitability$ value with a constant factor CF to prefer those hosts which have a cached copy (Lines 10 – 12).

This allows us to prioritize those hosts that provide a fast startup while maintaining the resource-based ranking. The result of this algorithm is either a negative value for a host, i.e., the host can run the new operator instance, or a positive value, whereas the lowest value among several hosts shows the best suitability.

Algorithm 2 Host Selection Algorithm

1: **function** UP(h, o)
2: feasibilityThreshold = $\min((h_{cpu*}/o_{cpu}),(h_{memory*}/o_{memory}))$
3: **if** feasibilityThreshold < 1 **then**
4: **return** -1
5: **end if**
6: remainingCPU = $h_{cpu*} - o_{cpu}$
7: remainingMemory = $h_{memory*} - o_{memory}$
8: difference = $|\frac{\text{remainingCPU}}{h_{cpu}} - \frac{\text{remainingMemory}}{h_{memory}}|$
9: suitability = $\frac{\text{difference}}{\text{feasibilityThreshold}}$
10: **if** s $\in h_{img}$ **then**
11: suitability = suitability * CF
12: **end if**
13: **return** suitability
14: **end function**

The third algorithm, the *Operator Selection Algorithm* (see Algorithm 3), is used to select operator types which can be scaled down without violating the SLOs. Therefore, this algorithm considers several static as well as runtime aspects of the operator types. The goal of the algorithm is to obtain a value which describes the suitability of a particular operator type to be scaled down. Whenever the value is negative, the operator type must not be scaled down, i.e., all operator instances for this type are required to fulfill the SLO.

First, the algorithm ensures that there is at least one operator instance for the given operator type (Lines 2 – 4). Second, the function considers the amount of all currently running instances for the specific operator type and normalizes it to obtain a value between 0 and 1 (Line 5). This normalization is carried out based on the maximal respectively minimal amount of instances for all operator types. This value represents the aspect that it is better to scale down an operator type with numerous operator instances because the scale down operation removes a smaller percentage of processing power compared to an operator type with fewer operator instances.

Furthermore, we consider the SLA compliance of the particular operator. Here, we consider the actual compliance for the processing duration and multiply with the penalty cost as a weighting factor (Line 7). Whenever the processing duration o_d takes longer than the SLO o_{slo}, the delay value will be less than one, but when there is any delay, the delay value can become arbitrarily high. The next value for consideration is the relative amount of scaling operations (both up and down) in contrast to the entire scaling operations (Lines 7). Here, we penalize previous scaling operations because we want to avoid any oscillating effects,

Table 1. Sensor Types

	Emission Rate / min	Size (Bytes)
Availability Sensor (S1)	5	95
Production Data (S2)	1	12500
Temperature Sensor (S3)	10	90

i.e., multiple up- and downscaling operations for a specific operator. The last factor is the queueLoad. In the course of our evaluations, we have seen that the algorithm may take a long time to recover after a load peak, i.e., release obsolete operator instances as soon as the data is processed. This can be observed when the SPE is confronted with a massive data spike followed by a small data volume for some time. For this scenario, the heuristic discourages any downscaling operation due to the delay factor, which may be high due to the delayed processing of the data spike. To resolve this shortcoming, we introduce the queueLoad factor QL, which encourages the downscaling of an operator type, as soon as no data items are waiting in the incoming queue o_{queue} (Lines 8 – 12).

Finally, we join the distinct aspects to obtain the overall utility value. While the number of instances represents a positive aspect to scale down an operator, all other aspects discourage a scaling operation. Therefore, we apply different weights W_1, W_2, W_3, and W_4 on the individual values and deduce all other aspects from the instance value. The result is the utility value, which describes the suitability of the particular operator to be scaled down, whereas a higher value suggests a better suitability (Line 13).

Algorithm 3 Operator Selection Algorithm

1: **function** DOWN(o)
2:     **if** $o_{\#} < 2$ **then**
3:         **return** -1
4:     **end if**
5:     instances = $\frac{o_{\#} - \min(o_{\#} \in O)}{\max(o_{\#} \in O) - \min(o_{\#} \in O)}$
6:     delay = $\frac{o_d}{o_{slo}} * P$
7:     scalings = $\frac{o_s}{\sum_{o_s \in O} o_s}$
8:     **if** $o_{queue} < 1$ **then**
9:         queueLoad $=$ QL
10:     **else**
11:         queueLoad $= 0$
12:     **end if**
13:     **return** (instances $* W_1$) $-$ delay $* W_2$ $-$ scalings $* W_3$ $+$ queueLoad $* W_4$
14: **end function**

5 EVALUATION

5.1 Evaluation Setup

For our evaluation, we revisit out motivational scenario (see Section 2) and discuss the concrete implementation of this topology.

5.1.1 Sensor Types

First, we are going to discuss the sensors which emit the data items for our topology. In this topology, we consider three different sensor types, as listed in Table 1. Each of these sensor types generates a data item, with a particular structure, which can be only processed by a dedicated operator type, e.g., O1 for sensor type S2. Due to the different structure, the size of the data items also differs. The first and the last sensor type (S1 and S3) encode the information in plain text that results in rather small data items with a size of 9 to 95 Bytes. The second sensor type encodes the information with an image and is therefore much larger, i.e., around 12500 Bytes.

Table 2. Stream Processing Operator Types

	Processing Duration (ms)	CPU Shares	Memory (MB)	Storage (MB)	State	Outgoing Ratio
Parse and Distribute Data (O1)	900	660	452	89		1:3
Filter Availability (O2)	600	131	524	68	✓	50:1
Calculate Performance (O3)	750	100	430	68		1:1
Calculate Availability (O4)	750	83	502	68	✓	1:1
Calculate Quality (O5)	750	77	527	68	✓	1:1
Monitor Temperature (O6)	600	65	440	68	✓	100:1
Calculate OEE (O7)	700	46	464	68	✓	3:1
Inform User (O8)	500	74	466	68		1:0
Generate Report (O9)	1300	47	452	70	✓	300:1

5.1.2 Operator Types

The second important implementation aspect for the topology are the operators. Each of these operator types performs a specific task with specific resource requirements and specific processing durations. Table 2 lists all operator types which are used in this evaluation. Each operator is assigned a number of different performance as well as resource metrics. The resource metrics represent mean values across several topology enactments. The processing duration represents the average times which are required to process one specific data item as well as the time the data item is processed within the messaging infrastructure between the previous operator and the one in focus. The CPU metric represents the amounts of shares, which are required by the operator when executed on a single core VM. The memory value represents the mean memory usage. This memory value accumulates the actual used memory by the operator instances and the currently used file cache, which results in a rather high value compared to the actual size of the operator image. The CPU metric and the memory metric are determined based on long term recordings, whereas the stated value in the table is calculated by adding both the absolute maximum and the average value of all observations for a specific operator and dividing this value by 2. For the processing duration, we have conducted several preliminary evaluations, where the SPE is processing constant data volumes in a fixed over-provisioning scenario to avoid any waiting durations for the recordings.

For the storage operator, we have three different sizes. Because the majority of the processing operators only implement processing logic, the size of the images is the same for them. The only two exceptions are the Generate Report (O9) image, which also contains a PDF generation library and the Parse and Distribute Data (O1) operator, which also contains the Tesseract binary, which is required to parse the images. Each of the stateful operators, as indicated in the table, can store and retrieve data from the shared state to synchronize the data among different data items and different instances of one operator type. The outgoing ratio describes whether a particular operator type consumes more data items than it emits, e.g., O7 combines three data items before it emits a combined one, or whether it emits more data items than it receives, e.g., O1 distributes the production information to three other operator types.

For our scenario, we have implemented nine different operators[5] as Spring Boot[6] applications, which are discussed in detail in the remainder of this section.

Parse and Distribute Data (O1) The Parse and Distribute Data operator type is designed to receive an image with encoded production data and parse this image to extract the information. For our implementation, we use the Tesseract OCR Engine[7] to parse the image and then the Spring Boot application forwards the machine readable production data to the downstream operator types.

[5] `https://github.com/visp-streaming/processingNodes`
[6] `https://projects.spring.io/spring-boot/`
[7] `https://github.com/tesseract-ocr/tesseract`

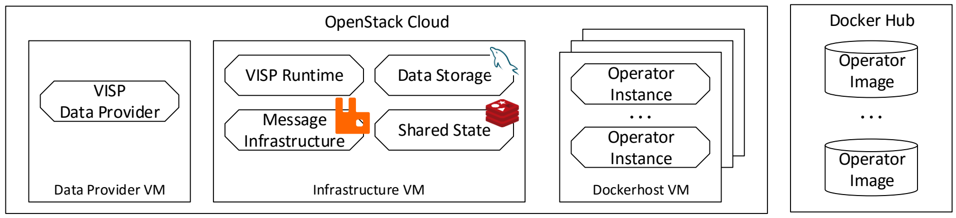


Figure 6. Deployment for the Evaluation Scenario

Filter Availability (O2) Each manufacturing machine can have three different availability types: available, planned downtime, and defect. While the first two types represent intended behavior, the last type signals a defect and should be propagated to a human-operator. This operator issues a new warning for each new defect notification and filters all other data items.

Calculate Performance (O3) The Calculate Performance operator type calculates the performance of the last reporting cycle, i.e., the time between two production data emissions. The actual performance is derived by the formula shown in Equation 2 (Nakajima, 1988).

$$performance = \frac{producedItems \cdot idealProductionTime}{reportingCycle} \tag{2}$$

Calculate Availability (O4) The Calculate Availability operator type represents the overall availability of the manufacturing machine from the beginning of the production cycle, e.g., the start of the evaluation. The availability is defined by the formula shown in Equation 3 (Nakajima, 1988).

$$availability = \frac{totalTime - scheduledDowntime - unscheduledDowntime}{totalTime} \tag{3}$$

Calculate Quality (O5) The Calculate Quality operator type represents the ratio between all produced goods against defect goods from the beginning of the production cycle. The quality is defined by the formula shown in Equation 4 (Nakajima, 1988).

$$quality = \frac{totalProducedGoods - totalDefectiveGoods}{totalProducedGoods} \tag{4}$$

Monitor Temperature (O6) The Monitor Temperature operator type filters all temperatures below a predefined threshold and issues a notification to the human operator for each new temperature violation.

Calculate OEE (O7) The Calculate OEE operator synchronizes the upstream operations based on the timestamp of the initial data item and calculates the overall OEE value according to the formula in Equation 5.

$$oee = availability \cdot performance \cdot quality \tag{5}$$

Inform User (O8) The Inform User operator type forwards the notifications to a human user. In our evaluation scenario, this operator type only serves as a monitoring endpoint for the SLA compliance and all incoming data items are discarded at this operator type.

Generate Report (O9) The Generate Report operator aggregates multiple OEE values and generates a PDF report which aggregates a predefined amount of OEE values. This report is then forwarded to the user for further manual inspection.

5.2 Evaluation Deployment

For our evaluation, we make use of the VISP Testbed (Hochreiner, 2017), which is a toolkit of different evaluation utilities that support repeatable evaluation runs. The most notable component of this toolkit is the VISP Data Provider, which allows simulating an arbitrary amount of data sources by emitting them according to a predefined message structure. Furthermore, the Data Provider also allows defining different arrival patterns (see Section 5.4) to evaluate the adaptation possibilities of the VISP Runtime, in particular of its scaling mechanism.

The evaluation runs are carried out in a private cloud running OpenStack[8], whereas the components are deployed on different VMs, as depicted in Figure 6. The most relevant VM for our evaluation is the Infrastructure VM, which hosts the VISP Runtime as well as all other relevant services, like the Message Infrastructure, i.e., RabbitMQ[9], the Shared State, i.e., Redis[10] and the Data Storage, i.e., a MySQL[11] database.

For the topology enactment, the VISP Runtime leases (and releases) an arbitrary amount of VMs, i.e., Dockerhost VMs, on the private OpenStack-based cloud at runtime. These Dockerhost VMs are used to run the Operator Instances, which take care of the actual data processing as described in Section 2.2. The BTU for these VMs is set to 600 seconds, which represents a rather short BTU for public cloud providers. Nevertheless, we have chosen this interval on purpose to evaluate the applicability of our BTU-based approach as often as possible during our evaluation runs. Furthermore, we use a homogeneous size for all Dockerhost VMs with 3 virtual CPU cores and 5 GB Ram. The Operator Images, which are required to run the Operator Instances, are hosted on an external service, i.e., Dockerhub[12]. Finally, the Data Provider VM is in charge of simulating the data stream from the sensors, as described in Section 5.1.1.

5.3 Baseline

To evaluate our BTU-based optimization approach, we have selected a threshold-based baseline provisioning approach. The baseline implements a commonly used provisioning approach which was also used in our previous work (Hochreiner et al., 2016a). The approach considers the amount of data items waiting on the incoming queue for processing as scaling trigger. As soon as the variable o_{queue} exceeds an upper threshold, i.e., 250, the SPE triggers an upscaling operation for this operator and as soon as o_{queue} falls below a lower threshold, i.e., 1, the SPE triggers one downscaling action of an operator. Besides the single upscaling trigger, our threshold-based approach triggers the upscaling operation twice, if o_{queue} surpasses a second upper threshold of 1000 data items waiting for processing. Regarding the leasing of VMs, we apply an on-demand approach, where the SPE leases a new VM as soon as all currently used VMs are fully utilized and releases a VM, as soon as the last operator instance on that VM is terminated. Analogous to the BTU-based provisioning, the threshold-based provisioning mechanism was also executed every 20 seconds.

5.4 Data Arrival Pattern

For our evaluation, we have selected two different arrival patterns which simulate different load scenarios for the SPE by submitting different data volumes to the SPE. The first arrival pattern has three different data volume levels, which are changed stepwise, so that the resulting arrival pattern could be approximated to a sinus curve, as shown in Figure 7. These three different volume levels simulate different amounts of manufacturing machines ranging from two to six machines that emit different amounts of data items, as shown in Table 1. To speed up the evaluation, we simulate the data emissions, which would arise every minute every 480 milliseconds. This enables us on the one hand to simulate 500 real time minutes within only four minutes in the course of our evaluation and therefore also increases the load on the SPE. This also results in a volume level change every four minutes.

The second arrival pattern has only two levels, i.e., the lowest and the highest of the first pattern, which confronts the SPE with more drastic volume changes. Due to the fact that we only apply two different levels, the state changes are twice as long as for the first pattern, i.e., eight minutes.

[8] `https://www.openstack.org`
[9] `https://www.rabbitmq.com`
[10] `http://redis.io`
[11] `https://www.mysql.com`
[12] `https://hub.docker.com`

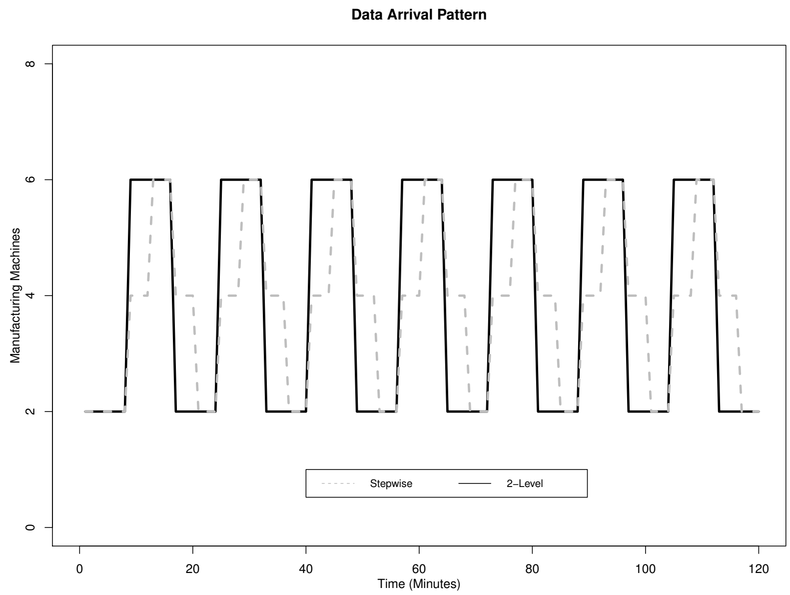


Figure 7. Data Arrival Pattern for the Evaluation

Both patterns are continuously generated by the VISP Data-Provider[13] for the whole evaluation duration of 120 minutes.

5.5 Metrics

To compare the evaluation results for both the BTU-based and the threshold-based resource provisioning approaches, we have selected several metrics to describe both the overall cost as well as QoS metrics. After each evaluation run, these metrics are extracted by the VISP Reporting Utility[14]. The most important metric is *Paid BTUs*, which describes the total cost for data processing. This value comprises all *VM Upscaling* and *VM Prolonging* operations, which either lease new VMs or extend the leasing for another BTU for existing ones. The *VM Downscaling* sums up all downscaling operations, which are conducted before the end of the BTU.

The next set of metrics describes the QoS of the stream processing application. Each stream processing operator is assigned a specific processing duration which describes the processing duration in a constant over-provisioning scenario. Due to the changing data volume in our evaluation scenarios, it is often the case that the system suffers from under-provisioning for a short time, which results in longer processing durations. To assess the overall compliance of the processing durations, we define three different SLA-compliance level. The first compliance level requires *real-time* processing capabilities, and states the share of data items that are produced within the given processing duration. The second level applies *near-realtime* requirements, which is defined by processing durations that take at most twice as long as the defined processing duration, and the third level applies a *relaxed* strategy, which means that the data items need to be processed within at most five times the stated processing duration. These SLA metrics are obtained from the processing duration of the data items, which are recorded by the operators. To reduce the overall monitoring overhead, we only measure the processing duration of every tenth data item. Nevertheless, preliminary evaluations with other intervals, e.g., every data item or every third data item have shown a similar metric reliability. This similar reliability can be explained due to the fact that observing every tenth data item still yields about 20-40 performance readings/second (depending on the data volume). Therefore it is save to assume that these metrics cover all effects of the SPE because all other activities, e.g., spawning a new operator instance takes 5-10 seconds or leasing a new VM takes about 30-60 seconds.

[13]`https://github.com/visp-streaming/dataProvider`
[14]`https://github.com/visp-streaming/reporting`

Table 3. Evaluation Results

	Stepwise Pattern		2-Level Pattern	
	BTU-based	Threshold-based	BTU-based	Threshold-based
Paid BTUs	72	82	76	86
VM Upscaling	27	29	26	25
VM Prolonging	45	53	50	61
VM Downscaling	21	0	18	0
VM Early-Downscaling	0	24	0	18
Real-time Compliance	40.58%	40.28%	33.71%	40.53%
Near-real-time Compliance	72.79%	68.86%	60.08%	68.56%
Relaxed-time Compliance	76.24%	72.72%	63.07%	71.21%
Real-time Compliance without O2 and O6	50.19%	58.28%	50.05%	57.54%
Near-real-time Compliance without O2 and O6	86.10%	87.83%	86.61%	85.45%
Relaxed-time Compliance without O2 and O6	89.29%	89.27%	90.38%	86.94%
Mean Time To Adapt (s)	16.90 (σ = 23.46)	2.09 (σ = 5.69)	17.71 (σ = 28.28)	2.05 (σ = 5.53)
Operator Instance Up	151	178	176	177
Operator Instance Down	126	159	148	159
Operator Instance Migration	93	0	81	0

The *Time To Adapt* metric states the arithmetic mean duration, which is required until the delayed processing for an operator type is back to real-time processing.

The last metrics describe the scaling operations of operator instances. Here we consider *Upscaling*, *Downscaling* as well as *Migration* operations among different hosts.

6 RESULTS AND DISCUSSION

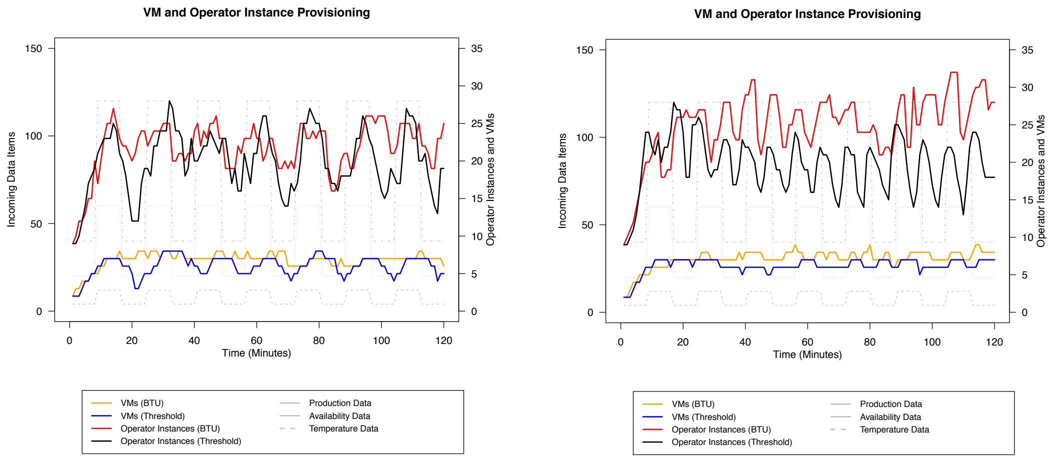
To obtain reliable numbers, we have conducted three evaluation runs for each provisioning approach and data arrival pattern, which results in 12 total evaluation runs. These evaluations have been executed as three batches over the time span of one week to avoid any corruption of the results due to different loads on the private OpenStack-based testbed.

The raw data for all evaluation runs is provided as supplemental material, nevertheless for our discussion we have selected the evaluation run for each evaluation scenario, which we are analyzing in detail in the remainder of this section. The overall results of these evaluations are listed in Table 3, whereas the description for the individual metrics can be found in Section 5.5. To visualize the results, we provide one comparison figure (see Figures 8a and 8b) for each data arrival pattern. These figures depict the total amount of leased VMs as well as operator instances over time. For reference, we also provide the amount of data items, which are emitted by the sensors. Furthermore, we provide for each evaluation scenario a dedicated figure (see Figures 9 and 10), which show the scaling activities mapped to the total number of leased VMs as well as operator instances.

The discussion of the evaluation consists of two parts, which are separated into two data arrival patterns.

6.1 Stepwise Data-arrival Pattern

For the stepwise pattern we can see that the overall enactment cost, i.e., paid BTUs, is 12% lower for the BTU-based approach (72 paid BTUs) than for the threshold-based, which requires 82 paid BTUs. The



(a) Running Operator Instances and VMs for Stepwise Pattern

(b) Running Operator Instances and VMs for 2-Level Pattern

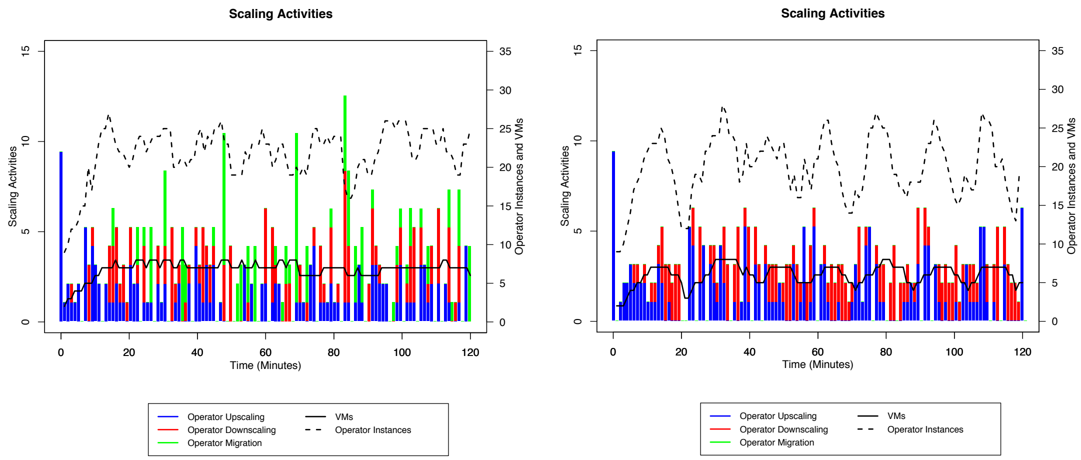
Figure 8. Resource Provisioning Recordings

reason for this cost reduction is that the threshold-based approach does not consider the end of a BTU, which results in potentially unnecessary VM prolonging. Furthermore, this also results in premature VM downscaling operations before the actual end of the BTU, which results in a waste of already paid computational resources. Nevertheless, it must be noted, that the threshold-based approach is still more cost-efficient than a fixed over-provisioning one, which would have resulted in 96 paid BTUs. For the over-provisioning scenario we assume the constant leasing of eight VMs, which is the maximum of leased VMs for both the threshold-based and the BTU-based scenario according to Figure 8a.

Regarding SLA compliance, i.e., compliance to the predefined processing durations, both approaches perform at similar levels, although it must be noted that the BTU-based approach performs better despite less cost. For the SLA compliance, we can also observe that the operators O2 and O6 have a higher impact on the SLA compliance compared to the other operators. This is because these two operators receive the majority of the incoming traffic and in the case of a volume change, it is harder for them to scale up immediately. When we compare the SLA compliances of all operators against the SLA compliances without the operators O2 and O6, we can see that the SLA compliance for the realtime restriction is 9.61% higher for the BTU-based approach and 18% higher for the threshold-based one. This difference can be explained due to the lazy release approach of the BTU-based approach. While the threshold-based approach releases obsolete operator instances as soon as possible, the BTU-based approach only releases them when it is required, i.e., one VM reaches the end of a BTU, or other upscaling operations need computational resources. Due to the eager downscaling activities of the threshold-based approach, it is often required to compensate the lacking resources for the operators O2 and O6 within a short time, while the BTU-based approach may maintain this overcapacity and does not need to adapt to the volume change.

Regarding the mean time to adapt, the threshold-based approach outperforms the BTU-based one, because the BTU-based one is more conservative in leasing new VMs. Instead of immediately leasing a new VM when there are no computational resources available, the BTU-based approach first evaluates whether it can scale down other operator instances for the new operator instance. This evaluation operation and the further downscaling operation take around 20 seconds because we apply a graceful downscaling approach for the operator instances. This graceful approach deregisters the operator for new data items and waits for 20 seconds before being released. We choose these 20 seconds waiting time to ensure that all data items are safely processed, and none of them are lost. Nevertheless, these additional waiting time to release computational resources results in the higher time to adapt compared to the threshold-based approach.

When we analyze the scaling activities in Figure 9a, we can see that there are several migration operations which are triggered whenever the BTU of a VM is at its end. Although such a migration operation triggers a downscaling and upscaling operation, which causes additional overhead for the SPE,



(a) BTU-based Approach

(b) Threshold-based Approach

Figure 9. Scaling Activities for the Stepwise Pattern

it is still more cost-efficient than the threshold-based approach, which does not consider these migrations. Furthermore, it must be noted that the BTU for our evaluation is rather short, which triggers a high amount of migration operations. When the SPE would operate with longer BTUs, these migration operations would be less and thus improve the overall performance of the SPE, including the SLA compliance. For the threshold-based approach (Figure 9b), it is more straightforward because the number of operator instances are aligned with the incoming load.

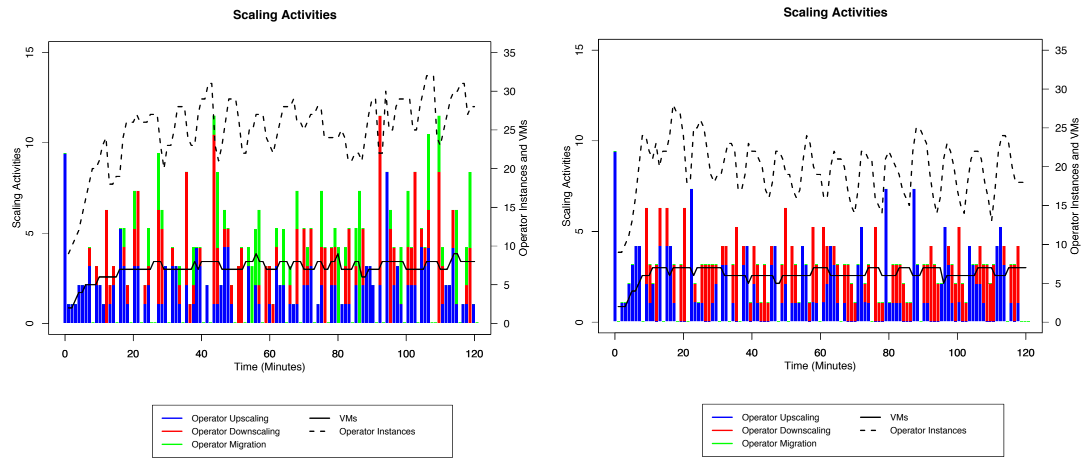
In Figure 8a, it can can be further observed that the number of leased VMs is several times higher for the BTU-based approach than for the threshold-based one, although the total cost are lower. This can be explained due to the fact that the threshold-based approach often releases VMs prematurely, which may render a lower number of VMs in the picture, but still, leads to higher cost.

6.2 2-Level Data-arrival Pattern

For the second part, we discuss the 2-level pattern. This data arrival pattern poses significant volume changes to the SPE, compared to the stepwise pattern, which requires more scaling effort to comply with the incoming load. Regarding the overall enactment cost, we can also observe for this scenario, that the BTU-based renders less cost (12%) compared to the threshold-based approach and about 29% less cost compared to a fixed-provisioning scenario, where we assume that we constantly lease nine VMs (according to Figure 8a) which results in 108 paid BTUs.

Nevertheless, it can be observed that this cost reduction also has a negative impact on the SLA compliance. In Table 3, it can be seen that the BTU-based approach performs worse than the threshold-based one. This is mainly due to the higher agility of the threshold-based approach, which can lease new computational resources on demand, and does not need to wait until existing resources are freed by getting rid of obsolete operator instances. This effect can be especially observed when comparing the overall compliance metrics to the compliance metrics without the operators O2 and O6. Here we can see that the realtime compliance without the operators O2 and O6 (50.05%) is more than 16% higher than the total compliance of 33.71%. This difference shows that the operators O2 and O6 are confronted with a large volume increase every 16 minutes, and the SPE needs to adapt. This adaptation takes some time, which in turn results in a lower SLA compliance. Nevertheless, when we observe the near-time compliance and the relaxed-time compliance, we can see that the BTU-based approach provides better compliance results because the majority of the other components is not changed in the low volume phases and therefore require less adaptation than for the threshold-based approach. For the time to adapt approach, we can observe a similar behavior compared to the stepwise pattern, where the BTU-based approach has a higher mean time to adopt than the threshold-based approach.

Regarding operator instance scaling activities it can be observed in Figure 8b that the number of operators for the threshold-based approach roughly follows the data volume pattern. Nevertheless, we can sometimes observe two operator instance spikes for one volume spike. This can be explained by the fact



(a) BTU-based Approach **(b)** Threshold-based Approach

Figure 10. Scaling Activities for the 2-Level Pattern

that the threshold-based approach operates based on the amount of data items waiting on the incoming queue. At the beginning of the volume spike, the queued data items increase, until the SPE has enough computational resources available. As soon as this is the case, the queued data items drop to zero, and the threshold-based approach starts to get rid of operator instances due to its downscaling threshold. After a short time, i.e., the next optimization cycle, the threshold-based approach again realizes that there are too little computational resources available and triggers the upscaling procedure again. While the BTU-based approach is immune to this preliminary downscaling, it must be noted that the BTU-based approach also needs to deal with up- and downscaling incentives. Due to the rather short BTU of 10 minutes for the evaluation, it is often the case that the downscaling procedure triggers downscaling activities, which results in a similar situation as for the threshold-based approach, e.g., around minute 45 in Figure 8b. This can be also observed in Figure 10a, where each VM downscaling approach triggers a number of migrations. Nevertheless it must be said, that the number of active up- and downscaling activities for the BTU-based approach (see Figure 10a) is lower than the ones for the threshold-based approach (see Figure 10b).

In conclusion, it can be said that the BTU-based approach renders less cost for the enactment of stream processing topologies due to the better usage of computational resources while maintaining a similar or even higher level of SLA compliance. Furthermore, it must be noted that the BTU-based approach is rather suited for smaller volume changes, like the stepwise pattern in contrast to drastic changes, like the 2-level pattern. Nevertheless, based on the evaluation, we can see that the BTU-based approach performs better than a threshold-based approach and way more cost-efficient than a fixed over-provisioning approach.

6.3 Threats to Applicability

Although the presented system model builds on top of real world observations, there are nevertheless several aspects which may have an adverse impact to its applicability in real world environments. The first aspect is a threat to the validity for our evaluation results. Due to the cloud-based evaluation environment, we cannot rule out any influences of other VMs that are running on the same hardware. We tried to mitigate this effect repeating our evaluation three times on different workdays (including the weekend), but still, we cannot rule out these effects to our evaluation. The second aspect considers the applicability of the BTU-based approach to different data arrival pattern. While the evaluation shows that BTU-based approaches are a promising fit for data volumes which are not exposed to high changes, like the stepwise pattern, they sometimes struggle to keep up with the SLA compliance for rapidly changing data volumes. Finally, it also needs to be mentioned that the BTU-based approach has only limited applicability for private clouds since they do not require any BTU-based payments.

7 RELATED WORK

In the last couple of years, the landscape of SPEs has been constantly increasing. In contrast to the rather basic SPEs, like Aurora (Balakrishnan et al., 2004) or Borealis (Abadi et al., 2005), which have been designed more than a decade ago, today's SPEs incorporate technological advances like cloud computing and can process large volumes of data in parallel. While some of these SPEs are rather focused on cluster-based deployments, like System S (Gedik et al., 2008), most are designed to utilize cloud-based deployments, like Apache Spark (Zaharia et al., 2010), Apache Flink (Carbone et al., 2015), Apache Storm (Toshniwal et al., 2014) or its derivative Heron (Kulkarni et al., 2015). Despite the focus on designing efficient SPEs, there are to the best of our knowledge no established SPEs which support elastic stream processing, especially the cost-efficient enactment of stream processing topologies. Nevertheless, there are a couple of prototypes and concepts in the literature, which propose a mechanism for elastic stream processing.

Several research groups have picked up the challenge of replacing the previously dominant strategy of data quality degradation, i.e., load shedding (Babcock et al., 2004; Tatbul et al., 2007), with resource elasticity. One of the first publications was authored by Schneider et al. (2009), which proposed the parallelization of stream processing operations with System S. Because this first approach only considered stateless operators, the authors complemented their approach in a succeeding publication to consider the replication of stateful operators (Gedik et al., 2014). Besides the elasticity extension to System S, there are also several proposed extensions to Apache Storm, which replace the default scheduler with custom implementations to optimize the parallelization of operators as well as the placement thereof on different computational resources. Two of these approaches have been presented by Aniello et al. (2013) and Xu et al. (2014). These two publications present threshold-based custom schedulers, which can adopt the topology deployment at runtime, depending on the incoming data volume and the actual load for Apache Storm. Although any replication of a specific operator provides additional processing capabilities, it needs to be noted that any reconfiguration of the topology enactment has a negative impact on the processing performance. To minimize these reconfiguration aspects, Stela (Xu et al., 2016), introduces new performance indicators to focus on the actual throughput of the SPE and to reduce any reconfiguration aspects.

To extend the rather static aspect of the threshold-based scaling approaches, Heinze et al. (2015) propose a threshold-based resource optimization, whose thresholds are adopted based on an online learning mechanism within a custom SPE. This allows resource optimization to adapt the otherwise fixed thresholds, which are predefined before the topology enactment, at runtime to improve the resource utilization based on actual monitoring data. SEEP (Castro Fernandez et al., 2013), another custom SPE, also proposes a simple threshold-based replication mechanism. In contrast to the other already discussed approaches, SEEP focuses on stateful operators and employs a dedicated fault tolerance mechanism.

Besides the basic replication approaches, there are also some works that optimize specific aspects for the topology enactment. One of these aspects is the partitioning of data to optimize the data flow among the operators, especially regarding stateful operators. The Streamcloud (Gulisano et al., 2012) SPE, proposes a mechanism to partition the incoming data to distribute it efficiently among different replicas of one operator type. Another approach for optimizing the overall efficiency of a topology enactment is to optimize the placement of operators within a potential heterogeneous pool of computational resources. Cardellini et al. (2015) propose an extension to Apache Storm, which considers an optimal placement of operators in terms of QoS criteria on different cloud resources. Furthermore, De Matteis and Mencagli (2016) present a predictive approach to minimize the latency and improve the energy efficiency of the SPE. This approach furthermore allowed them to reduce the reconfiguration of SPEs, which is also one of the objectives in our approach. The last notable approach for optimizing the topology enactment on cloud resources is to optimize the deployment of operators according to their specific processing tasks. Hanna et al. (2016) consider different types of VMs, e.g., with an emphasis on CPU or GPU, and optimize the deployment based on the suitability of these machines to conduct specific operations, e.g., matrix multiplications are significantly faster when executed on the GPU.

Although the literature already provides different optimization approaches, to the best of our knowledge, none of these approaches considers the BTU aspect of VMs when optimizing processing resources as proposed in this paper. Furthermore, most of the discussed approaches only aim at optimizing the amount of replicas for processing operators, but do ignore the reconfiguration overhead during the topology enactment.

8 CONCLUSION

Within this paper, we have discussed the most important requirements for optimizing data stream processing in volatile environments. Based on these requirements, we have developed an extensive system model for which we have presented a BTU-based optimization approach. This optimization approach has also been evaluated against a threshold-based approach, which is currently commonly used in the literature. The evaluation has shown that the BTU-based approach results in a more cost-efficient manner due to the better usage of the leased VMs and less reconfigurations to the SPE while still maintaining a higher QoS most of the time compared to the threshold-based approach. Furthermore, as a result of the evaluation, we have also identified one potential evolution for our BTU-based approach, namely the addition of a more sophisticated predictive component. So far we only consider the trend for upscaling operator instances, but we do not consider historical information nor other monitoring information, e.g., as suggested by Copil et al. (2016), for downscaling purposes, which could yield even better results in the 2-level approach. In our future work, we plan to apply our BTU-based approach to hybrid clouds. This requires an extension of the optimization model regarding the network capabilities among these clouds. Furthermore, we plan to investigate the actual topology in more detail, e.g., to identify critical paths or high volume operators, such as the operators O2 and O6 in our topology. These details are a promising source to apply different scaling priorities, especially for downscaling to avoid oscillating effects.

ACKNOWLEDGMENTS

This paper is supported by TU Wien research funds and by the Commission of the European Union within the CREMA H2020-RIA project (Grant agreement no. 637066).

REFERENCES

Abadi, D. J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A., Rasin, A., and Ryvkina, E. (2005). The Design of the Borealis Stream Processing Engine. In *Conference on Innovative Data Systems Research*, pages 277–289.

Andonov, R., Poirriez, V., and Rajopadhye, S. (2000). Unbounded knapsack problem: Dynamic programming revisited. *European Journal of Operational Research*, 123(2):394–407.

Aniello, L., Baldoni, R., and Querzoni, L. (2013). Adaptive online scheduling in Storm. In *7th International Conference on Distributed Event-based Systems (DEBS)*, pages 207–218. ACM.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., and Zaharia, M. (2010). A view of cloud computing. *Communications of the ACM*, 53(4):50–58.

Babcock, B., Datar, M., and Motwani, R. (2004). Load shedding for aggregation queries over data streams. In *20th International Conference on Data Engineering*, pages 350–361. IEEE.

Balakrishnan, H., Balazinska, M., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Galvez, E., Salz, J., Stonebraker, M., Tatbul, N., Tibbetts, R., and Zdonik, S. (2004). Retrospective on aurora. *Proceedings of the VLDB Endowment*, 13(4):370–383.

Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. (2015). Apache Flink^{TM}: Stream and Batch Processing in a Single Engine. *Data Engineering Bulletin*, 38(4):28–38.

Cardellini, V., Grassi, V., Lo Presti, F., and Nardelli, M. (2015). Distributed qos-aware scheduling in storm. In *9th International Conference on Distributed Event-Based Systems (DEBS)*, pages 344–347. ACM.

Castro Fernandez, R., Migliavacca, M., Kalyvianaki, E., and Pietzuch, P. (2013). Integrating scale out and fault tolerance in stream processing using operator state management. In *International Conference on Management of Data (SIGMOD)*, pages 725–736.

Copil, G., Moldovan, D., Truong, H.-L., and Dustdar, S. (2016). rSYBL: A Framework for Specifying and Controlling Cloud Services Elasticity. *ACM Transactions on Internet Technology (TOIT)*, 16(3):18.

De Matteis, T. and Mencagli, G. (2016). Keep calm and react with foresight: strategies for low-latency and energy-efficient elastic data stream processing. In *21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12. ACM.

Gedik, B., Andrade, H., Wu, K.-L., Yu, P. S., and Doo, M. (2008). SPADE: The System S Declarative Stream Processing Engine. In *International Conference on Management of Data (SIGMOD)*, pages 1123–1134. ACM.

Gedik, B., Schneider, S., Hirzel, M., and Wu, K.-L. (2014). Elastic scaling for data stream processing. *Transactions on Parallel and Distributed Systems*, 25(6):1447–1463.

Genaud, S. and Gossa, J. (2011). Cost-wait trade-offs in client-side resource provisioning with elastic clouds. In *International Conference on Cloud computing (CLOUD)*, pages 1–8. IEEE.

Gulisano, V., Jimenez-Peris, R., Patino-Martinez, M., Soriente, C., and Valduriez, P. (2012). Streamcloud: An elastic and scalable data streaming system. *IEEE Trans. on Parallel and Distributed Systems*, 23(12):2351–2365.

Hanna, F., Marchal, L., Nicod, J.-M., Philippe, L., Rehn-Sonigo, V., and Sabbah, H. (2016). Minimizing rental cost for multiple recipe applications in the cloud. In *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 28–37. IEEE.

Heinze, T., Roediger, L., Meister, A., Ji, Y., Jerzak, Z., and Fetzer, C. (2015). Online parameter optimization for elastic data stream processing. In *6th ACM Symposium on Cloud Computing*, pages 276–287. ACM.

Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R. H., Shenker, S., and Stoica, I. (2011). Mesos: A platform for fine-grained resource sharing in the data center. In *8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, volume 11, pages 22–22.

Hochreiner, C. (2017). VISP Testbed – A Toolkit for Modeling and Evaluating Resource Provisioning Algorithms for Stream Processing Applications. In *9th ZEUS Workshop (ZEUS 2017)*, pages 37–43. CEUR-WS.

Hochreiner, C., Schulte, S., Dustdar, S., and Lecue, F. (2015). Elastic Stream Processing for Distributed Environments. *IEEE Internet Computing*, 19(6):54–59.

Hochreiner, C., Vögler, M., Schulte, S., and Dustdar, S. (2016a). Elastic Stream Processing for the Internet of Things. In *9th International Conference on Cloud Computing (CLOUD)*, pages 100–107. IEEE.

Hochreiner, C., Vögler, M., Waibel, P., and Dustdar, S. (2016b). VISP: An Ecosystem for Elastic Data Stream Processing for the Internet of Things. In *20th International Enterprise Distributed Object Computing Conference (EDOC)*, pages 19–29. IEEE.

Kulkarni, S., Bhagat, N., Fu, M., Kedigehalli, V., Kellogg, C., Mittal, S., Patel, J. M., Ramasamy, K., and Taneja, S. (2015). Twitter heron: Stream processing at scale. In *International Conference on Management of Data (SIGMOD)*, pages 239–250. ACM.

Lohrmann, B., Janacik, P., and Kao, O. (2015). Elastic stream processing with latency guarantees. In *35th International Conference on Distributed Computing Systems (ICDCS)*, pages 399–410. IEEE.

McAfee, A., Brynjolfsson, E., Davenport, T. H., Patil, D., and Barton, D. (2012). Big data. *The management revolution. Harvard Business Review*, 90(10):61–67.

Nakajima, S. (1988). Introduction to tpm: Total productive maintenance. *Productivity Press, Inc.*

Satzger, B., Hummer, W., Leitner, P., and Dustdar, S. (2011). Esc: Towards an elastic stream computing platform for the cloud. In *International Conference on Cloud Computing (CLOUD)*, pages 348–355.

Schneider, S., Andrade, H., Gedik, B., Biem, A., and Wu, K.-L. (2009). Elastic scaling of data parallel operators in stream processing. In *International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12.

Schulte, S., Hoenisch, P., Hochreiner, C., Dustdar, S., Klusch, M., and Schuller, D. (2014). Towards process support for cloud manufacturing. In *18th International Enterprise Distributed Object Computing Conference (EDOC)*, pages 142–149. IEEE.

Tatbul, N., Çetintemel, U., and Zdonik, S. (2007). Staying fit: Efficient load shedding techniques for distributed stream processing. In *Proceedings of the VLDB Endowment*, pages 159–170. VLDB Endowment.

Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J. M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., Bhagat, N., Mittal, S., and Ryaboy, D. (2014). Storm@twitter. In *International Conference on Management of Data (SIGMOD)*, pages 147–156.

Xu, J., Chen, Z., Tang, J., and Su, S. (2014). T-storm: traffic-aware online scheduling in storm. In *34th International Conference on Distributed Computing Systems (ICDCS)*, pages 535–544. IEEE.

Xu, L., Peng, B., and Gupta, I. (2016). Stela: Enabling stream processing systems to scale-in and scale-out on-demand. In *International Conference on Cloud Engineering (IC2E)*. IEEE.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster computing with working sets. *HotCloud*, 10:10–17.