FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

An Elastic Data Stream Processing Ecosystem for Distributed Environments

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

eingereicht von

Christoph Hochreiner
Matrikelnummer 00726292

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dr. Schahram Dustdar
Zweitbetreuung: Assistant Prof. Dr.-Ing. Stefan Schulte

Diese Dissertation haben begutachtet:

Schahram Dustdar	Harald C. Gall	Gertrude Kappel

Wien, 11. Jänner 2018

Christoph Hochreiner

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

An Elastic Data Stream Processing Ecosystem for Distributed Environments

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Christoph Hochreiner
Registration Number 00726292

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dr. Schahram Dustdar
Second advisor: Assistant Prof. Dr.-Ing. Stefan Schulte

The dissertation has been reviewed by:

Schahram Dustdar	Harald C. Gall	Gertrude Kappel

Vienna, 11th January, 2018

Christoph Hochreiner

Erklärung zur Verfassung der Arbeit

Christoph Hochreiner
Vorgartenstrasse 129-143/1/6/18
1020 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 11. Jänner 2018

Christoph Hochreiner

Danksagung

Diese Abschlussarbeit bildet das Finale meines 10-jährigen Informatik Studiums, welches einen persönlichen Meilenstein darstellt. Die Erreichung dieses Ziel wäre jedoch nicht ohne der Mithilfe vieler Menschen möglich gewesen, denen ich an dieser Stelle meinen Dank aussprechen möchte. Zuallererst möchte ich mich bei meinem Doktorvater Prof. Schahram Dustdar für die Chance bedanken, meine Dissertation in der Distributed Systems Group (DSG) in einer inspirierenden Arbeitsatmosphäre zu schreiben. Ein großer Dank gilt auch Assistant Prof. Stefan Schulte, der mich in seinen Forschungsprojekten aufgenommen hat und mir den Freiraum gab, meine Forschungsideen eigenständig zu verfolgen und zu entwickeln. Des Weiteren möchte ich mich bei Prof. Harald Gall und Prof. Gertrude Kappel bedanken, die sich als Gutachter zur Verfügung gestellt haben.

Ich bedanke mich zudem bei meinen Kollegen in der DSG, die mich in den letzten Jahren begleitet haben und mit denen ich viele interessante Projekte umsetzen konnte. An dieser Stelle möchte ich Michael Vögler hervorheben und mich für die vielen anregenden Diskussionen und die Motivation in den entscheidenden Momenten bedanken.

In diesem Zuge darf auch die Stammtischrunde nicht vergessen werden, welche immer ein offenes Ohr für meine Forschungsfragen hatte und wertvolles Feedback gab. Ebenso gilt der Dank meinen Eltern, die mich während meines gesamten Studiums unterstützt haben und meinen Schwestern, welche mich immer wieder motiviert haben nicht aufzugeben. Hier möchte ich besonders Eleonora danken, welche mir bei der Korrektur der Arbeit half. Zu guter Letzt gebührt mein Dank Gudrun für ihre Unterstützung und dass sie mich immer wieder aufgebaut hat, wenn ich mit meiner Arbeit nicht zufrieden war.

Acknowledgements

The research leading to this thesis has received funding from the European Community's Seventh Framework Programme (FP7-ICT) under grant agreement 318201 (SIMPLICITY), from the European Community's Horizon 2020 Framework Programme under grant agreement 637066 (CREMA), and TU Wien research funds.

Kurzfassung

Im Laufe der letzten Jahre konnte ein konstantes Wachstum von Datenströmen beobachtet werden. Bisher waren diese Datenströme in der Regel primär im Betrieb von sozialen Netzwerken, in der Finanzindustrie oder in der Medizintechnik zu finden. Durch die zunehmende Verbreitung des Internets der Dinge existieren heute jedoch Sensor-basierte Datenquellen in vielen verschiedenen Anwendungsbereichen. Diese Datenquellen liefern volatile Datenströme, welche jedoch von aktuellen Datenverarbeitungssystemen nicht effizient verarbeitet werden können, da diese nur für konstante Datenströme konzipiert wurden. Des Weiteren sind neue Benutzergruppen mit den bestehenden Systemen oft überfordert, da diese in erster Linie für Experten entwickelt wurden. Aus diesem Grund ist es notwendig, die Architektur von etablierten Datenverarbeitungslösungen zu überdenken, um dem verteilten Aspekt des Internets der Dinge gerecht zu werden und neue Ansätze für die Erstellung von benutzerfreundlichen Anwendungen zur Datenstromverarbeitung zu entwickeln.

Daher wird in dieser Dissertation das VISP-Ökosystem präsentiert, welches sowohl die Bedürfnisse neuer Benutzergruppen als auch die verteilte Struktur des Internets der Dinge berücksichtigt. In diesem Zusammenhang wurde auch eine neue Konfigurationssprache für Anwendungen zur Datenstromverarbeitung geschaffen, da viele nicht-funktionale Anforderungen von bestehenden Konfigurationssprachen nur eingeschränkt unterstützt werden. Zusätzlich zu den Grundfunktionalitäten für die Erstellung und Ausführung von Anwendungen zur Datenstromverarbeitung wurden zwei Ansätze entwickelt, um die benötigten IT-Ressourcen elastisch an die sich ändernden Anforderungen anpassen zu können. Der erste Ansatz basiert auf einer kontinuierlichen Analyse von verschiedenen Kennzahlen, welche bei Bedarf eine Veränderung der Ressourcenkonfiguration veranlasst. Der zweite Ansatz ist eine Verfeinerung des ersten Ansatzes, da dieser neben den Kennzahlen auch zusätzliche Faktoren wie die Mietvereinbarungen für Ressourcen berücksichtigt und daher in vielen Fällen unnötige Anpassungen vermeidet. Beide Ansätze wurden auf Basis verschiedener Szenarien evaluiert und ermöglichen eine Kostenreduktion ohne die Verarbeitungsgeschwindigkeit zu beeinträchtigen.

Abstract

In the last couple of years, we have observed a trend towards an ever-growing number and volume of data streams. Up to now, these data streams were mainly originating from social media services running in the cloud but today the emergence of the Internet of Things (IoT) also contributes to the growth of data streams. Besides the growth of the data volume, the IoT also introduces several new challenges, like the geographically distributed locations of IoT-devices, i.e., data sources and processing capabilities, as well as a differentiation of the user base who uses Stream Processing Applications (SPAs). Previously, SPAs were only used by data stream processing experts to process large data volume primarily for social media, medical or financial purposes in a centralized setting. However, the emergence of the IoT allows a larger user base, like companies from the manufacturing domain or even individual users, to process data streams to extract valuable insights. To address these challenges, it is required to evolve the system design of today's stream processing engines and create an ecosystem for data stream processing, which considers all aspects of designing and operating SPAs.

Therefore, we introduce the VISP Ecosystem in this thesis, which provides a holistic approach for creating SPAs and propose novel concepts to operate SPAs in a distributed environment. To improve the creation of SPAs, we present a novel description language for SPAs that supports distributed deployments as well as several non-functional aspects for SPAs that are not considered in today's approaches. In addition to the fundamental aspects of designing and operating SPAs, we also introduce two resource provisioning approaches. These two approaches use the resource elasticity provided by the cloud computing paradigm to reduce the operational cost for running SPAs under volatile data volume. The first resource provisioning approach is threshold-based approach and can find the optimal resource configuration depending on the current data volume for the SPA. This dynamic resource provisioning approach allows this approach to outperform established fixed resource provisioning strategies regarding cost efficiency. The second approach represents an evolution of the first approach by considering additional external aspects like the billing time units to avoid any unnecessary operational overhead for updating the resource configuration. According to our evaluation, we can see that our second approach outperforms the first one for most real-world scenarios and allows for an even more cost-efficient operation of SPAs while ensuring the timely processing of data streams.

Contents

List of Figures

List of Tables

List of Publications

The work in this thesis is based on research that has been published in the following conference papers and journal articles. For a full publication list of the author please refer to the Curriculum Vitæ in Appendix B.

- C. Hochreiner, S. Schulte, S. Dustdar, and F. Lecue, "Elastic Stream Processing for Distributed Environments", *IEEE Internet Computing*, vol. 19, no. 6, pp. 54–59, 2015. DOI: 10.1109/mic.2015.118
- C. Hochreiner, M. Vögler, S. Schulte, and S. Dustdar, "Elastic Stream Processing for the Internet of Things", in *9th International Conference on Cloud Computing (CLOUD)*, IEEE, 2016, pp. 100–107. DOI: 10.1109/cloud.2016.0023
- C. Hochreiner, M. Vögler, P. Waibel, and S. Dustdar, "VISP: An Ecosystem for Elastic Data Stream Processing for the Internet of Things", in *20th International Enterprise Distributed Object Computing Conference (EDOC)*, IEEE, 2016, pp. 19–29. DOI: 10.1109/edoc.2016.7579390
- C. Hochreiner, "VISP Testbed – A Toolkit for Modeling and Evaluating Resource Provisioning Algorithms for Stream Processing Applications", in *9th ZEUS Workshop (ZEUS)*, CEUR-WS, 2017, pp. 37–43
- C. Hochreiner, M. Vögler, S. Schulte, and S. Dustdar, "Nomadic Applications Traveling in the Fog", in *2nd EAI International Conference on Cloud Networking for Internet of Things Systems (CN4IoT)*, vol. 189, Springer International Publishing, 2018, pp. 151–161. DOI: 10.1007/978-3-319-67636-4_17
- C. Hochreiner, M. Vögler, S. Schulte, and S. Dustdar, "Cost-efficient enactment of stream processing topologies", *PeerJ Computer Science*, vol. 3, e141, 2017. DOI: 10.7717/peerj-cs.141
- C. Hochreiner, M. Nardelli, B. Knasmüller, S. Schulte, and S. Dustdar, "VTDL: A Notation for Stream Processing Applications (accepted for publication)", in *12th International Symposium on Service-Oriented System Engineering (SOSE)*, IEEE, 2018, NN–NN

CHAPTER 1

Introduction

The rise of the Internet of Things (IoT), a new paradigm where devices are equipped to communicate with each other, leads to more and more IoT devices in the industry as well as the private sector [8], [9]. Most of these IoT devices offer data streams that can be used to extract information about their surrounding environment, like the weather, or their internal state like the production progress for manufacturing machines. This data extraction is conducted by Stream Processing Applications (SPAs) that are composed of different operators which perform dedicated tasks, such as data filtering or data aggregation [10]. The composition of these operators is defined by a topology which connects the operators based on a directed acyclic graph that represents the data flow through the SPA. To run an SPA, the topology needs to be operated by an Stream Processing Engine (SPE). An SPE provides a runtime environment for SPAs which is in charge of two major tasks. The first task is to provide enough computational resources for the operators that perform the data processing and the second task is to ensure the data flow among the operators based on the topology.

The first SPEs [11]–[13] emerged from database systems over a decade ago to process constant data streams on single computers or computer clusters. These SPEs were designed to extract information from data streams, but the high popularity of social networks introduced a new challenge for SPEs. This new challenge was the huge data volume of data streams which lead to a redesign of the first SPEs to address this shortcoming [14]–[16] This redesign built on top of the principles of cloud computing enabling a high degree of parallel data processing on virtually unlimited resources [17], [18]. While these SPEs represent the current state-of-the-art, the rise of the IoT introduces additional challenges for SPEs. These challenges originate from the distributed locations of IoT devices as well as the growing user base for SPAs which is discussed in detail in Section 1.1. To address these challenges, we propose to evolve the concept of data stream processing again and create a holistic ecosystem around SPEs.

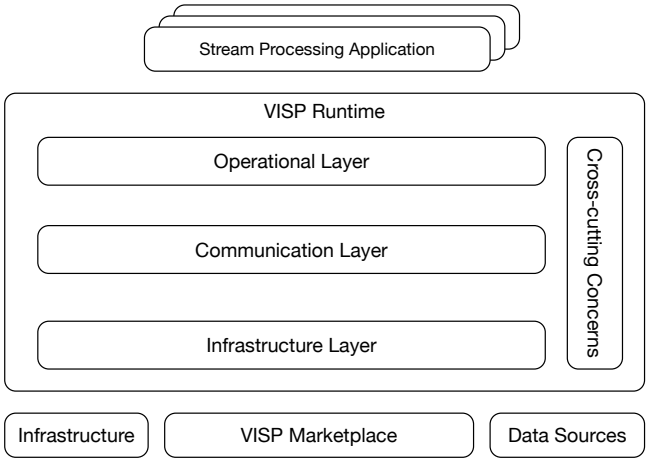


Figure 1.1: Vienna Stream Processing Ecosystem

By introducing the VIenna ecosystem for Stream Processing (VISP) as shown in Figure 1.1, we provide an ecosystem which allows designing and operating SPAs as well as optimizing their data throughput at run time. This ecosystem is built around a novel SPE, the VISP Runtime and other supporting components like the VISP Marketplace whose components are presented in detail in Chapter 5.

1.1 Problem Statement

Up to now, data stream processing is mainly used for social networks [14], [19] or in the financial domain [10]. Nevertheless, due to the continuous rise of the IoT, data stream processing becomes also relevant for other domains. These other domains range from factory owners monitoring the health and productivity of their manufacturing machines based on sensor data up to individual private user. Today, numerous private users operate IoT devices, like health monitors [20] or weather stations [21], to collect information about themselves or their environment. To cater for this growing user base, the focus of data stream processing needs to be revised to consider the novel challenges that stem from the rise of the IoT.

The first challenge is the geographic distribution of different data sources, i.e., sensors mounted on IoT devices, which are often far away from computational resources [22]. This requires the data streams to be transferred either to rather small and expensive local private clouds or to centralized public clouds for processing [23]. After the data is processed, it has to be delivered to the designated receiver, which can be in the proximity of the data source, e.g., actuators for the IoT device, or at an arbitrary geographic location, e.g., a system operator who wants to monitor the status of manufacturing machines on a mobile device. Nowadays, SPEs are only designed to operate in one location and cannot manage the data flow and processing across different geographic locations.

Therefore, SPA operators tackle this challenge by either deploying multiple SPE instances close to the data sources on private clouds or by deploying one centralized SPE on a public cloud [22], [24]. Both approaches exhibit major problems: The centralized approach exposes a low network-efficiency as well as a high latency because the data needs to be transferred to the SPE over the network, i.e., the Internet, before it can be processed [25]. The distributed approach promises low latency but requires high setup cost by establishing the communication among different SPEs running in different geographic locations. Furthermore, SPEs often need to deal with resource constraints like limited computational resources in a specific location due to the deployment on a computer cluster instead of an cloud. This challenge calls for a novel approach for data stream processing, which integrates the different geographic locations and provides a transparent user interface for managing and controlling SPAs in a similar manner as it is the case for SPEs running in only one location.

The second challenge for SPEs considers the economic aspects of data processing. Although SPEs are highly efficient regarding data processing, they struggle with volatile data volume over time [1]. Most SPEs operate on a fixed amount of computational resources and cannot adapt to changes of the data volume at run time [2]. One solution for this issue is the over-provisioning of computational resources so that the SPE can process any volume of incoming data while complying with given Service Level Agreements (SLAs) [26]. Although 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. A more economically feasible approach is under-provisioning, where an SPE is only equipped with computational resources to cover most of the incoming data volume. However, in the case of under-provisioning, the SPE may violate given SLAs for high volume scenarios because the data cannot be processed in time. A potential solution approach to this challenge is to elastically provision computational resources on demand based on the data volume emitted by the data sources. This approach allows to update the amount of computational resources at run time. Furthermore, this results in low operational costs compared to the over-provisioning approach and a better SLA compliance than the under-provisioning approach.

The third challenge considers the usability of SPEs. Until now, SPAs are only used by experts who are both able to create SPAs as well as administrate SPEs. Due to the continuous expansion of IoT devices, this user base is going to grow and it is required to introduce an abstraction for the predominating code-based representation of SPAs towards a flexible, extensible and easy readable description of topologies. Furthermore, it is also necessary to evolve the current design process, where each SPA is implemented from scratch to a reuse-oriented one similar to the library ecosystem used in software engineering [27]. In addition it is required to enable the use of SPEs in a Platform as a Service (PaaS) manner, which does not require any expert knowledge to operate SPAs.

The three above-mentioned challenges cannot be addressed with currently existing SPEs and require a novel system design for SPEs as well as a holistic data stream processing ecosystem.

1.2 Research Questions

The motivation for this thesis is based on the challenges presented in Section 1.1 and formulated as the following research questions.

Research Question I

How can geographically distributed data streams be processed efficiently?

The rise of the IoT results in more and more geographically distributed IoT devices, whose data streams are processed by SPAs running on SPEs in centralized clouds or secluded on private clouds next to the data sources. This data processing deployment is a result of the system design of today's SPEs that are designed to run in only one geographic location. Since the current predominant system design for SPEs exhibits major drawbacks, as discussed in Section 1.1, it is required to come up with a new system design for SPEs. Such a new system design must be able to deal with the geographic distribution of data streams as well as of computational resources. Therefore, it is required to design a novel generation of SPEs which enable a network-efficient data flow for the data streams while ensuring a transparent management of the distributed geographic locations for operating SPAs.

Research Question II

How can stream processing applications be described to allow a distributed deployment model and respect service level objectives?

Based on the distributed runtime environment provided by novel SPEs, it is also required to consider the different geographic locations for SPAs. Up to now, SPAs are only designed to run in one specific geographic location and their design only considers the choreography of the individual operators. When we operate SPAs on a distributed SPE, it is required to consider the geographic location of fixed entities like sensors on IoT devices. Furthermore, there may also be legal restrictions to transfer privacy-sensitive data outside of legal jurisdictions or even just outside a company's premises that host the data source. In addition, it is also required to explicitly model Service Level Objectives (SLOs) for SPAs. Any distributed system introduces higher data transfer times among the individual system components compared to a centralized deployment which needs to be considered for the operation of SPAs. To address this lack of description functionalities for SPAs, it is required to investigate how already established notions for describing SPAs can be extended to cater for the new challenges introduced by distributed SPEs.

Research Question III

How can stream processing applications be optimally executed on computational resources?

Besides the geographic distribution aspect, it is also necessary to consider the economical aspect of operating SPAs. For this aspect, it is crucial to minimize the computational resources that are used by SPAs to reduce the operational cost. However, it is also required to comply with predefined SLOs, e.g., the maximum processing duration, to avoid penalty cost for delayed processing. While this compliance is rather simple for a constant data volume, it is challenging for changing data volume, e.g., a varying number of active manufacturing machines through the course of the day. At the moment, the most common approach is an over-provisioning approach, where the SPE has enough computational resources at its hand that can be assigned to the SPAs to handle any data volume provided by the data sources. Although this approach allows a high SLA compliance, it results in unnecessary high resource cost. Therefore, it is required to adopt elastic provisioning approaches and investigate how these approaches can reduce the operational cost without causing penalty cost for delayed processing.

1.3 Scientific Contributions

Contribution I

A system design for a holistic stream processing ecosystem in distributed environments

The constant rise of IoT devices leads to numerous unbound data streams in different geographic locations. These data streams can be hardly processed in a network-efficient manner, due to the centralized architecture of today's SPEs. To address this shortcoming, we analyze how SPAs are designed and operated with today's SPEs. Based on this analysis and the challenges presented in Section 1.1, we revise the concept on how SPAs are designed or operated today and introduce the VISP Ecosystem. The VISP Ecosystem features not only a novel SPE, the VISP Runtime, which is capable of processing data streams across distributed runtime environments, but also presents a novel approach on how SPAs are designed. This novel approach is a decomposition of SPAs into self-contained operators, which can be used as building blocks. These building blocks can be shared among different SPAs to reduce the implementation effort compared to today's model, where each SPA is implemented from scratch. The details for this contribution are presented in Chapter 5 and Contribution I was originally presented in [3].

Contribution II

An approach to define quality of service- and location-specific aspects for stream processing applications

While Contribution I introduces a new ecosystem for designing and operating SPAs, it is also required to revise the notation for topologies that define the structure of SPAs. Until now, existing notations for topologies only consider centralized runtime environments which do not require any deployment location constraints because there is only one possible deployment location. Nevertheless, due to the new possibilities provided by the VISP Ecosystem, it is required to extend existing notations. To address this issue, we extend the Stream Processing Language (SPL), which was introduced for IBM System S [28] and serves as a commonly used notation for SPAs. Our novel notation, the Vienna Topology Description Language (VTDL), also considers different deployment locations and computational resource preferences as well as deployment restrictions. In addition, we also introduce a fine-granular definition model for SLOs, which provides the foundation for any resource optimization and scaling activities as presented by Contribution III. The details for this contribution are presented in Chapter 6 and Contribution II was originally presented in [7].

Contribution III

A cost-efficient resource provisioning approach for stream processing applications

The first two contributions provide the foundation for designing and operating SPAs in a distributed environment. However, in today's industry, there is also a constant cost pressure, which needs to be addressed. In terms of data stream processing, the two most prominent cost factors are the cost for computational resources and potential penalty cost, if the data processing does not meet predefined SLOs as defined by given SLAs. In order to solve this issue, we design two different resource provisioning approaches. These approaches are not only in charge of providing an initial deployment of the SPA, but also to update the resource configuration at run time to comply with the processing requirements of the SPA. To evaluate the applicability of the resource provisioning approaches under varying data volume, we implement them within the VISP Runtime and conduct comprehensive experiments within a single location as well as in a distributed environment. The details for this contribution are presented in Chapter 7 and Chapter 8 and Contribution III was originally presented in [2], [6].

1.4 Organization of this Thesis

The remainder of this thesis is structured as follows.

Chapter 2 provides background information on fundamental concepts used in this thesis like data stream processing, resource provisioning and the IoT.

Chapter 3 discusses the related work to the contributions presented in Section 1.3.

Chapter 4 presents our motivational scenario which is used throughout the thesis to motivate the requirements and evaluate the contributions.

Chapter 5 presents the requirements and the system design for the VISP Ecosystem.

Chapter 6 introduces the VTDL, a novel notation for describing SPAs.

Chapter 7 presents an threshold-based resource provisioning approach for SPAs.

Chapter 8 introduces a novel resource provisioning approach for SPAs, which focusses on a high resource usage for cloud resources.

Chapter 9 concludes the thesis by summarizing the contributions and providing an outlook to future research challenges.

CHAPTER 2

Background

In this chapter, we introduce three basic concepts that are used in this thesis. First, we discuss the terminology for data stream processing. Then, we present the most common resource provisioning approaches, provide a short introduction to the IoT and highlight why the IoT poses new challenges for data stream processing.

2.1 Data Stream Processing

The goal of data stream processing is to process live data without any delay to extract information for users. The domain of data stream processing has been established in the course of the last 15 years as discussed in Section 3.2 and the data stream processing community has established a terminology which is presented in this section [29].

In terms of data stream processing, there are two different viewpoints that are used within this thesis: a user-oriented and an operational one whose concepts and terms are visualized in Figure 2.1. The user-oriented viewpoint (shown in normal font in Figure 2.1) describes the overall task of data stream processing as it is perceived by users who are only interested in the result, i.e., the extracted information. For a user, a *Stream Processing Application* is in charge of extracting information from *Data Streams*. Such an SPA is a black box from the user's point of view that is in charge of a specific task, e.g., monitoring manufacturing machines. The SPAs are supplied with data streams, which are composed of a continuous sequence of *Data Items*. These data items contain the raw information, e.g., sensor data, which needs to be processed by the SPA to extract insights. The data items are generated by *Data Sources*, which can either be physical devices, e.g., sensors mounted on IoT devices, or external software services, like online news portals. After processing the data items, the results are provided as a continuous stream of data items to *data sinks*, which can either be users who monitor IoT devices, arbitrary software services or other IoT devices that use the information for further activities.

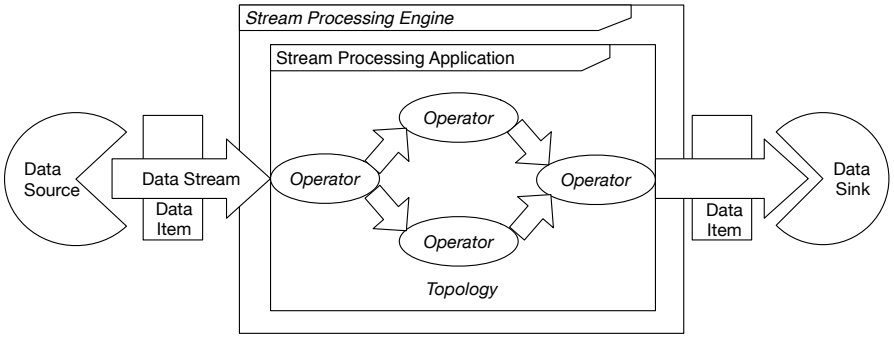


Figure 2.1: Exemplary Stream Processing Scenario

Besides the user-oriented viewpoint, there is also an operational one, which describes the implementation of the actual information extraction functionality shown in italic font in Figure 2.1. SPAs are typically managed by a *Stream Processing Engine* that is in charge of handling the data streams and providing enough computational resources for *Operators* to process the data. Each SPA is composed of arbitrarily many operators whereas each performs a specific task, such as data filtering or data aggregation [10]. The composition of the operators is defined by a *Topology* which describes the flow of data streams between the individual operators. This topology is represented by a directed acyclic graph that is designed individually for each SPA at design time. Whenever the SPA is started, this topology serves as a blueprint which lists all operators that need to be instantiated on computational resources and instructs the SPE on how to wire the individual operators to achieve the desired functionality of the SPA.

2.2 Resource Provisioning

Resource provisioning, i.e., the provisioning of computational resources for software-based services, is one of the most important domains for operating software systems because software systems like SPAs cannot be executed without any computational resources. The literature distinguishes between two fundamental approaches, namely fixed resource provisioning and elastic resource provisioning [30], which are discussed in detail in the remainder of this section.

2.2.1 Fixed Resource Provisioning

The fixed resource provisioning approach represents a very simple and therefore the most common resource provisioning approach until the emergence of the cloud computing paradigm [30]. For this approach, the computational resources are provisioned upfront based on arbitrary estimations. After the software system is started, i.e., at run time, these computational resources do not change. This approach is a perfect solution for software systems that deal with a constant load, e.g., data volume, which can be estimated beforehand, e.g., based on historic data.

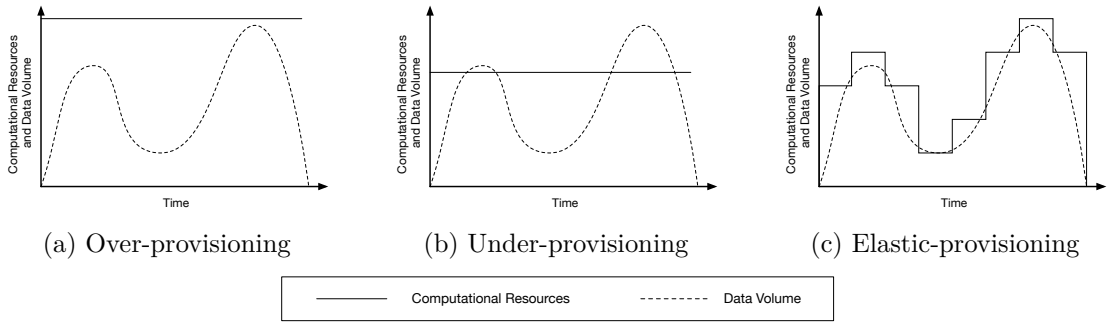
Nevertheless, systems with a fixed resource provisioning configuration can face challenges when the load for the software system changes. These load changes can occur due to predictable changes, e.g., day and night cycles [31], [32] or due to peak scenarios, which are triggered by extraordinary events like elections or catastrophes that can result in very high loads for a short time span [33].

To ensure a high Quality of Service (QoS), system providers can be pessimistic when calculating the resource requirements for a software system and plan significantly more resources than the system requires in an average case scenario [34]. This approach is called *over-provisioning* and is shown in Figure 2.2a. It shows that the provisioned computational resources are enough to process any data volume in the shown timeframe without any negative impact to the QoS. Although this approach allows for a high QoS, the resource usage of the computational resources is rather low because a large part of the computational resources is not used for most of the time. Due to the fact that provisioned computational resources need to be paid regardless of their actual usage, the over-provisioning approach is economically very inefficient and results in unnecessary high operational cost.

To reduce the operational cost, some system providers pursue a more optimistic approach and only provision computational resources to cover most data volume scenarios [34]. This approach is called *under-provisioning* and can be seen in Figure 2.2b. Although the under-provisioning approach is sufficient for most data volume, there are some data volume peaks that cannot be dealt with. This results in a lower QoS compared to the over-provisioning approach. Since this approach anticipates situations where the data processing needs to be postponed or data needs to be discarded due to the lack of computational resources, the literature already provides several strategies to actively manage these situations, i.e., to apply *load shedding* [35]–[37]. The simplest strategy is to apply a First-In, First-Out (FIFO) approach, where all data has the same priority and data peaks result in a delay because some data items need to be buffered before they can be processed. Although the FIFO strategy is the fairest approach, it is often more efficient from a users' point of view to apply more sophisticated load shedding strategies, like prioritizing specific data items or processing only the most recent data items to ensure a timely information extraction of the data streams [37].

2.2.2 Elastic Resource Provisioning

Due to the fact that fixed resource provisioning exhibits several disadvantages like the inefficient resource usage or low QoS, the cloud computing paradigm has been introduced [30]. This paradigm builds on the concept of pooling computational resources, like processing capabilities, storage, networks or applications [38], which can then be used by different stakeholders on-demand depending on their actual need and return the resources to the pool when they are not required anymore. This approach is called elastic resource provisioning because it can change the resource configuration at run time to closely couple the computational resources to the data volume as shown in Figure 2.2c.



(a) Over-provisioning (b) Under-provisioning (c) Elastic-provisioning

Figure 2.2: Resource Provisioning Approaches

The pooling of computational resources within one company leads to the creation of private clouds, where different software systems can use the computational resources from a common resource pool according to their demand – as long as the resources are not already used by other software systems [30]. This approach enables system providers to apply an under-provisioning strategy when starting the system and change the resources on demand when more resources are needed to cover peak loads. Therefore, this elastic approach allows for a better cost structure because resources are only provisioned when they are needed to maintain a high level of QoS [34]. Although the pooling of computational resources within a private cloud allows software systems to comply with most data volume, there may be some events where even these private clouds are not sufficient to process the data without any delay. To cover these scenarios as well, it is required to extend the scope of the pooled resources and pool computational resource among different companies within public or community clouds [38]. These public and community clouds have the same characteristics as private clouds. However due to the fact that they pool more computational resources, they can provide practically unlimited resources for individual software systems. This almost unlimited resource access allows software systems to process any incoming data volume without any delay or data loss and achieve a high QoS at a reasonable cost structure.

2.3 Internet of Things

The Internet of Things is an emerging concept which connects different physical objects with the core of the Internet and therefore represents an extension to today's perception of the Internet according to Shelby [39] that is visualized in Figure 2.3. The core of the Internet is represented by a composition of backbones, routers, and servers that change very seldom and provide the core functionality of the Internet by connecting different stakeholders and providing content all over the world. This Core Internet is accessed and used by the Fringe Internet that consists of personal computers, smartphones, and laptops [39]. Therefore, the Fringe Internet can be considered as an additional layer on top of the Core Internet which is volatile in terms of its structure, since its users follow a day and night cycle [31] and only connect to the Core Internet when they want to use it.

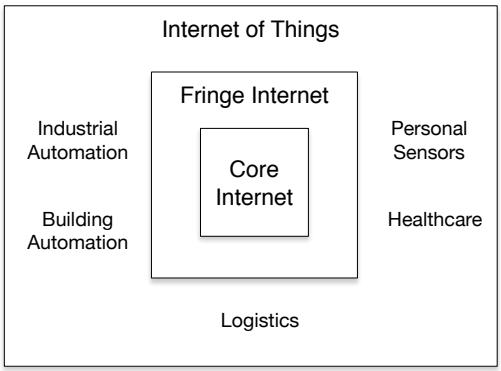


Figure 2.3: Onion Ring Model for the Internet of Things (adapted from [39])

In the past, this Fringe Internet layer has also been used as a proxy to share the data of physical devices that could not be directly connected to the Internet due to a lack of suitable network adapters [39]. Nevertheless, due to the technological advances in decreasing the size of the network adapters, it is nowadays also possible to connect these physical devices directly to the Internet and promote them to IoT devices. These IoT devices form an additional layer on top of the Fringe Internet, the Internet of Things and are capable of autonomously participating in the Internet.

Although IoT devices can be found in almost any domain, there are four core application domains for the IoT [8]: smart environments (building or industrial automation), transportation and logistics, personal sensors and healthcare. These core domains are the main drivers for the realization of the IoT and according to an analysis by Gartner [9], it is assumed that up to 8.4 billion IoT devices are used in 2017, which indicates a growth of 30% compared to 2016. Similar to the Fringe Internet, the IoT also exhibits a very distributed and volatile character in terms of their geographical locations, because the IoT devices may be weather sensors in remote locations or GPS-tracker on moving vehicles that constantly report their current location [40]. This volatile structure of the IoT as well as the lack for fixed geographic locations pose high challenges to systems which process their data since these processing systems, e.g., SPAs need to be constantly updated either in terms of their structure due to changing data sources or in terms of their deployment due to the changing locations of the IoT devices.

CHAPTER 3

Related Work

In this chapter, we present the relevant related work for this thesis, which covers the domains of ecosystems for the IoT, elastic SPEs, resource provisioning approaches for SPAs and topology definition approaches for SPAs.

3.1 Ecosystems for the Internet of Things

Due to the increased spread of IoT devices, the number of IoT ecosystems grew in the last couple of years both in the academic and commercial domain [41], [42]. The goal of these IoT-ecosystems is to integrate IoT devices and provide their data for further processing [43]–[45] as well as to integrate existing SPEs to enable an efficient data processing. Besides the integration of IoT devices and SPEs, there are also first approaches towards building marketplaces to share data and operators among users. Munjin and Morin [46] propose a marketplace, which can be used to connect to IoT devices with external systems, like SPEs. This approach focuses on the connection and integration of IoT devices and does not consider any further data processing or information extraction mechanism. Akpinar et al. [47] propose the ThingStore concept that extends the portfolio of the marketplace by also considering SPAs to process the data as well as raw data provided by IoT devices. This approach follows similar design principles as the VISP Marketplace (see Section 5.3.1) and allows users to create complex SPAs for the IoT. In addition to the operator-oriented marketplaces, there is also the MARSA ecosystem presented by Tien-Dung et al. [48] which focuses only on the monetization of data provided by IoT devices. MARSA allows data providers to offer their data to different users. These users can either integrate the data into their SPAs or refine the data filtering to resell the improved data again on the MARSA marketplace. Although MARSA only focusses on the monetization of data, it is so far the only IoT-related marketplace that supports different business models, ranging from one time payments to subscription models.

Table 3.1: Ecosystems for the Internet of Things

	VISP Ecosystem	Groovestreams	IFTTT	Apache Edgent	ThingStore	ThingWorx	Amazon Web Services IoT	Google Cloud Dataflow	Microsoft Stream Analytics
Runtime	✓	✓		✓	✓	✓	✓	✓	✓
Resource Elasticity	✓				✓	✓	✓	✓	
Marketplace	✓		✓		✓	✓			
Topology Builder	✓	✓	✓						
On-Premise Deployment	✓			✓					
Support for SLAs	✓					(✓)	(✓)	(✓)	

Besides the different proposals from the academic domain, there are also already several commercial ones. Table 3.1 lists the most relevant commercial ones and compares their capabilities to ThingsStore [47] as well as the VISP Ecosystem, which is introduced in Section 5.3. The Groovestreams [49] ecosystem provides a simple cloud-based runtime environment that allows users to upload data streams and analyze them with a set of operators that are provided by Groovestreams. On top of the default functionality to use single operators for data analysis, this ecosystem also provides the capabilities to combine different operators in a linear topology by chaining them. A similar feature set is provided by the IFTTT ecosystem [50] that focuses on the integration of external data providers and SPAs hosted on external service providers. Therefore, this ecosystem provides an extensive marketplace of external SPAs and data sources to be combined by a dedicated topology builder that is provided by IFTTT. This topology builder automatically includes any data transformations between the external SPAs to create a smooth user experience for domain experts that are only focussed on the overall information extraction task and do not care about the concrete implementation. Besides the data processing and operator integration functionality, there is also the ThingWorx [51] platform, which focusses on integrating the data provided by IoT devices with external data processing solutions.

In addition to the dedicated IoT-based ecosystems, the three major cloud service providers also provide services to process data originating from IoT devices: Amazon Web Services IoT [52], Google Cloud Dataflow [53] and Microsoft Stream Analytics [54]. While Amazon already provides a large toolkit of different services tailored to process IoT-based data, the other two providers only offer generic data stream processing capabilities. These ecosystems require dedicated data integration solutions like ThingWorx or Apache Edgent [55]. In contrast to the other ecosystems which operate on a Software as a Service (SaaS)-basis, Apache Edgent is a software library that can be installed anywhere to integrate IoT devices from the edge of the network with cloud-based data processing solution as those presented above.

When comparing these different ecosystems for processing IoT-based data, it can be seen that no ecosystem besides the VISP Ecosystem provides a holistic approach to create SPAs, which is suitable to address the challenge presented in Section 1.1. Although the major cloud service providers already offer reliable data stream processing capabilities, they only provide limited support for users to create new SPAs. This aspect has been addressed by specialized providers like Groovestreams or IFTTT, which implement the concept of operator reuse among different SPAs and provide marketplaces to share the operators. This also applies to the user support for designing topologies. While the major cloud providers only provide a code-based topology definition approach for SPAs, IFTTT provides an easy to use Web-based user interface. This interface allows users to design topologies by simply chaining different operators together while all other providers require the user to provide concrete implementation details for the SPA. Finally, most ecosystems focus on a centralized cloud-based solution approach and do not consider the particularities of privacy-sensitive data, which renders them infeasible for most business scenarios [56].

3.2 Elastic Stream Processing Engines

The roots for data stream processing can be found in the domain of databases. Although databases are a perfect solution for storing and processing static data, they are too inefficient to process continuous data streams without any delay [29]. This challenge lead to the development of the first SPE called Aurora [11] and its successor Borealis [12]. While Aurora is designed in a monolithic manner, Borealis already considers a distributed deployment on a cluster system. This distributed deployment allows to increase the overall fault tolerance for the SPE and to provide a higher performance due to the parallel data processing [57]. Since Borealis has been designed before the emergence of the cloud computing paradigm [38], it is only able to operate on a fixed set of computational resources and needs to apply load-shedding mechanisms to cope with changing data volume [35]. The same also applies for other pioneering SPEs like STREAM [58] or the first versions of IBM System S [13].

Besides these pioneering SPEs, there are also several more recent SPEs that leverage cloud resources to process huge amounts of data almost instantly [59]–[62]. The usage of cloud resources for SPEs has first been proposed by Ishii and Suzumura [63]. Their system design supports an elastic resource provisioning strategy based on cloud resources to cope with changing data volume. This approach results in a cost-efficient data stream processing solution which renders less operational cost compared to a fixed over-provisioning approach. While Ishii and Suzumura rely on a hybrid cloud model, there are also several cloud-based SPEs that are designed to run on a single homogenous cloud. The most prominent representatives are Apache Storm [17], Apache Spark [18], Apache Samza [16], Apache Flink [64], and Heron [14].

Table 3.2: Elastic Stream Processing Engines

	VISP Runtime	Borealis	IBM System S	StreamCloud	Spring Cloud Data Flow	Apache Spark	Apache Storm	Distributed Apache Storm	Google Cloud Dataflow	Amazon Web Services IoT
Distributed Runtime	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Distributed Clouds	✓				✓			✓	(✓)	(✓)
Resource Elasticity	✓		(✓)	✓	✓			✓	✓	✓
Cost Efficiency	✓							✓	(✓)	(✓)

These SPEs are motivated by the Big Data domain and are designed to efficiently process huge amounts of data in parallel without any delay. Nevertheless, these SPEs hardly support any topology changes and each modification of a topology or the resource configuration for the SPA triggers a complete redeployment of to the SPA which is not feasible for IoT-based environments as discussed in Section 2.3. To resolve this challenge, Cardellini et al. [65] propose an extension to Apache Storm. This extension implements an adaptive scheduler that can reorganize the topology deployment of the SPA at run time to allow for an elastic resource provisioning strategy for each operator individually.

Besides this academic approach, there are also other projects like Spring Cloud Data Flow [66], which also picks up the concept of elastic resources for each operator to enable fine granular resource provisioning strategies. The downside of these SPEs is that they only support stateless SPAs [67]. In order to also support stateful stream processing operators, it is required to implement synchronization strategies among all replicas of each operator. To solve this problem, some preliminary work has already been presented: Fernandez et al. [68] propose to implement a checkpointing mechanism whose snapshots are distributed among all replicas of the same operator to synchronize the state. Gedik et al. [69] propose the usage of a shared storage, e.g., a key-value store, to synchronize the state among multiple operators. The major advantage of this external shared storage is that it can be integrated into any SPE without changing the fundamental system design of the SPE in contrast to the checkpointing mechanism proposed by Fernandez et al. In addition to the fundamental challenges of designing an elastic SPE that supports stateful SPAs, there are also several challenges on how to implement cost-efficient resource provisioning strategies for SPAs [26]. These challenges have received a lot of attention in recent years and Section 3.3 provides a detailed discussion on this topic.

When we compare the above mentioned SPEs based on Table 3.2, we can see that all SPEs already support a distributed runtime. This distributed runtime can be either on a fixed resource cluster, like for Borealis, on cloud resources, like the stream processing infrastructures by Google and Amazon, or can be completely dynamic, like the distributed Apache Storm extension proposed by Cardellini et al. [65].

Some of these SPEs are also capable of running on more than one cloud across different geographic locations. This feature enables the SPEs to operate on hybrid clouds, where the majority of the data is processed within a private cloud but it is also possible to cover peak loads with computational resources from external public clouds. For the two major cloud service providers Google and Amazon, there are no concrete publications on how they handle the resource provisioning across different clouds, but there is evidence that they also integrate different computational resource pools to provide their services to users [70]. Furthermore, there are also several SPEs which can update their resource provisioning configuration at run time like Spring Cloud Dataflow or the services by Amazon and Google. Nevertheless, they hardly provide any optimized resource provisioning algorithms to minimize the operations cost for SPAs. However, there are already several proposals for cost-efficient resource provisioning approaches on top of the SPEs listed in Table 3.2 as presented in the next section.

3.3 Resource Provisioning Approaches for Stream Processing Applications

Alongside the emergence of the cloud computing paradigm [30], several research groups have started to propose different resource provisioning approaches to leverage the resource elasticity of the cloud. In the area of data stream processing, most early publications focus on an optimal resource configuration only when deploying a topology and do not consider any updates at run time, e.g., Setty et al. [71] for pub/sub systems or Florescu et al. [72] for database systems. The next step towards resource elasticity was proposed by Lim et al. [73], who proposed to redeploy complete SPAs whenever the data volume, i.e., resource requirements, change. Although this approach already supports resource elasticity for SPAs, it is required to refine this approach to only consider individual operators instead of the complete SPA. This individual handling of operators is addressed by Schneider et al. [74], who propose the individual parallelization of operators within IBM System S. Because this first approach only considers stateless operators, the authors complements their approach in a succeeding publication to also consider the replication of stateful operators [69].

Besides the elasticity extension to IBM System S, there are also several 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. [75] and Xu et al. [76] who introduce threshold-based custom schedulers that can update the topology deployment at run time, depending on the incoming data volume. Although any replication of a specific operator provides additional processing capabilities, it has to be noted that any reconfiguration of the topology deployment has a negative impact on the processing performance. To minimize these reconfigurations, Stela [77] introduces new performance indicators to focus on the actual throughput of the SPA and to eliminate all unnecessary reconfigurations.

To extend the rather static aspect of the threshold-based scaling approaches [75], [76], Heinze et al. [78] propose a threshold-based resource optimization whose thresholds are updated on a regular basis according to an online learning mechanism. This allows the resource optimization approach to refine the otherwise fixed thresholds to improve the resource usage based on actual monitoring data. The SEEP SPE [68], also proposes a threshold-based replication mechanism but in contrast to the already discussed approaches, SEEP focuses on stateful operators and employs a dedicated fault tolerance mechanism.

In addition to the threshold-based replication approaches, there are also some works that optimize specific aspects for operating SPAs. One of these aspects is the partitioning of data to optimize the data flow among the operators, especially for stateful operators. The Streamcloud [79] SPE proposes a mechanism to partition the incoming data to distribute it efficiently among the different replicas of one operator type. De Matteis and Mencagli [80] present a predictive approach to minimize the latency of the SPAs and improve the energy efficiency of the SPEs. This approach allows to reduce the reconfigurations of the topology to reduce the overall management effort for the SPA.

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. [81] consider different types of Virtual Machines (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 instead of the CPU.

3.4 Topology Definition Approaches

The large number of established SPEs results in a large variety of different topology definition approaches that require SPAs to be designed for each SPE individually. To address this incompatibility issue, the Apache Beam project has been initiated [82]. This project introduces the abstract Beam Model based on the Dataflow model [83] that can be directly translated into concrete instructions for different SPEs, like Apache Spark [18] or Apache Flink [64]. However, Apache Beam only supports a small set of features for SPAs because it is an abstraction layer for other topology definition approaches and therefore can only support the least common denominator among these SPEs.

Besides Apache Beam, there are also several native topology definition approaches for SPEs as shown in Table 3.3. This table lists several topology definition approaches besides the VTDL, which is introduced in Chapter 6 and analyzes them whether they support any advanced features for SPAs besides composing the individual operators to form an SPA, like deployment preferences or QoS-related aspects.

One of the most important topology definition approaches is the SPL, which has been proposed by Hirzel et al. [84] for IBM System S. The SPL has already been proposed several years ago and only considered the composition of operators.

Table 3.3: Topology Definition Approaches

	VTDL	Dataflow, Apache Beam	SPL	CQL	FLUX, Apache Storm	Apache Spark Streaming
Deployment Preferences	✓		(✓)			
QoS Aspects	✓				(✓)	(✓)
Fault Tolerance	✓					(✓)
Semantic Annotation	✓		✓			
Runtime Modification	✓				(✓)	(✓)
Data Transfer Aspects	✓					

In recent years, the SPL has been revised and now also supports the placement of operators on specific computational resources within one IBM System S instance [28]. However, the SPL does neither support topologies across different IBM System S instances nor considers any QoS-related or fault tolerance-related instructions. Another approach for defining the topology of SPAs, the Continuous Query Language (CQL), has been presented by Arasu et al. [85]. The CQL follows the design principles of the Structured Query Language (SQL), which allows users to create new SPAs based on a SQL-like syntax without considering any SPE-related aspects. The downside of this approach is that the CQL only focuses on filtering data streams and therefore does not support any complex features for SPAs. Furthermore, the CQL is not widely supported by SPEs. To address this issue, Soule et al. [86] proposed an intermediate language called River, which allows running CQL-based SPAs on IBM System S. This abstraction layer follows a similar approach as the Apache Beam project and also supports the integration of StreamIt [87], a programmatic topology definition approach for IBM System S.

Although abstract topology definition approaches like SPL or CQL have been around for several years, the majority of today's established SPEs only supports a code-based topology definition approach. This is mainly due to the fact that code-based approaches can be directly run on SPEs while abstract approaches need an additional layer that translates the topology definition into concrete composition and wireing instructions. For the analysis of the related work, we focus on Apache Storm [17] and Apache Spark Streaming [18] as representatives for established SPEs. Nevertheless their feature set is similar for other SPEs like Apache Apex [88], Apache Flink [64], and to some extent also Apache Kafka [89]. These established SPEs already consider basic QoS-related aspects and can redistribute computational resources at run time, but none of them supports any structural changes to topologies, like the replacement of a specific operator, at run time. Within the Apache Storm ecosystem, there is also the Flux project [90], which provides an abstraction layer for SPAs for Apache Storm. This abstraction layer follows similar design principles as the SPL and allows users to create SPAs with hardly any knowledge on data stream processing.

CHAPTER 4

Motivating Scenario

In this chapter, we present a motivating scenario from the manufacturing domain and provide high-level requirements that guide through the remainder of this thesis.

4.1 Monitoring of Manufacturing Machines

One of the most prominent scenarios for the rise of the IoT is the manufacturing domain, where more and more manufacturing machines are equipped with sensors that generate data streams. These data streams can be used to extract vital information about the health or the production progress of the machines. For our motivational scenario, we consider a monitoring SPA, which transforms raw sensor data into an human-readable report. The task and structure of this SPA is based on a real world scenario, which was analyzed in the course of the EU H2020 project on Cloud-based Rapid Elastic Manufacturing (CREMA) [91]. First, we present the topology of this monitoring SPA as well as the tasks of the individual operators and second, we discuss the deployment for this SPA.

4.1.1 Design of the Monitoring Stream Processing Application

Figure 4.1 provides an overview of the topology of the monitoring SPA on the right hand side, which is composed of nine different stream processing operator types (O1 – O9) to process the data originating from three different data sources (S1, S2, and S3). Each of the operator types performs a dedicated operation to transform the raw data from manufacturing machines step-by-step into value-added and human-readable information. The information provided by the data sources is used to monitor three different aspects: the availability of the manufacturing machines, the temperature to avoid overheating and the Overall Equipment Effectiveness (OEE), a commonly used Key Performance Indicator (KPI) to evaluate the efficiency of manufacturing machines [92].

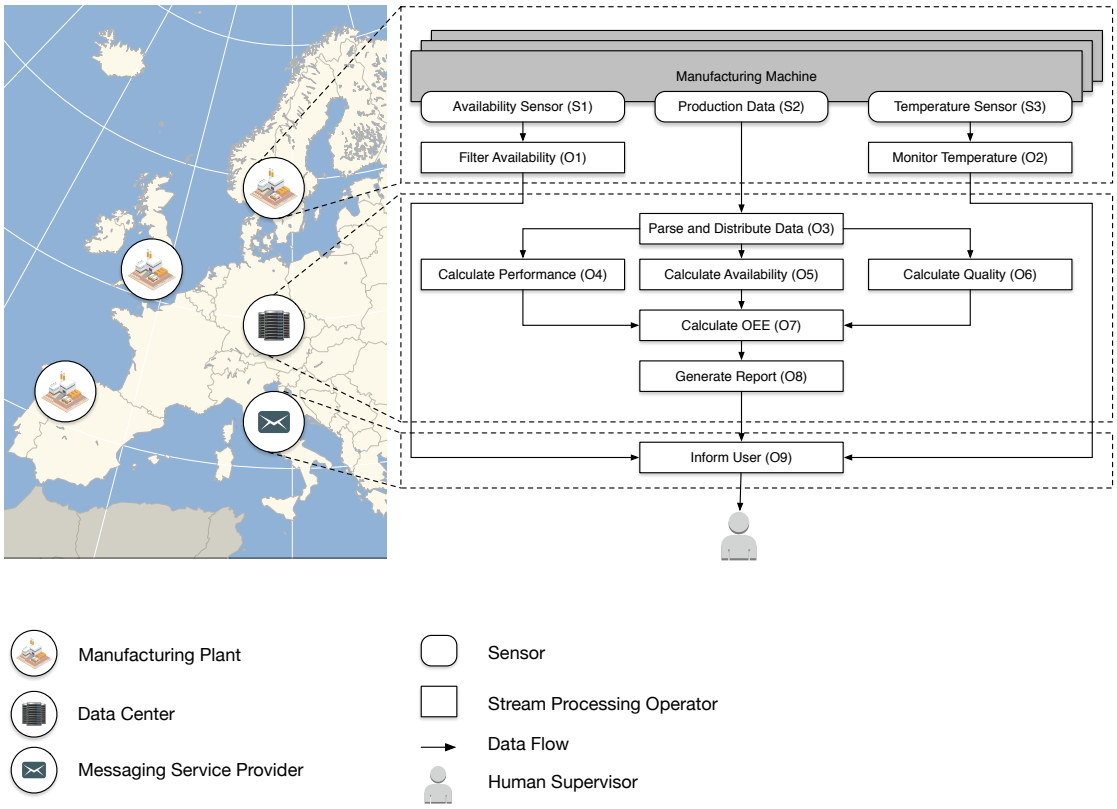


Figure 4.1: Motivating Stream Processing Application

For this motivational scenario, there are two different types of data sources. The first type of data sources are sensors, i.e., S1 and S3, which emit machine-readable data and can be directly accessed via an Application Programming Interface (API). The second data source type, i.e., S2, is a video feed, which scans a display of the manufacturing machines because the production information is not directly accessible via an API. To extract the information from this video stream, it is required to apply 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 filtered by the *Filter Availability* (O1) operator, which generates warnings for each new defect incident of a specific manufacturing machine. The warning is then forwarded to the *Inform User* (O9) operator that notifies the human supervisor. All other status reports are discarded, since they only signal intended behavior which does not require any action by the supervisor.

The second data type 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* (O3) operator parses the image and distributes the information to three operators (O4, O5, and O6) that calculate the different components of the OEE value. These individual components are then united by the *Calculate OEE* (O7) operator and the calculated OEE value is forwarded to the *Generate Report* (O8) operator, which generates a report every minute. This report aggregates the information of all monitored machines and is forwarded to the *Inform User* (O9) operator where it can be accessed by the human supervisor.

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 to avoid overheating. This warning is also forwarded for each new incident to the *Inform User* (O9) operator to inform the human supervisor about potentially problematic situations of the manufacturing machines.

4.1.2 Deployment of the Monitoring Stream Processing Application

For the deployment of the monitoring SPA, we consider two different scenarios. The first scenario is a centralized one, which represents the state-of-the art deployment for SPAs running on SPEs. For this scenario, all sensors and operators are located in one geographic location on a common set of computational resources. Although this scenario is straight forward in terms of handling the data flow and assigning computational resources, it does not consider the requirements for organizations operating plants in different locations. Therefore, we also consider a distributed execution environment across three different geographic locations as shown on the left hand side in Figure 4.1 The first geographic location is the manufacturing plant, which is located in Sweden. This manufacturing plant hosts the manufacturing machines as well as the sensors that generate the data streams. Two of these data streams are already filtered by operators O1 and O2 that are running in a private cloud on the premises of the manufacturing plant. The data originating from the third data source (S2) is directly forwarded to a public cloud in Germany that provides massive computational resources. The operators in this public cloud are in charge of transforming the image data into machine-readable text as well as calculating the OEE of the manufacturing machines. The individual OEE values are then aggregated in a report which is sent to an operator maintained by an external service provider who is in charge of forwarding the information to the human supervisor.

4.2 Identified Requirements

In order to create a holistic data stream processing ecosystem, which is capable of addressing the challenges originating from the rise of the IoT, we identify four high-level requirements that will serve as guidelines throughout this thesis:

1. We need to design an SPE that is capable of processing data streams originating from different geographic locations in a network-efficient manner.
2. We need to design a model for topologies, which allows to describe deployment constraints for SPAs running in distributed environments.
3. We need to identify suitable SLOs to assess the performance of SPAs and design suitable mechanisms to model SLAs for topologies.
4. We need to design suitable resource provisioning and scaling approaches to minimize the operational cost for SPAs both in centralized and distributed deployments.

CHAPTER 5

A System Design for Stream Processing Ecosystems

In this chapter, we analyze the previously introduced motivational scenario and present the requirements for a stream processing ecosystem. Based on these requirements we then propose the system design for the VISP Ecosystem whose core components are the VISP Runtime and the VISP Marketplace. To evaluate the feasibility of the VISP Ecosystem, we also provide a use case evaluation for designing and operating SPAs. The main focus of this use case evaluation is to analyze the efforts which are required for an user to design and operate an SPA in a distributed environment.

5.1 Overview

In recent years, not only the number of IoT devices, but also their application areas increased dramatically. While the first IoT devices were only considered as technological proofs of concept, they are constantly evolving to become suitable for day-to-day application scenarios. Up to now, most IoT devices are only used in point-to-point scenarios, where the sensor data of IoT devices is only processed by one software service, e.g., adaptive lighting based on the current content on a TV screen. Nevertheless, it is necessary to take the communication and data processing capabilities to the next level by integrating heterogeneous IoT devices with multiple software services [93]. Therefore it is required to design appropriate ecosystems that allow for easy integration of IoT devices with software services in order to realize SPAs. Currently, there are already several IoT platforms available that support the integration of IoT devices. Nevertheless, most of these platforms have deficiencies such as missing support for heterogeneous IoT devices and open challenges concerning the privacy of the processed data [41].

In addition, state-of-the-art IoT platforms do not sufficiently support complex SPAs and do not provide means for reusing stream processing operators. Traditional providers of most IoT platforms are IoT device manufacturers with the primary goal of processing sensor data of their IoT devices. Therefore, they do not support any complex SPAs that allow integrating heterogeneous IoT devices. When it comes to creating complex SPAs, these IoT platforms are only suitable for preprocessing data, i.e., transforming raw sensor data into a machine-readable format. For all other processing steps, it is inevitable to fall back to established SPEs like Apache Storm [17] or Apache Spark [18]. Although these established SPEs excel at processing streaming data, they only provide very basic mechanism in terms of resource elasticity or reconfiguration at run time.

Since complex SPAs often integrate IoT devices that are situated at different geographic locations, it is in many cases necessary to already perform operations, such as data filtering, close to the data source. This distributed deployment reduces the amount of data that has to be transferred among the different geographic locations and supports real-time data processing [94]. Nevertheless, an overlay, i.e., an user interface, which handles the complexity of the distributed deployment transparently for an SPA user is vital [95]. It is also essential to provide a graphical user interface that is easy to use for visualizing and designing topologies that represent the structure of SPAs [96].

Furthermore, it is required to devise strategies to reuse already existing building blocks, i.e., stream processing operators, for future topologies to reduce the workload of creating SPAs. To address the deficiencies of state-of-the-art IoT platforms, especially in terms of data stream processing, we propose the VISP Ecosystem. In this chapter, we present the system system design of the VISP Ecosystem and discuss its capabilities by evaluating a real-world industry scenario.

The remainder of this chapter is structured as follows: Based on the motivational scenario presented in Chapter 4, we discuss the challenges for realizing SPAs in Section 5.2. In Section 5.3 we present the system design and implementation details of the VISP Ecosystem. We evaluate the feasibility of the VISP Ecosystem based on a use case evaluation in Section 5.4, discuss the evaluation regarding the identified challenges in Section 5.5, and summarize the chapter in Section 5.6.

5.2 Challenges

In addition to the challenges presented by Mineraud et al. [41], e.g., the support of heterogeneous devices, data ownership or data fusion, we have identified three further research challenges for designing and operating SPAs, like the one described in the motivational scenario.

5.2.1 Distributed Deployment

Since some operators require low latency, they need to be deployed close to the data source, e.g., on edge resources [94]. Although edge resources allow for a better response time due to the lack of the data transfer over the Internet, edge-based computational resources are more expensive than typical cloud-based resources, due to the economy of scale [97], [98]. Therefore, it is required that the VISP Ecosystem supports distributed deployment and migration strategies, to migrate operators among different deployment locations [79].

5.2.2 Ease of Designing Stream Processing Applications

In order to establish IoT ecosystems, i.e., holistic toolchains that enable users to easily integrate IoT devices and extract information of the data streams provided by these IoT devices, it is essential to minimize the entry barriers for new users. Such entry barriers can be found when designing, deploying, and operating SPAs. At design time, it is required to provide an easy to use toolkit, which enables domain experts to design SPAs. The same also applies at deploy and run time. Here it is required that the runtime environment covers all aspects from obtaining external dependencies, i.e., the concrete implementation for the operators, for ensuring the compliance with predefined SLOs by obtaining enough computational resources for the operation. Therefore, it is also required to design mechanisms, which enable the reconfiguration of a SPA at run time to eliminate downtimes when the topology or the deployment of an SPA needs to be updated.

5.2.3 Reuse of Operators

To minimize the required effort for implementing SPAs, it is desirable to reuse already existing components similarly to the library usage for generic software development [99]. The data stream processing domain is predestined to decompose topologies into single operators and use them as building blocks. These building blocks serve as the foundation for a data stream processing ecosystem for the IoT, where all participants of the ecosystem can share their operators. Based on the motivational scenario, we have identified three different data stream processing operator categories with different potentials for reuse.

Pre-built operators Operators of this category may be already implemented and available to be integrated into SPAs. These operators may be either free of charge, e.g., transformation operators, which are provided by the producers of the IoT devices, or generic operators. Generic operators, like filters, may be provided by third party developers and may be obtained by paying a one-time fee to the operator developer. Pre-built operators have the same characteristics as software libraries since they are integrated into the SPA and the user has full control over these operators including their SLA compliance that can be controlled by provisioning enough computational resources.

External operators The second type of operators is represented by external services that can only be integrated on a SaaS basis. For this kind of operators, the user cannot control any aspected related to the QoS. Furthermore, these operators typically imply a pay-per-use policy, where the provider of the operator charges the user for each data item processed. The main reason for providing operators in a SaaS manner is that the owner may not able to give away the code or even the binary of these operators either due to business-related reasons or due to legal aspects. Typical examples for such operators are analysis algorithms, e.g., the interpretation of OEE values, where the owner does not want to reveal the intellectual property of the algorithm.

Domain specific operators The last type of operators represents operators with unique business logic for specific SPAs, like the *Generate Report* operator in our motivational scenario. Due to their uniqueness, they are not available upfront and need to be implemented by the designer of the SPA.

5.3 System Design

In order to address the challenges as discussed in Section 5.2, we propose the VISP Ecosystem. This ecosystem consists of two major components: the VISP Marketplace and the VISP Runtime, as depicted in Figure 5.1. The VISP Marketplace aims at creating a repository of operators for SPAs to minimize the required effort for the user to design a SPA. The VISP Runtime complements the VISP Marketplace, by providing a runtime environment to execute the previously designed SPAs. In the remainder of this section, we are going to present the system design as well as its underlying design rationales.

5.3.1 VISP Marketplace

The VISP Marketplace is the first place to go for users who want to design SPAs. It provides four core functionalities: the Operator Registry, the Operator Distribution, the Topology Builder and the Monetization component.

The *Operator Registry* provides the graphical user interface, where any user who wants to create an SPA can browse for data stream processing operators. User can also submit custom operators to the VISP Marketplace to either integrate them into their SPAs or to be integrated by other SPAs. When submitting a new operator to the Operator Registry, it is required to provide an *Operator Image* (see Section 5.3.2), which can be instantiated by the VISP Runtime. The user also needs to provide a semantic annotation to define all possible input as well as output data types for this operator and to provide details on its monetization model. This Operator Image and the respective metadata are then stored in the *Operator Registry*, where it can be accessed by the VISP Runtime by means of the *Operator Distribution* component.

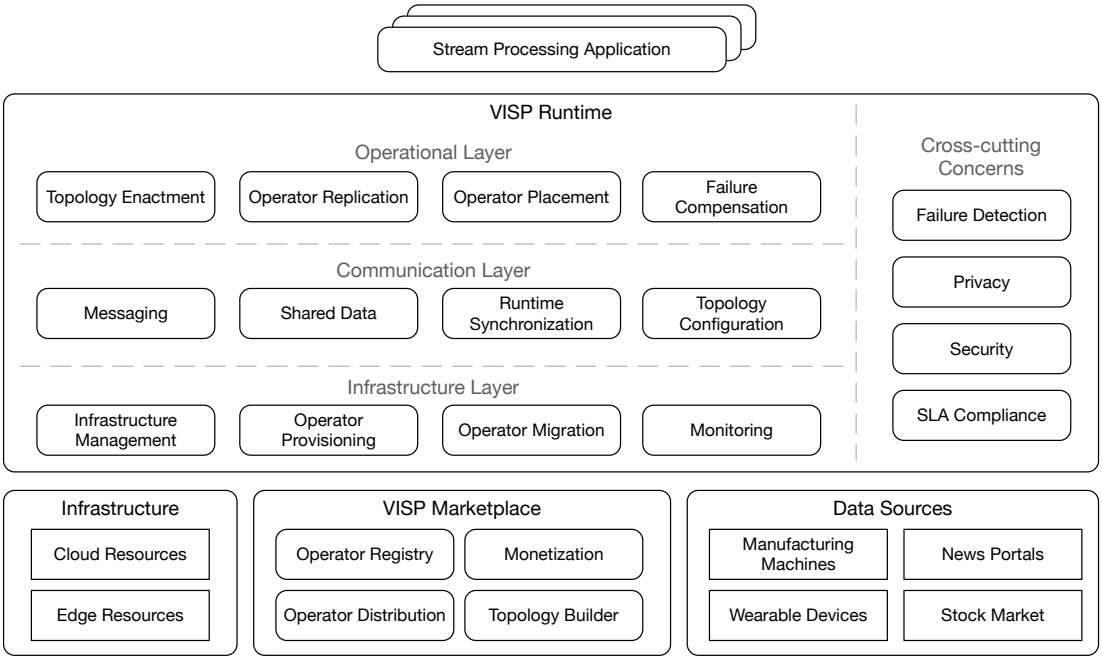


Figure 5.1: VISP Ecosystem

Based on the Operator Registry, the VISP Marketplace also offers a *Topology Builder*, which represents a graphical toolkit for designing SPAs. The Topology Builder provides a graphical user interface that allows users to create topologies by dragging operators from the Operator Registry to a digital drawing board, where they can wire the operators to realize the planned SPAs. The Operator Registry offers custom and external operators, as well as data sources and data sinks. Each operator has different characteristics, like QoS metrics or deployment restrictions, which are discussed in detail in Section 6.4. These characteristics can be defined for each operator based on a context menu in the Topology Builder.

Besides the basic design functionality, the Topology Builder also provides a context-aware search, where the user can search the Operator Registry based on the semantic annotations for a suitable successor for the currently selected operator on the digital drawing board. As soon as the topology is designed, the user can then export the topology based on the VTDL (see Chapter 6) to be deployed on the VISP Runtime. The goal of the Topology Builder is to allow domain experts to create SPAs for different use case scenarios.

The last component of the VISP Marketplace is the *Monetization* component. This Monetization component takes care of billing aspects for operators. Currently three different business models are supported: For the first business model, the Monetization component charges the user once for the usage of an operator and the user can then use the operator for an unlimited timespan and unlimited amount of operations.

This model can only be applied to pre-built operators. The second business model implements a time restricted model, where the user can lease an operator for a specific timespan. As soon as this time span is over, the user is required to lease the operator again or the data processing operation cannot be performed anymore. The last business model implements a pay-per-use model, where the user has to pay for each operation carried out by an operator. While the first model allows for an SPA deployment on a secluded location without any connection to the Internet, the other two models require a regular connection to the VISP Marketplace to either check the validity of the rented operators as well as to report the performed operations for billing purposes. These three models represent the most common business models for an IoT-based data stream processing topology, but we expect more elaborate business models in the future [100].

5.3.2 Operator Images

Operator Images represent building blocks, i.e., concrete implementations of operators, for stream processing topologies. These Operator Images are instantiated by the VISP Runtime to create *Operator Instances* that process data. Since Operator Images can be contributed by individual developers, we identified several requirements which need to be met to use operators within the VISP Runtime.

First of all, we do not restrict the Operator Image implementation to any programming language, as long as the Operator Image can be packaged within a container, i.e., a Docker Container [101]. This common format is required to enable the deployment for the VISP Runtime, since a Docker Container already provides a pre-configured execution environment.

In addition, the Operator Image needs to implement several functionalities. The most important functionality is the *Processing Logic*. The Processing Logic represents the functionality of the stream processing operator. This functionality ranges from simple filters over SQL-like aggregations to complex business logic [10]. Some operators can also include complex software systems, such as Business Process Management Systems (BPMS) or machine learning-based decision systems, to conduct the desired operations. Besides the actual processing, the Operator Image also needs to implement a functionality to subscribe to data sources, i.e., preceding operators, as well as to publish the processed functionality.

In order to ease the implementation for operator developers, VISP already provides several APIs to access the *Shared Data* for synchronization among Operator Instances as well as the *Messaging* infrastructure, which are both part of the VISP Runtime. Nevertheless, it is also possible to access the Messaging infrastructure directly, either by using the AMQP [102] or the MQTT [103] protocol. For the Shared Data, it is required to either use the APIs or send queries directly for the Redis data structure store [104].

To ensure an SLA-compliant execution of the data processing, it is required to obtain metrics from each running Operator Instance, like CPU or memory usage as well as network I/O.

These metrics are gathered within the *Monitoring* component of the VISP Runtime in a configurable interval by means of the Messaging infrastructure. VISP already provides a monitoring framework that can be integrated into the Operator Images without any further implementation required. Nevertheless, it is also possible to implement a custom monitoring infrastructure as long as it implements the same API as the provided one.

Besides the QoS monitoring, it is also required to establish a failure propagation mechanism. Since Operator Instances are running autonomously on computational resources, it is necessary to aggregate failure reports at a central location. Therefore, each Operator needs to implement a failure reporting component that collects all exceptions and forwards them to the Monitoring component of the VISP Runtime by means of the Messaging infrastructure.

In order to configure an Operator Instance at startup, they also have to implement a *Configuration API*. This Configuration API allows the *Operator Provisioning* component to configure the location of the Shared Data as well as the information about the data origin and the destination of the processed information in the Messaging infrastructure. In addition, the Configuration API may also be used to configure further functionalities of the Operator Instances, e.g., by defining thresholds or filter criteria.

5.3.3 VISP Runtime

The VISP Runtime enables the user to operate SPAs and has been designed, to be either deployed on a public cloud, e.g., on Amazon EC2 [105], or on computational resources within the user's premises on a OpenStack-based private cloud [106]. Therefore, the VISP Runtime is provided as a pre-built package, i.e., a VM image, which can be deployed on computational resources. After providing the credentials for the VISP Marketplace and computational resources for the operators, the VISP Runtime is ready to be used. Nevertheless, it is also possible to deploy single components of the VISP Runtime on different hosts if required.

Furthermore, it is also possible to deploy several instances of the VISP Runtime at different geographic locations, e.g., on edge resources, and use them to realize a distributed execution environment for SPAs. This is often necessary to reduce the latency between the data provider and the data processing operators. The deployment as well as the communication among the individual VISP Runtimes does not require any interaction or configuration by the user, since the distributed deployment is already considered in the fundamental design of the VISP Runtime. The communication among these individual VISP Runtimes is conducted by the Runtime Synchronization component. Besides the overall replication mechanism of the VISP Runtime, it is also possible to replicate individual components on clusters, e.g., the Messaging infrastructure or the Shared Data, within one instance of the VISP Runtime to cope with high system load. The VISP Runtime considers four aspects: Configuration, Operator Instance Management, Elasticity, and Communication, which are discussed in the remainder of this section.

Configuration

The *Topology Configuration* component is responsible for importing the VTDL-based topology description from the Topology Builder and for propagating the configuration to other VISP Runtime instances by means of the Runtime Synchronization component. The synchronization procedure is discussed in detail in Section 6.5. After the synchronization, the VISP Runtime configures the Messaging infrastructure and fetches the Operator Images from the VISP Marketplace, as described in Section 5.3.4.

In order to ease the communication between the Topology Builder and the VISP Runtime, we designed the VTDL. This description language builds on the concepts of the SPL description language [84] and is presented in detail in Chapter 6.

Operator Instance Management

The Operator Provisioning component takes care of deploying Operator Instances on computational resources, monitoring their availability, and aggregating runtime exceptions. Since the VISP Runtime deploys Operator Instances on computational resources, it is required to obtain sufficient cloud resources, which is carried out by the *Infrastructure Management* component. This component is able to lease computational resources from public clouds, e.g., Amazon EC2 [105], or private clouds, e.g., OpenStack [106], and provides a suitable runtime environment for the Operator Instances. As soon as the computational resources are obtained, the Operator Provisioning component is able to deploy respectively un-deploy Operator Instances on these computational resources, whereas the deployment decisions are generated by the *Operator Replication* or *Operator Placement* components. For each operator, there is, at least, one, but up to arbitrary many, Operator Instances running at the same time. However, there may also be redundant Operator Instances for a specific operator to cope with high system loads.

Besides the resource provisioning and deployment mechanism (see Section 6.4), the Operator Instance Management also takes care of monitoring the availability of all Operator Instances and aggregating failure reports. The *Monitoring* component constantly checks if all Operator Instances are available and capable of processing data. Whenever a downtime or failure of an Operator Instance is recorded, this component triggers a new instantiation for the affected operator or applies more complex failure compensation mechanism based on the reasoning of the *Failure Compensation Component*. In addition, the Monitoring component also aggregates all exceptions reported by the Operator Instances, and informs the SPA user about these failures. Optionally, it is also possible to inform the developer of the operator by generating an error report, including the input data, information about the current state in the Shared Data, and a complete failure log output.

Elasticity

The Elasticity aspect of the VISP Runtime takes care of the SLA compliance for the SPA operation. Therefore, the *Monitoring* component records three performance indicators. The first performance indicator is based on the individual Operator Instances, which continuously report their performance metrics, i.e., CPU and memory usage. On the one hand, a high system usage is an indicator that the current processing capabilities are not sufficient and it is required to add additional Operator Instances to cope with the system load. On the other hand, a low system usage can indicate the possibility to remove one replicated Operator Instance for a specific operator to reduce the overall cost of computational resources.

The second indicator is the system load on the Messaging infrastructure. Since the Messaging infrastructure not only connects the individual operators, but also acts as a short term buffer, the size of all currently buffered messages can suggest either upscaling or downscaling actions. The third indicator is observed by an introspection of the individual messages on the Messaging infrastructure. Here the Monitoring component selects individual messages in a configurable interval and identifies the processing duration of these messages based on the processing timestamp within the message. Whenever a message waits longer to be processed than defined by the operator, it may be required to add additional Operator Instances for this specific operator. Since the Monitoring component only observes the performance indicators of the VISP Runtime and the individual Operator Instances, the *Operator Replication* component analyzes the performance indicators and decides then whether to scale up or down based on the SLOs provided for each operator. A more detailed discussion on different scaling approaches can be found in Chapter 7 and Chapter 8.

Communication

Within VISP, there are three different communication layers: first, the communication within the SPA, second, the synchronization among multiple Operator Instances for one operator and third the communication among different VISP Runtimes. The Messaging infrastructure provides a message broker for realizing the communication within the SPA. While most established SPEs rely on a tight-coupled communication among the different operators, the Messaging infrastructure for VISP allows for loose coupling, which can be reconfigured at run time. These reconfiguration capabilities eliminate downtimes or SPA redeployments, whenever an updated SPA is deployed, which is not possible for established SPEs as discussed in Section 3.2. Another advantage of this loose coupling strategy is the possibility to deploy a single SPA across different VISP Runtime instances. For the communication within the Messaging infrastructure, we require each message to contain a header to define the target operator of the message and a timestamp, stating when the last processing activity was performed, alongside any arbitrary payload in the message body. Finally, the Messaging infrastructure also supports a short time buffer for messages that is required to compensate short communication outtakes or store the messages until enough processing capabilities, i.e., Operator Instances, are available.

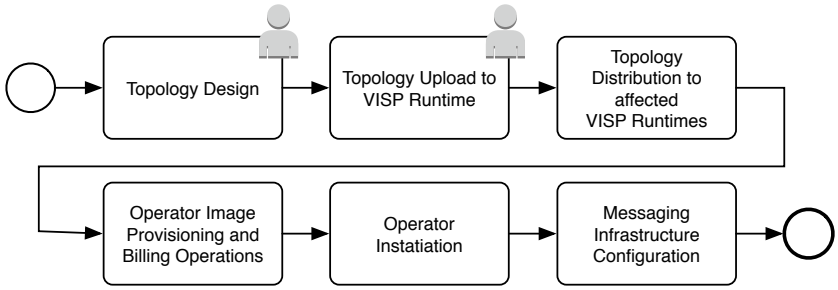


Figure 5.2: Deployment Process for a Stream Processing Application

For the synchronization among multiple Operator Instances for one operator, VISP relies on a shared key-value storage system, as suggested by Gedik et al. [69]. This eases the operator instance management of the topology, because this mechanism allows to easily add and remove Operator Instances at run time to adapt to the system load. Finally, each VISP Runtime implements a dedicated REST-based API, which allows the communication among different VISP Runtimes, e.g., to forward the topologies of distributed SPAs.

5.3.4 Topology Creation

Each creation of an SPA follows a specific workflow as depicted in Figure 5.2. At the beginning, the SPA user, i.e., a domain expert, designs the topology of the SPA by means of the Topology Builder located in the VISP Marketplace. This topology design only requires domain knowledge for the SPA and can be carried out by dragging suitable operators to a digital drawing board and wiring them. When the topology design is finished, the user exports the topology by storing the topology in a file, based on the VTDL (see Chapter 6).

The user then uploads the file to an arbitrary VISP Runtime, which is also the last task where the SPA user is involved. All other tasks are conducted automatically by the VISP Runtimes. The uploaded file is analyzed by the Topology Configuration component to identify changes compared to previous topologies and whether it is required to deploy any Operator Images on other VISP Runtime instances. After the analysis phase is finished, the Runtime Synchronization component forwards the deployment decision to all affected VISP Runtime instances. Further details on this analysis and topology distribution phase are discussed in detail in Section 6.5. Each affected VISP Runtime then obtains the required Operator Images from the Operator Registry and performs optional billing operations for the operator by using the Monetization component of the VISP Marketplace. As soon as all Operator Images are obtained, one Operator Instance is deployed for each operator on computational resources which are provided by the Infrastructure Management component. Finally, the Topology Configuration component triggers all required modifications to the Messaging infrastructure and the SPA is ready to process the data streams.

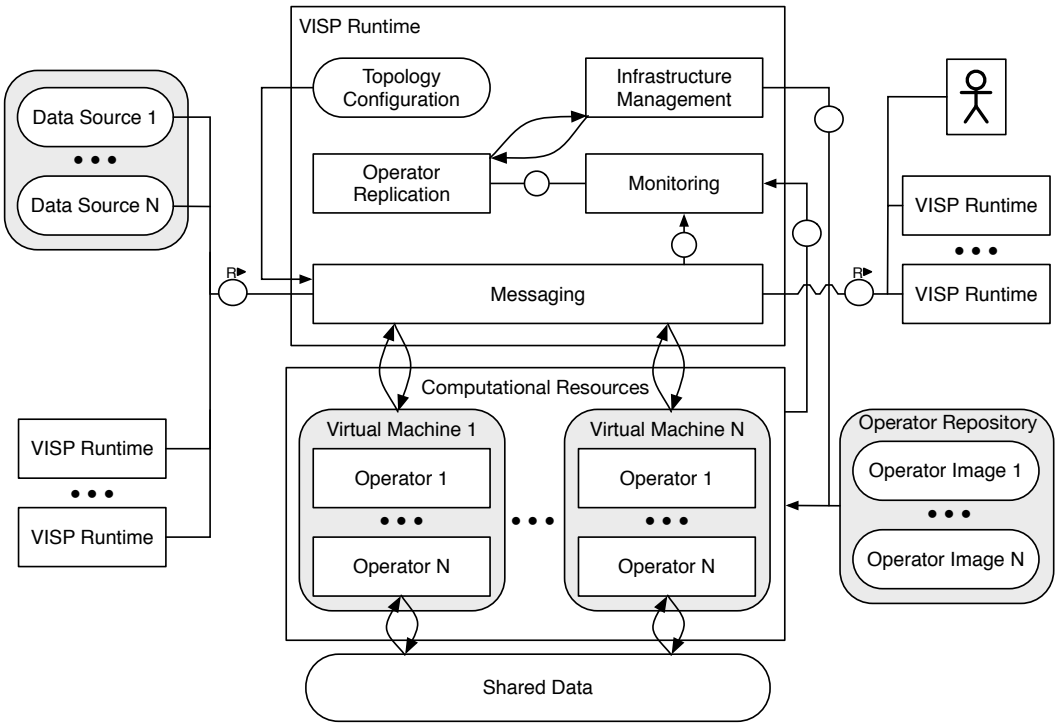


Figure 5.3: System Design of the VISP Runtime

5.3.5 Deployment of a VISP Runtime on Cloud Resources

Although it is possible to operate a VISP Runtime and its associated Operator Instances on fixed computational resources, a cloud-based environment is the natural choice for the VISP Ecosystem to leverage the necessary resource elasticity for volatile loads. Figure 5.3 shows an exemplary deployment of a VISP Runtime using the FMC notation [107]. The figure shows the deployment of one exemplary VISP Runtime in the center, which is connected to arbitrary many other VISP Runtimes, data sources or human actors, who receive the processed data. Besides these interactions, each VISP Runtime also depends on the computational resources for running Operator Instances and the Operator Repository to obtain Operator Images.

When an SPA is deployed within the VISP Ecosystem on computational resources, it processes data emitted by data sources (on the left-hand side of Figure 5.3), to extract information for users, as shown on the right-hand side of Figure 5.3. The SPA can be deployed across multiple VISP Runtimes, which are self-contained regarding managing the data flow, resource provisioning and data processing. This allows the integration of VISP Runtimes as data sources as well as data sinks to realize a distributed execution environment for SPAs. The data to be processed is provided by data sources or preceding VISP Runtimes that push the data to the Messaging infrastructure of the VISP Runtime shown in the center of Figure 5.3.

The actual data processing is conducted by operators which are running on computational resources, i.e., VMs, which can be hosted both on private or public clouds. Whenever an SPA is deployed, each required operator is obtained from the Operator Repository which is hosted in a centralized location that has to be accessible for all VISP Runtimes. As soon as all Operator Instances are started, they fetch the data from the Messaging infrastructure, process it and return the results to the Messaging infrastructure. These results are then either fetched from succeeding operators within the same VISP Runtime or forwarded to other VISP Runtimes in a different geographic location. The remaining components of the VISP Runtime are in charge of provisioning enough computational resources, monitoring the state of the Operator Instances or replicating operators to deal with a high data volume.

5.3.6 VISP Implementation

The implementation of the VISP Ecosystem is built on top of established technologies and libraries in the domain of cloud computing and an established container technology stack. The source code is available on Github under the Apache 2.0 license [108]. For the implementation of the different components of the VISP Ecosystem, we rely on the Spring Cloud software stack [109]. This software stack provides several libraries to implement distributed systems which require a reliable communication as well as failure detection and failure mitigation mechanism. The Messaging infrastructure for the VISP Runtime, is provided by RabbitMQ [110] to realize a reliable communication and Redis [104] for an efficient Shared Data storage. We selected these two established software solutions due to their efficient data processing design and the possibility to create clusters to deal with high loads.

To lower the entry barrier for third-party developers who contribute custom operators, we decided to rely on the Docker tool stack for packaging Operator Images as well as for the backend for the VISP Marketplace. The VISP Marketplace builds on top of the Docker Registry [101] to provide the Operator Registry as well as an efficient and easy to use foundation for the Operator Distribution. This technology decision allows us furthermore to deploy the Operator Instances either on a private OpenStack instance, as currently supported by the VISP Runtime, or on commercial Docker Container Hosting services [111]. Besides the deployment-related aspects, the Docker tool stack also provides a basic monitoring infrastructure which is used to monitor the resource consumption of the Operator Instances. In contrast to established SPEs, our data processing model does not require a specific programming language or API usage to realize SPAs. As long as the Operator Images provide the basic set of features as described in Section 5.3.2, the operator developers are able to use any technology stack to implement the data processing functionality. This technology stack can differ based on the concrete stream processing functionality of the operator. For simple functionalities, such as filtering data, it may be sufficient to implement a custom solution whereas more complex ones, like the aggregation of data, may be built based on established SPEs, such as Apache Storm, to use already existing stream processing capabilities provided by these SPEs.

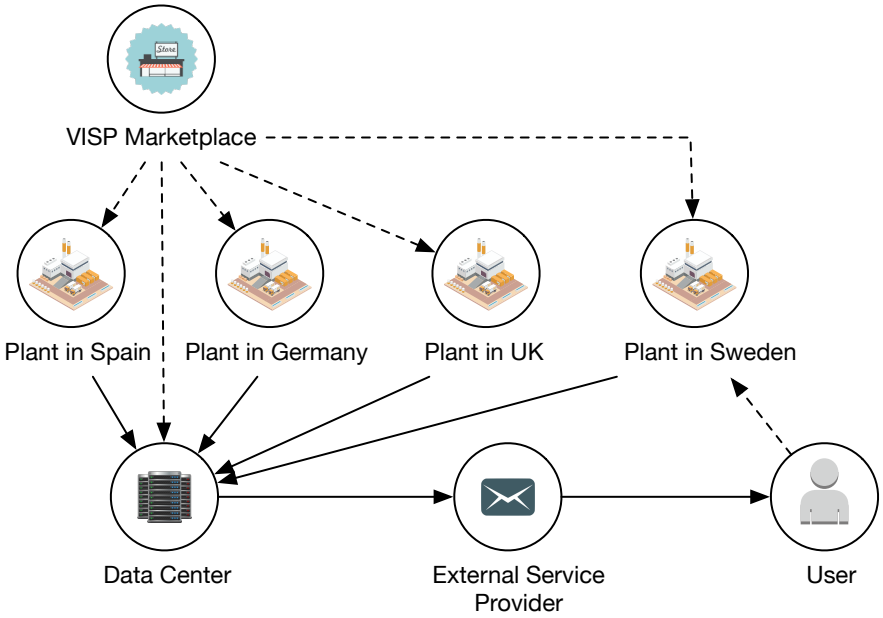


Figure 5.4: Evaluation Scenario

5.4 Use Case Evaluation

To discuss the feasibility and the usability of the system design for the VISP, we conduct a use case evaluation based on the motivational scenario described in Chapter 4. For the evaluation, we consider multiple identical manufacturing machines located in four manufacturing plants, whose data is ultimately sent to a human supervisor as depicted in Figure 5.4. Here, the configuration steps for a new SPA are visualized by the dashed arrows, while the data flow based on the SPA is shown by the solid arrows.

In the following paragraphs we discuss the whole process of realizing a new SPA. The first step for creating a new SPA is to analyze the use case, i.e., the monitoring scenario, and to identify its required functionality. Based on this requirement analysis, the SPA user then checks the Operator Registry to identify already available operators. For our scenario, we assume that the Operator Registry already offers several operators: The source operators and the transformation operator are already provided by the manufacturing machine producer free of charge because the producer may want to support the integration of their manufacturing machines. The filter operators and the OEE-related operators are also available on the VISP Marketplace. Since these operators are provided by third party developers, it is required to pay one time fees to integrate them in topologies. Furthermore, the Inform User operator is also available as an external operator. The only missing operator is the domain-specific one: the Generate Report operator.

This remaining operator needs to be implemented by the user based on the APIs provided by the VISP Ecosystem as described in Section 5.3.2. As soon as this Operator Image is implemented, there are two possibilities to make it available for the Topology Builder. The first approach is to upload the operator, i.e., the Operator Image, to the Operator Registry and make it either publicly available for other users or flag them as private. Either way, they can be used to design the monitoring SPA and are also available for the deployment on VISP Runtime instances. The second approach is to create an operator stub, which only contains the metadata of the operator. This stub is then uploaded to the Operator Registry to allow the designing the monitoring SPA in the Topology Builder. Nevertheless, the Operator Images need to be deployed manually to the VISP Runtime instances to operate the SPA. While the first approach is the recommended one, it is sometimes necessary to keep the Operator Images within the companies premises' to ensure the secrecy of data processing algorithms. When all operators are available in the Operator Registry, the user can start to design the topology for the SPA by means of the Topology Builder. Simultaneously to the topology design, the user needs to deploy the VISP Runtimes. Therefore, the user instantiates the pre-packaged VM image, containing all VISP Runtime components, on cloud resources for the centralized data processing as well as three times on computational resources within the factories premises'. After instantiating these VISP Runtimes, the user needs to provide the credentials for the VISP Marketplace and the computational resources for the Operator Instances as well as the location of other VISP Runtime instances. After completing the topology design, the user uploads the topology for the SPA to the VISP Runtime located in Sweden. There the Topology Configuration component analyzes the topology and forwards the relevant information to all other affected VISP Runtimes. All affected VISP Runtimes then obtain the required Operator Images from the VISP Marketplace, perform billing operations, and wire the operators as described in Section 5.3.4. This concludes the design time and deploy time aspects of the system and the SPA is ready to process data as visualized in the graphical user interface of the VISP Runtimes (see Figure 5.5). The runtime aspects of the predictive maintenance topology do not require any user interactions. The VISP Runtime takes care of deploying sufficient replicas for Operator Instances by using different elasticity approaches that will be introduced in Chapter 7 and 8, to handle the potentially volatile data input and, therefore, enable elastic SPAs. The VISP Runtime also provides several tools, which support the operational aspects for expert users to debug SPAs or operators, like the Failure Management component that represents a single point to obtain all failures from the otherwise hardly accessible Operator Instances.

5.5 Discussion

The use case evaluation shows that the VISP Ecosystem resolves the challenges identified by us based on the motivational scenario (see Section 5.2) as well as those identified by Mineraud et al. [41], like the support of application developers or the need for dedicated IoT marketplaces.

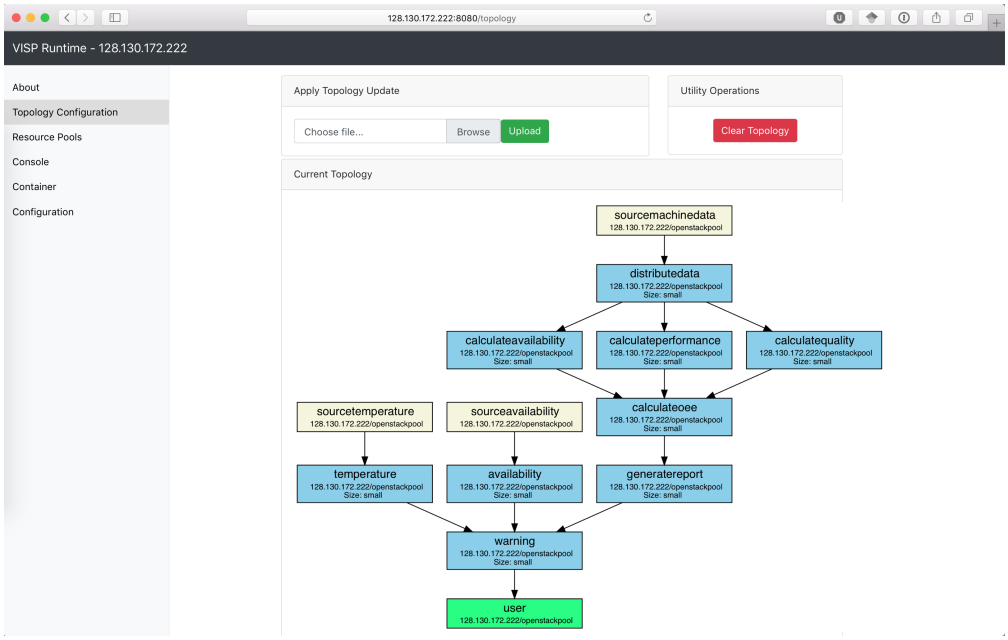


Figure 5.5: Deployed Topology Shown in the Web UI of a VISP Runtime

First, the modular system design for SPAs based on individual source operators and transformation operators allow integrating heterogenous IoT devices as well as already existing SPEs. Whenever a user wants to integrate a new IoT device, into the VISP Ecosystem, it is required to implement one dedicated source operator. The implementation of this source operator should typically be done by the IoT device manufacturer, since the manufacturer has the required knowledge to transform the often binary data, into a textual representation. Although, each new IoT device requires a specific data handling approach, and therefore a custom source operator implementation, our system design only requires one operator, which can be reused by all other SPAs. Second, the VISP Ecosystem facilitates a privacy-sensitive data processing approach. State-of-the-art IoT platforms, like Groovestreams [49], ThingWorx [51] or the cloud computing infrastructures by Amazon [52], Microsoft [54] or Google [53], require the data to be uploaded to a public cloud. The VISP Ecosystem can be deployed either on a public cloud, like the other IoT platforms, but it can be also deployed on a private cloud or on fixed computational resources due to its self contained system design. This privacy-preserving deployment allows the data to be processed within the premises of the company and to maintain the control over the data at all times. Due to the flexible deployment possibilities of the VISP Ecosystem, this privacy-sensitive approach can be applied to the whole SPA, or only to parts of the SPA, which is currently not supported by established SPEs. The distributed deployment capabilities do not only improve the granularity of privacy control for data processing, but also enable the benefits of different computational resource types, like the real-time processing capabilities on edge resources close to the IoT devices or the more cost-efficient, but slower, processing capabilities in a centralized cloud [94].

Next to supporting the distributed deployment approach, the VISP Ecosystem also explicitly addresses the challenges of user support and operator reuse. In order to address the user support, the VISP Ecosystem provides the Operator Registry as a graphical user interface and the Topology Builder to enable an easy topology design for SPAs. This enables domain experts to design SPAs and can therefore help to raise the acceptance rate for the VISP Ecosystem. Furthermore, the VISP Ecosystem also supports the exchange of operators among all SPA users since the Operator Images represent self-contained operators that can be executed on any VISP Runtime. The Operator Images also require only a minimal set of requirements, e.g., subscribing to the Messaging infrastructure, when they are implemented. These minimal requirements lower the entry barrier for potential users of the VISP Ecosystem, since it allows users to quickly package already existing stream processing implementations and participate in the VISP Ecosystem.

Our use case evaluation, as well as further feasibility tests of our prototype, show that the VISP Ecosystem addresses all challenges that we have identified based on our motivational scenario. In addition, the modular system design of VISP allows to realize a distributed deployment of the individual Operator Instances, based on their SLAs, but also for the core components of the VISP Runtimes to provide a scalable backend infrastructure. We can also see, that our topology design approach for SPAs allows a better usability in contrast to established SPEs. Due to the fact that each established SPE requires the usage of a specific API to implement the SPA, each new SPA needs to be implemented from scratch, including the operators, which results in high (re-)implementation efforts if the user wants to switch SPEs. Our approach addresses this problem by applying the VTDL to design an API agnostic topology for SPAs, which is then executed by individual operators that can be reused across different SPAs. Besides the decreased implementation effort for future SPAs, our approach also supports the monetization of operators for operator developers. This monetization aspects are vital for attracting operator developers to realize a vivid ecosystem. VISP currently supports several business models for operators, but it also allows the integration of additional ones in the future, like bundling strategies for IoT devices or the monetization of the provided data by the sensors of IoT devices.

5.6 Summary

In this chapter, we present an holistic approach for realizing elastic data stream processing topologies for the IoT. We propose the VISP Ecosystem that supports the user at design time by reusing existing components for SPAs and by providing a graphical user interface for creating new SPAs. These SPAs can then be deployed in any VISP Runtime, which automatically propagates required configuration settings to other VISP Runtimes, autonomously provisions computational resources, and wires the operators for the SPA. The VISP Runtime also takes care of using sufficient computational resources to comply with given SLAs at any time. In conclusion, we can say that the above-mentioned contributions lower the entry barriers for users to participate in the VISP Ecosystem dramatically and allow domain experts to design and operate SPAs easily and efficiently.

CHAPTER 6

Describing Distributed Stream Processing Applications

In this chapter, we analyze the structural and organizational requirements for SPAs that are running in distributed environments. Based on this requirements analysis we extend the SPL, an established topology description language which was introduced for IBM System S and introduce the Vienna Topology Description Language (VTDL). The VTDL addresses the deficits for state-of-the art topology description languages and serves as a topology description model for SPAs that considers distributed deployments as well as a fine granular QoS model. In order to evaluate the feasibility of using the VTDL, we provide a reference implementation of the VTDL within the VISP Ecosystem. Based on this implementation we evaluate different deployment scenarios and show that the usage of features enabled by the VTDL reduces the time required for deploying and updating SPAs in distributed environments by up to a factor of 18 times.

6.1 Overview

Up to now, most SPEs do not consider the geographic location of IoT devices and cloud-based computational resources that are used to run SPEs [24]. Due to the lack of distributed runtime environments, it was not required to consider this distribution aspect for designing and operating SPAs. Therefore, there are hardly any topology description approaches that address this aspect as discussed in Section 3.4. Besides the geographic distribution aspects, the topologies of today's SPAs are often subject to change [112]. The changes are mostly triggered by operational aspects at run time, e.g., due to reconfigurations of operators, like adding other Operator Instances. These reconfigurations can occur due to QoS-related aspects, like changes in the underlying computational infrastructure, e.g., adding or removing computational resources [113], or due to changes in the incoming workload which require the replication of operators.

In addition to these operational changes, there are also other reasons to update the topologies of SPAs. These reasons range from updates for single operators, e.g., to fix software bugs, to organizational changes, like the addition of new data sources and consumers, e.g., sensors and users, or new legal restrictions that enforce the processing of data within a distinct geographic area [114].

To improve the description of SPAs and to enable their deployment in geographically distributed environments, we first identify the required features for next-generation SPEs, like the VISP Ecosystem. Then, we present the VTDL, which extends the concepts of the SPL [28] to support the required features identified during our analysis. Additionally, we have integrated the VTDL into the VISP Ecosystem [108] to provide a reference implementation. For this implementation we have furthermore designed a protocol which allows the use of the features enabled by the VTDL, like the isolated operator updates while continuing the data processing in the unaffected part of the SPA or automatic topology updates across multiple geographic locations.

The remainder of this chapter is structured as follows: First, we refine the motivational scenario from Chapter 4 to model different deployment possibilities as well as external events which can trigger changes for the topology of an SPA in Section 6.2. Based on the motivational scenario, we identify several features that are essential for next-generation SPEs in Section 6.3. In Section 6.4, we then introduce the VTDL and in Section 6.5 we discuss the requirements for an SPE to support the VTDL. Finally, Section 6.6 presents the evaluation of VTDL as well as a discussion thereof and Section 6.7 concludes the chapter.

6.2 Extended Motivational Scenario

6.2.1 Topology Structure and Deployment Scenarios

For this scenario, we refine the motivational scenario presented in Chapter 4 to consider the data flow and deployment possibilities among different geographic locations as visualized in Figure 6.1. Besides the production facilities in the United Kingdom, Sweden, and Spain visualized in Figure 4.1, we also consider a production facility in Germany that has the same geographic location as the data center. Each production facility hosts a different amount of manufacturing machines which are equipped with sensors. The goal of the monitoring SPA is to collect the information from all four production facilities and combine them in a report, which is then provided to the human supervisor alongside any temperature- or failure-related alerts from the manufacturing machines. Although the topology is identical for all four production facilities, there are different deployment possibilities that result in different data flows within the SPA, as shown in Figure 6.1. The first deployment scenario is to split the topology into two parts and deploy all metric calculation operators near the data source, e.g., on a private cloud, as done for the plants in the United Kingdom and in Spain. Due to the high amount of data from the manufacturing machines, it is reasonable to preprocess the data next to the data source and only transfer the filtered data over the Internet.

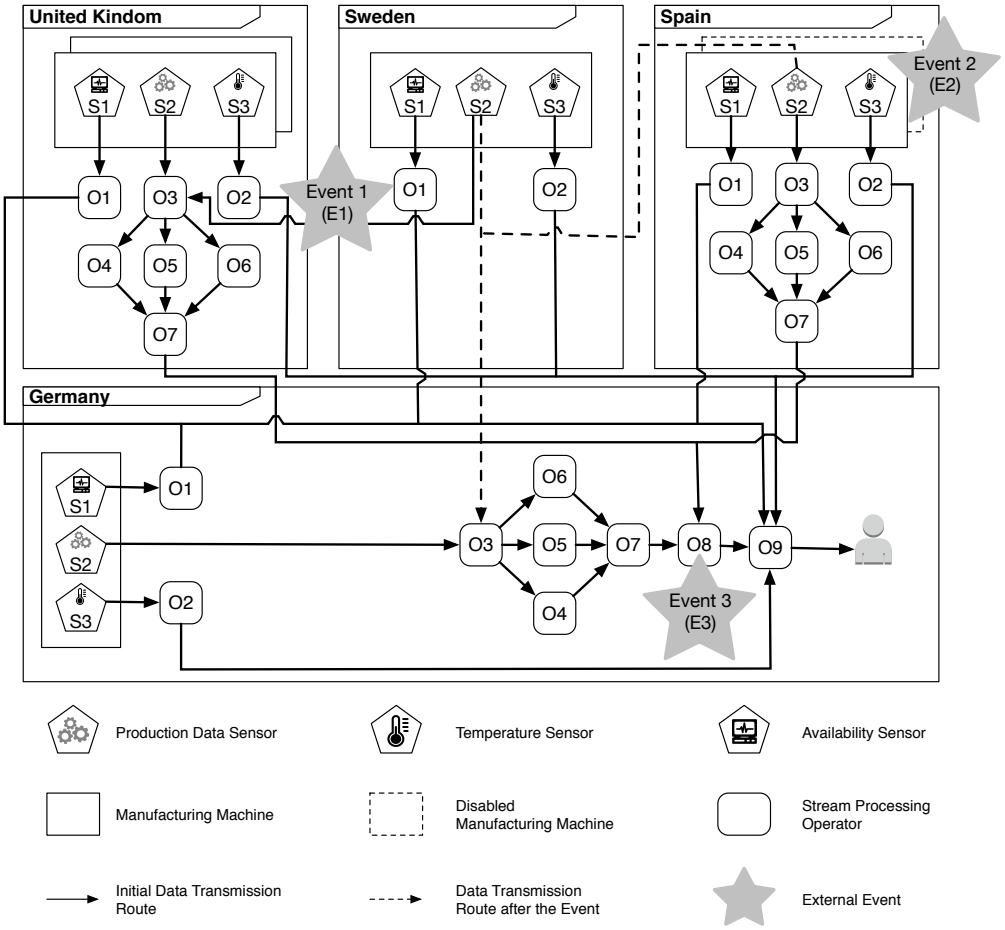


Figure 6.1: Motivational Scenario

For the second deployment scenario, we consider the production facility in Sweden, which only hosts the filter operators locally and forwards all other data to the private cloud located at the production facility in the United Kingdom. This common usage of computational resources requires only little computational resources for the plant in Sweden, relatively low network cost due to the small number of machines, and little geographic distance as well as a higher usage of the private cloud in the United Kingdom. After calculating the metrics on computational resources close to the data sources, the results are then sent to the data center in Germany, which hosts all other operators of the SPA.

Finally, the last deployment scenario can be found for the production facility in Germany. For this scenario, all operators are located within the data center which is in the same geographic location as the production facility. This geographic co-location avoids any data transfer from the manufacturing machines over the Internet and is therefore the most network-efficient solution.

6.2.2 Topology Changes

While the topology in Figure 6.1 may appear rather fixed at first sight, there can be several events which require an update to the deployment of the SPA. These events can occur at any point in time at run time, which requires bot the SPA as well as the SPE to be flexible throughout the whole operation of the SPA.

The first event (E1 in Figure 6.1) is a communication outage. In our scenario, the manufacturing machines in the United Kingdom and Sweden use the same computational infrastructure in the United Kingdom to calculate the OEE. This setup is feasible as long as the network connection between these two production facilities is intact. However, whenever there is a communication outage, the SPA needs to be reconfigured to continue data processing. Based on the geographic location, it is possible to reroute the raw data to Germany to compensate the communication outtake and resume data processing. The alternative data flow is visualized by the dashed line in Figure 6.1.

The second event (E2) represents a volume reduction for the SPA. Occasionally, manufacturing machines have downtimes and the full data processing capacities of the SPE are not required anymore. Whenever the full processing capabilities are not required anymore, it is feasible to release computational cloud resources and reroute the data to reduce the total operational cost. For E2, several manufacturing machines are switched off in Spain and similar to the previous event it is possible to reroute the raw production data to Germany for processing.

While the first two events are triggered by operational aspects, it may also be required to replace individual operators due to organizational reasons. For the third event (E3), we consider a software update for the Generate Report operator (O8). This software update fixes an internal flaw of the operator implementation, but the overall functionality of the operator, i.e., the input and output data structure, remains the same. For this update, it should not be required to redeploy the whole topology for the SPA, as required for most of the established SPEs, like Apache Storm or Apache Spark. The SPE should only need to buffer the incoming traffic for a short period of time until the new Generate Report operator is in place and can continue with its operations.

6.3 Features for Next-generation Stream Processing Engines

Based on the extended motivational scenario, we identify some basic and six next-generation features that are not available in today's SPEs (see Section 3.4). The main difference between the basic features and the next-generation features is that the basic features are already mostly covered by existing topology description languages, such as SPL [28] or CQL [85].

6.3.1 Basic Features

The primary feature of any topology description approach is to define how data sources, operators, and information consumers, are connected to realize a SPA. Stream processing topologies are usually represented as directed acyclic graphs [28], where vertices represent the operators and edges represent the data streams between the operators. A data source feeds data into the topology and has no incoming data streams. Hence, each topology requires one or arbitrary many data sources. The data provided by the sources is then processed by one or more operators. Operators execute user-defined code, whether it is a simple operation (e.g., filtering, aggregation) or a more complex one (e.g., regression, classification) [10]. Operators can obtain data from arbitrarily many vertices, i.e., data sources or other operators, and emit new data to other vertices, i.e., other operators or data sinks. Data sinks or consumers represent the endpoints of an SPA since they only consume data, i.e., only have incoming edges and each SPA needs at least one data sink.

6.3.2 Next-generation Features

Deployment Preferences (F1) The most important feature for geographically distributed SPAs is deployment preferences for individual operators [22]. While this is not relevant for SPAs in a single location, it becomes crucial for geographically distributed ones. Each operator needs to be able to provide a set of admissible deployment locations where it can operate and satisfy real-world constraints. These constraints mainly affect data sources, e.g., a temperature sensor or a camera, which are mounted on a fixed location and cannot be relocated. Additionally, it may also be prohibited to transfer specific data, e.g., medical data, to certain locations like public clouds [114], which also limits the deployment of some operators.

QoS Compliance (F2) Although QoS compliance is commonly used for software services such as SPAs [67], it is, to the best of our knowledge, not considered by state-of-the-art SPEs on an operator level. While the application-level QoS compliance may be sufficient for most users, research for microservices has shown that a more fine-grained approach, i.e., on an operator level, allows to identify bottlenecks [115]. Based on such a bottleneck analysis, resource provisioning algorithms can achieve lower costs by only scaling specific operators instead of the whole SPA. Furthermore, an operator-level QoS compliance also allows the integration of external operators on a SaaS basis into SPAs.

Fault Tolerance (F3) To compensate operator failures or hardware failures, it is imperative that SPEs are capable of applying automatic failure compensation mechanisms. Nowadays, SPEs already provide basic fault tolerance mechanisms, like the automatic restart of operators whenever they fail [116]. Nevertheless, it is required to also support more sophisticated fault tolerance mechanisms, like deploying an updated topology for SPAs to reroute the traffic (as required for event E1). This feature builds on top of the QoS compliance feature (F2), which allows the SPE to detect operator failures, e.g., based on a high latency, or infrastructure outages.

Operator Composability (F4) To ensure the compatibility among the operators, it is required to provide basic semantic annotations, regarding incoming as well as outgoing data types as already proposed for the streaming data itself [117]. These semantic annotations can be used to check whether operators are compatible and in a further step to also apply an automatic semantic operator selection. This would allow the user to only provide an abstract description for the operator task, and the SPE can autonomously create the topology, which reduces the users' workload for creating SPAs. This feature is already available for other domains like sensor networks [118], but is still missing for SPAs running on SPEs.

Topology Modifications at Run Time (F5) The need for topology modifications at run time, as required for the third compensation mechanism (E3) in the motivational scenario, has also been identified in the literature [119], [120]. Stream processing topologies are often deployed for long-term data processing, which makes it hard to apply minimal updates, such as bug fixes for individual operators, without redeploying the whole SPA. Therefore, the SPE needs the possibility to pause the data flow for individual operators, to apply the update. As long as the operator composability (F4) does not render any inconsistencies, it is sufficient to only pause the processing for the operator that needs to be updated. This allows to update SPAs with software updates or enable fine-grained failure compensation measures.

Different Data Transfer Modes (F6) Due to the geographically distributed deployment, it is required that some operators are connected via the Internet instead of a local connection, which is common for centralized SPE deployments. The network connection of the Internet may result in a high communication overhead because each data item is sent individually, which is only efficient in local settings. To mitigate this communication overhead, it is often more efficient to create so-called microbatches, i.e., to aggregate multiple data items and send them as a single group (or batch) over the Internet. This reduces not only the communication overhead and network load, but also the performance of the data processing [83].

6.4 VTDL – Vienna Topology Description Language

The goal of the VTDL is to support both the basic features, as already present for existing topology description approaches, as well as the next-generation features that we have identified in the previous section. We have chosen the SPL [28], which was initially developed for IBM System S as a starting point, since it already provides some of the identified features compared to other description approaches (see Section 3.4). The description of the SPA topology is provided through a VTDL file, which contains a list of the operators, their roles, and attributes, as well as their information on how they are connected.

Listing 6.1: Excerpt of a Topology Description

```
$temperature          = Source() {
  concreteLocation    : ":::::ffff:8083:c001/cpu",
  type                : "temperatureSensor",
  outputFormat        : "temperature"
}

$monitor              = Operator($temperature) {
  allowedLocations    : *,
  poolPreferences     : "cpu gpu",
  concreteLocation    : ":::::ffff:8083:c001/cpu",
  inputFormat         : "temperature",
  type                : "monitorTemperature",
  outputFormat        : "alert",
  stateful            : "false",
  replicationAllowed  : "true",
  responseTime        : "0.5",
  compensation        : "mailto:admin@tuwien.ac.at"
}

$informUser           = Operator($availability, $report, $monitor) {
  allowedLocations    : ":::::ffff:8083:c001"
                        ":::::ffff:8083:c002",
  inputFormat         : "alert, report",
  type                : "informUser",
  outputFormat        : "message",
  queueLength         : "200",
  protocol            : "microbatch/50items",
  compensation        : "deploy:www.visp.io/backup.vtdl"
}

$user                 = Sink($informUser) {
  concreteLocation    : ":::::ffff:8083:c002/general",
  inputFormat         : "message",
  type                : "user"
}
```

In the VTDL, each vertex of a topology is identified by a textual identifier, which is prefixed with a $ character, as presented in Listing 6.1. Directly after the identifier, the role of the vertex is indicated, namely *Source()*, *Operator()*, or *Sink()*; the latter two roles need to take at least one operator identifier as the input parameter, to describe the data flows between the operators. When the operator needs to receive data from several upcoming sources, their identifiers are specified in a comma-separated list, e.g., *$availability, $report, $monitor*. In addition to the structural definition, each vertex is assigned a set of key-value pairs, as listed in Table 6.1, which includes attributes of interest for the deployment time and run time management. These key-value pairs can be categorized into three categories: required attributes, attributes with default values, and optional attributes.

The most important attribute is the *type* of the vertex, which describes the functionality of the operator, whose concrete implementation is resolved by the SPE. The second most important attribute of the VTDL is the location aspect of the operator as required for F1.

Table 6.1: Operator Attributes

Attribute	Values	Default	Description
type	string	–	Operator logic
allowedLocations	IP+	–	Possible locations
poolPreferences	string+	general	Hardware preferences
concreteLocation	IP/poolIdentifier*	–	Preselected location
inputFormat	string+	–	Accepted inputs
outputFormat	string	–	Data output format
responseTime	numerical	5s	Response time
queueLength	numerical	100	Buffered items
stateful	{true | false}	true	Operator statefulness
replicationAllowed	{true | false}	false	Operator replication
protocol	{stream | microbatch:<X>items | microbatch:<X>ms }	stream	Processing mode
compensation	{redeploySingle | redeployTopology | deploy:<URL> | mailto:<email> | none}	none	Failure recovery mode

Each location is identified by an Internet Protocol (IP) address (the *allowedLocations* attribute), which defines the concrete location of the SPE, data source or sink and is a must have attribute. The default value is *, which does not restrict the deployment locations. Nevertheless, it is also possible to restrict the locations by providing a list of allowed ones.

Furthermore, each geographic location may have different resource types available, e.g., resources with specific hardware aspects, like solid state drives, high-performance CPUs or GPUs. VTDL assumes that these resources are handled as resource pools, e.g., a resource pool with high-performance CPUs, and the SPA designer can indicate deployment preferences by providing a list of *poolPreferences*. This allows the SPE to deploy the operators according to their resource preferences if the specific hardware is available. If the specific hardware is not available, the SPE will use any resource that is available. The *concreteLocations* value is composed of an IP and the pool identifier. This attribute needs to be provided at design time for data sources and sinks since they are fixed. For operators, this attribute is optional, but it can be used to indicate concrete deployment instructions. All other concrete locations are selected at deployment time, as discussed in Section 6.5. Later at run time, this concrete location can be updated if necessary to improve the performance of the SPA, e.g., to recover from a failure or to meet QoS requirements.

Besides the type attribute, the VTDL also features several attributes for the semantic description of operators to enable composability checks (F4) or automatic semantic operator selections. Due to the fact that these composability checks are not essential for the data processing, these attributes are optional. These composability checks are represented by the *outputFormat* attribute that defines the semantic type of the data, which is forwarded to succeeding vertices for each source and operator, and the counterpart is *inputFormat*, which is used by all operators and sinks.

The next category of attributes considers the QoS aspects of the operator (F2). Up to now, the VTDL considers the *responseTime* for one processing operation and the *queueLength*, which measures the amount of data items waiting for processing. Nevertheless, there are also other QoS aspects like the maximum CPU or memory usage for a particular operator, which could be easily added as attributes for the operators. To inform the execution framework regarding the operator behavior at run time, VTDL comprises three more attributes: stateful, replicationAllowed, and protocol. The *stateful* attribute describes whether the operator is stateful, i.e., computes the output data using the incoming data together with an internal state information. This attribute is required to indicate the effort required to migrate operators, because it is easy to relocate stateless operators, but it requires extra effort to migrate the state for stateful ones [68]. The *replicationAllowed* attribute describes whether multiple instances of the same operator can be executed concurrently by the SPE, whereas the SPE needs to take care of the concrete partitioning scheme of the data. Both operational attributes *stateful* and *replicationAllowed* are optional. To apply a conservative approach to preserve the application integrity, we recommend to assume by default each operator as stateful and with no replication allowed. Next, the *protocol* attribute (F6) allows for defining the data transmission type (see the *informUser* operator in Listing 6.1). It can take two types: *stream*, which is the default option, and *microbatch*, which requires collecting data in groups (i.e., batches) before applying the operator function. For the latter option, the batch size for the microbatch is provided by either the number of items for the microbatch, as shown in Listing 6.1 or by the time in milliseconds, e.g., 100 ms, for sliding intervals. The default setting for this attribute is the individual transmission and it only needs to be set to enforce the microbatch transmission.

The final attribute is the *compensation* attribute (F3), which describes the failure compensation mechanism in case one operator becomes unavailable. Currently, VTDL supports four different failure compensation mechanisms, as shown in Listing 6.1. The first compensation mechanism scope is motivated by the literature [121] and is identified by the keyword *redeploySingle*: it requires to deploy a new instance of the faulty operator, to cache the incoming data during the new instance startup time, and finally to replay the cached data. The second compensation mechanism, identified by *redeployTopology*, requires restarting the whole SPA, thus resulting in a possible loss of currently cached and processed data. This potential data loss is also the case for the third compensation mechanism, which allows to deploy an alternative topology when a failure of the current one occurs (as required for E1).

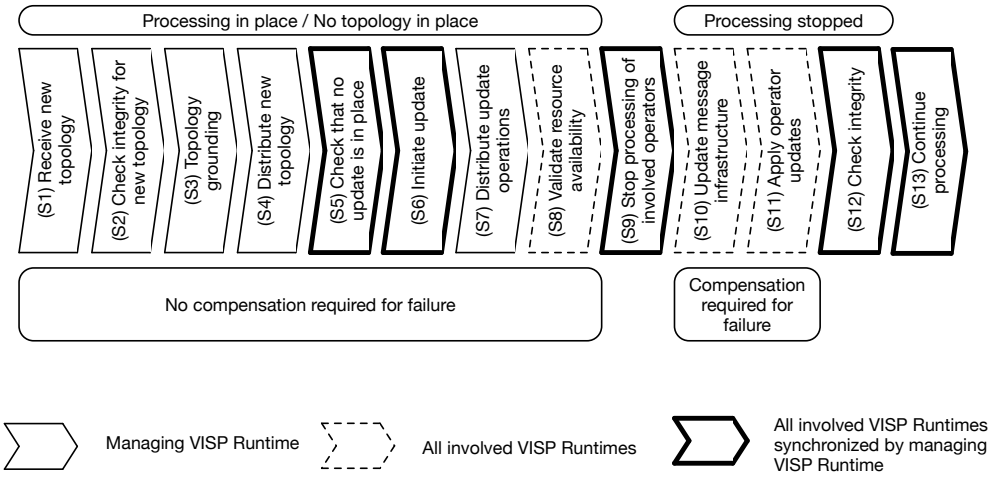


Figure 6.2: Topology Update Procedure

The option *deploy:<URL>* specifies this compensation mechanism, where the Uniform Resource Locator (URL) indicates the location of an alternative topology that is deployed as a replacement for the existing one. The last two options are strictly speaking not a compensation mechanism, because they only allow the SPE to notify a user via email, i.e., *mailto:<email-address>*, or to simply ignore the failure indicated by *none*.

6.5 Management for the VTDL

To enact topologies, which are defined based on the VTDL, SPEs are required not only to parse the incoming topology definitions but also to apply topology modifications at run time (F5). Therefore, we are going to discuss the required capabilities for SPEs based on the reference update procedure implemented for the VISP Runtimes and shown in Figure 6.2. The overall update procedure consists of 13 steps (S1 – S13), which are conducted by a managing VISP Runtime, i.e., the VISP Runtime that first receives the new topology description as well as all other involved VISP Runtimes, which are affected by the topology instantiation or update. Each new topology operation, including its initial deployment, is triggered by the upload of a VTDL file to any VISP Runtime (S1). This VISP Runtime is then promoted to be the managing VISP Runtime and checks the composability of the topology (S2) based on semantic annotations provided by the VTDL file. Then, the managing VISP Runtime evaluates if each operator is already assigned a concrete location. If this is not the case, the managing VISP Runtime assigns concrete locations based on the available locations in the topology grounding step (S3); this location is selected based on the allowed locations. The VISP Ecosystem currently supports two grounding approaches, i.e., selecting the first suitable location for a given pool preference for an operator or selecting a random location based on the available ones.

After the preparation phase, the VISP Runtime informs all other involved VISP Runtimes of the update. To apply only one update at a time, the VISP Runtime checks that no other updates are in place. Therefore, the managing VISP Runtime starts a synchronized query, asking for the status of the other VISP Runtimes (S5). If no other updates are in place, it initializes the update (S6): Each of the involved VISP Runtimes is blocked for other updates, and the managing VISP Runtime distributes the updates to all involved VISP Runtimes. The concrete number of updates depends on the actual changes for the SPA and is derived by comparing the currently deployed topology with the new one and by generating an update command for each change between the topologies. The update sets either consist of creation commands for new operators, deletion commands for operators which are not required anymore, or reconfiguration commands for the messaging infrastructure if a data flow is redirected. When there is no topology available, the set of update commands comprise of the whole topology. But in most cases, there are only small changes for existing topologies which result in partial updates. After all required update commands are distributed to the affected VISP Runtimes, they evaluate whether the update is feasible based on the locally available computational resources. If this is the case, the managing VISP Runtime is informed of the successful update checks; otherwise, an exception is raised (S8). Up to this step (S8), no changes have been applied to any already running operators, which allows a stop of the topology update operation without any compensation mechanism required. The first actual changes are applied in the next step (S9), which triggers a processing stop for all affected operators. This marks a major distinction in contrast to other SPEs, which need to terminate the complete SPA before applying any new or updated topologies and therefore suffer downtimes. The VTDL approach allows the SPE to continue the data processing for all operators that are not affected by the update. After stopping the data processing, each involved VISP Runtime applies the updates to its messaging infrastructure (S10), removes obsolete operators and instantiates new ones (S11). These two steps represent the only critical steps, where manual compensations may be required if any reconfiguration fails. As soon as all update commands have been executed, all VISP Runtimes apply another composability check to ensure that the topology is enacted as intended (S12) and, if this check does not raise any issues, the processing is continued for all operators (S13).

6.6 Evaluation

6.6.1 Evaluation Scenarios

To evaluate the VTDL and the reduced management overhead for applying updates, we conduct a case study consisting of several scenarios based on the motivational scenario. These scenarios are evaluated regarding duration as well as required user interactions against a baseline approach. The baseline approach represents the state-of-the-art for most established SPEs and does not support partial updates. In contrast to the VTDL approach, the baseline approach also does not supports SPAs across multiple geographic locations which requires the user to upload the topology for each geographic location.

1. New Topology in one Location For the first scenario, we assume a topology deployment for a single location. This scenario represents the state-of-the-art for established SPEs and requires no update activities for other SPEs.

2. New Topology across four Locations The second scenario represents an initial deployment for the motivational scenario. Here, the VTDL approach is only required to upload the topology to one VISP Runtime in one location, whereas the baseline approach requires to upload a subset of the topology to all involved locations one after another. The first and the second scenario consider both the messaging infrastructure configuration as well as the operator instantiation for the SPA.

3. Network Disruption (E1) For the network disruption scenario, the VTDL approach can rely on the automatic failure detection and compensation of VISP Runtimes to detect network disruptions between two regions and to reconfigure the data flow based on a given alternative topology. This feature is not available for other SPEs and therefore, the evaluation of the baseline approach for this scenario is not possible.

4. Resource Reconfiguration (E2) The resource reconfiguration scenario evaluates the time to reconfigure the data flow between a sensor and an operator for the VTDL approach. This reconfiguration only requires an update for the messaging infrastructure, since the operators are already running in the target location. For the baseline approach, it is required to upload a new topology for all the affected regions, which also requires the deployment of new operators.

5. Single Operator Update (E3) The last scenario evaluates the time required to update a single operator. For this scenario, it is sufficient for the VTDL approach to only update the specific operator, whereas the baseline approach requires the redeployment of the complete topology for the affected location.

6.6.2 Evaluation Setup

To conduct the evaluation, we set up four VISP Runtimes, which represent the individual locations of the motivational scenario, on an OpenStack-based private cloud [106] and on three regions of Amazon EC2 [105] to simulate the different geographic regions.

The interactions are conducted by Selenium scripts [122] to eliminate any human-based delays for the evaluation. Each task of these Selenium scripts, e.g., opening a Webpage, uploading a VTDL file or removing an enacted topology, is counted as an individual interaction whereas the duration is assessed by the total runtime of the Selenium script for the complete scenario.

Table 6.2: Evaluation Results

	VTDL Approach		Baseline Approach	
	duration (ms)	interactions	duration (ms)	interactions
1. New Topology in one Location	54977.33 (σ = 982.03)	3	54977.33 (σ = 982.03)	3
2. New Topology across four Locations	40953.67 (σ = 431.03)	3	68098.00 (σ = 1701.86)	12
3. Network Disruption (E1)	7532.67 (σ = 1921.05)	0	-	-
4. Resource Reconfiguration (E2)	3301.33 (σ = 281.56)	3	54056.33 (σ = 1554.08)	8
5. Single Operator Update (E3)	5688.67 (σ = 363.53)	3	35623.33 (σ = 1169.39)	4

6.6.3 Results and Discussion

To eliminate any potential corruption, e.g., side effects by other cloud users, due to the evaluation in a cloud environment, each scenario was executed three times based on the VTDL files which can be found in Appendix A. Table 7.2 shows the average results alongside with the standard deviations of the individual measurements. For the first scenario, there is no difference between the VTDL approach and the baseline approach, because both approaches follow the same instructions. Each topology update requires three interactions: opening the Web-based user interfaces, selecting the desired VTDL file in a file chooser, and initiating the update procedure by clicking on a button. The overall scenario takes about 55 seconds, which is mainly due to the instantiation of a Docker Container for each operator. The Docker Images are already available for all scenarios to avoid any network-related delays for downloading the Docker Images.

The first difference between the VTDL approach and the baseline approach can be seen in the second scenario, where the topology is deployed across four locations. The VISP Runtime is capable of deploying the topology to multiple locations in parallel which results in a shorter duration compared to the first scenario, although the update instructions need to be propagated over the network. For the baseline approach, it is required to upload the individual parts of the topology one after another to the individual SPEs to ensure the correct data flow wiring among the different regions. This sequential approach requires significantly more interactions, i.e., four times as much, than for the VTDL approach, which results in an about 40% faster topology instantiation. Hereby, the majority of the deployment time can also be attributed to the startup duration of the Docker Container.

The first event scenario (E1) can only be evaluated for the VTDL approach, because other SPEs do not support any sophisticated failure compensation on an operator level. For this scenario, we have selected the *deploy* option as compensation, which obtains a new VTDL file from a predefined location and applies the update.

Here, it is sufficient to reroute the traffic from Sweden to Germany, which results in a low duration between the event and the topology update. The event is detected by the Monitoring infrastructure of the VISP Runtime, which evaluates the availability and connectivity among the individual operators every ten seconds. The detection of the outtake takes on average 5 seconds, but in the worst case this can take up to 10 seconds, depending on the cycle of the monitoring interval. These changing detection times result in different compensation durations, which is also indicated by the high standard deviation for this scenario. This scenario also does not require any user interactions since the failure compensation is conducted autonomously by the VISP Runtimes.

The next scenario (E2) describes an active resource configuration within the topology, which leads to a shorter average duration as for E1. For the VTDL approach, it is sufficient to only reroute the data flow between the sensor and the operator, whereas the baseline approach requires the removal and redeployment of two sub-topologies, i.e., for Spain and Germany, which results in an update duration of almost 54 seconds. That is about 18 times as long as for the VTDL approach. The baseline approach also requires five interactions more than the VTDL approach, because it requires two topology removals and two topology uploads.

The last scenario (E3) requires the removal and update of the topology within one VISP Runtime for the baseline approach compared to the update of a single operator. For the new instantiation, all operators are newly deployed which results in a six times higher duration compared to the VTDL approach, where only one operator is removed and the updated one is deployed again.

The evaluation of the VTDL approach against the baseline approach that is used for established SPEs shows that the VTDL-based approach can reduce both the duration for applying changes to a topology as well as the required manual interactions.

6.7 Summary

Within this chapter, we have motivated the need for a new topology description approach by discussing the features for distributed SPAs and next-generation SPEs. Based on these features we have developed and introduced the VTDL, which extends the SPL with the required features to support distributed SPAs as well as fine granular QoS constraints for operators. Besides the abstract notion of the VTDL, we also presented a concrete management mechanism, which is required to use the features of the VTDL within SPEs. This management mechanism has been evaluated based on five scenarios and the evaluation shows that the VTDL approach has a significantly lower update duration for updating topologies compared to traditional approaches. Finally, the evaluation also shows that the VTDL enables new possibilities for SPAs, like automatic failure compensation.

CHAPTER 7

Resource Elasticity for Stream Processing Applications

In this chapter, we present a threshold-based resource provisioning algorithm for SPAs running on SPEs in a distributed environment. Therefore, we introduce a model to formulate and optimize the resource provisioning for individual operators of an SPA based on SLOs as well as resource restrictions. To evaluate our optimization approach, we design a new SPA for monitoring rides of a taxi fleet and deploy the SPA across six different geographic locations. The evaluation shows that our optimization model is able to reduce the cost by 20% with only minimal effects on the QoS compared to an over-provisioning baseline. Besides the cost reduction in contrast to an over-provisioning scenario, the evaluation also indicates an QoS improvement of 72% compared to an under-provisioning baseline.

7.1 Overview

One of the most important challenges for SPEs arises from the large volatile data, which have been first observed for social networks [17]. In order to cope with this challenge, most system providers applied a fixed-resource provisioning scenario, whereas neither an over-provisioning scenario nor an under-provisioning scenario is an optimal solution as discussed in Section 2.2.1. To address this issue, we propose to apply an elastic resource provisioning approach and update the resource configuration at run time depending on the actual data volume for the SPA. Therefore we introduce an elastic resource provisioning model for data stream processing in Section 7.2. This model is then evaluated based on a testbed-driven evaluation presented in Section 7.3 which is then discussed in Section 7.4. To conclude this chapter, we finally summarize the results in Section 7.5.

7.2 Elastic Resource Provisioning Model

The VISP Runtime supports elastic resource provisioning for each operator of an SPA based on monitoring information as discussed in Section 5.3.3. The aim of our elastic resource provisioning strategy is to minimize the cost for computational resources, i.e., VMs, while being compliant with given SLAs that require a near real-time data processing. This cost optimization is based on leasing Operator Instances that are required to guarantee near real-time processing capabilities. In addition it is reasonable to use the leased resources in the most efficient manner according to their Billing Time Unit (BTU). A BTU defines the minimum leasing duration for computational resources, e.g., VMs, and often amounts to one hour like on Amazon EC2 [105]. 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 [123], because releasing a VM before the end of the BTU results in a waste of resources.

For the optimization model at hand, we consider the optimization of one specific operator, which is used within an SPA. This operator is represented by at least one but up to arbitrary many Operator Instances depending on the data volume that need to be processed. Each of these replicated Operator Instances is running on a dedicated VM and all Operator Instances are running independently from each other.

To realize elastic resource provisioning, we define an optimization problem, which consists of the objective function given in (7.1) and nine constraints given in (7.2)–(7.9).

In our optimization problem, we use the decision variable r_i to denote one particular Operator Instance out of the set of all Operator Instances R. The variable R_U indicates the set of all currently running Operator Instances. The current CPU of an Operator Instance load is given by c_{r_i}, its current leasing duration is defined by ld_{r_i}, and we denote the load of the incoming queue for this specific operator as q_{in}.

$$\min \sum_{p \in P} p_i + \sum_{p \in P} p_{i_{BTU}} + u \cdot N + d \cdot N \tag{7.1}$$

The objective function (7.1), which is subject to minimization, comprises four terms. In the first term, we compute the total leasing cost by summing up the assigned values for all Operator Instances for a specific operator. These values can either take the value 1 when currently leased or the value 0 when not leased, as defined by (7.8). The second term sums up the remaining leasing time for each Operator Instance, which has already been paid according to its BTU, as defined by (7.6). This term ensures that those Operator Instances with the smallest remaining usage duration are released first. The remaining two decision variables u (upscaling) and d (downscaling) indicate the required scaling procedures. A value of 1 indicates a required scaling procedure and the default value of 0 indicates that no scaling procedure is required.

$$\sum_{r \in R_U} \frac{c_{r_i}}{R_U} < CPU_{max} + u \tag{7.2}$$

$$q_{in} < Q_{max} + u \cdot N \tag{7.3}$$

(7.2) represents the constraint which triggers an upscaling procedure, whenever the average CPU usage of all running Operator Instances exceeds the CPU_{max} threshold. The second upscaling constraint, defined by (7.3), represents the upscaling decisions based on the load of the incoming queue q_{in}. As soon as the load q_{in} exceeds the threshold of Q_{max}, the variable u becomes 1 and triggers an upscaling operation procedure. The variable N was introduced to decouple the values of u and d from the values of Q_{max} and q_{in}. Therefore, it needs to be an arbitrary large number, which must be larger than any possible value for q_{in}.

$$\sum_{r \in R_U} \frac{c_{r_i}}{R_U} > CPU_{min} - d \tag{7.4}$$

$$q_{in} > Q_{min} - d \cdot N \tag{7.5}$$

(7.4) and (7.5) represent the constraints for downscaling procedures. These constraints work in a similar manner as those for the upscaling operations, but consider the lower thresholds Q_{min} for the incoming queue and CPU_{min} for the average CPU usage.

$$r_{i_{BTU}} = \begin{cases} \frac{ld_{r_i} \ \%\ \text{BTU}}{\text{BTU}} & \text{, if } r_i = 1 \\ 0 & \text{, else} \end{cases} \tag{7.6}$$

(7.6) defines the remaining usage duration for a specific Operator Instance, while respecting the BTU. The result of this constraint is the remaining and already paid usage leasing duration in minutes, while ld_{r_i} represents the time for which the specific Operator Instance is already running.

$$\sum_{r \in R} r_i \geq 1 \tag{7.7}$$

(7.7) ensures that there is at least one Operator Instance running for a specific operator.

$$r_i, u, d \in \{0, 1\}; 0 \leq c_{r_i}, r_{i_{BTU}}, CPU_{max}, CPU_{min} \leq 1 \tag{7.8}$$

$$R, R_U, ld_{r_i}, Q_{max}, Q_{min} \in \mathbb{N}_0 \tag{7.9}$$

(7.8) and (7.9) define the possible values for the variables.

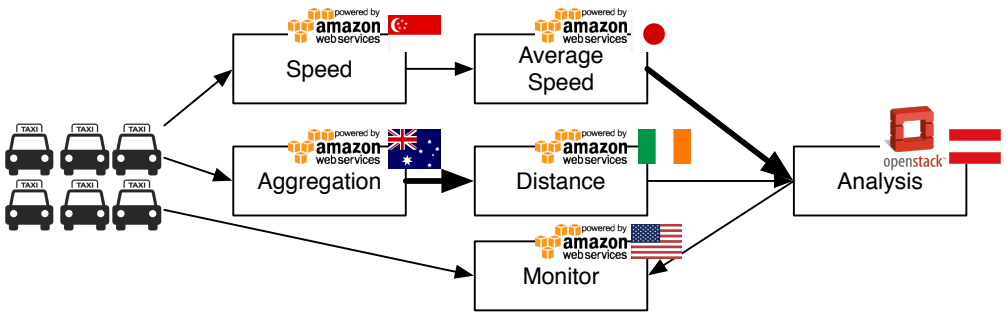


Figure 7.1: Evaluation Scenario

7.3 Evaluation

7.3.1 Evaluation Scenario

In the following paragraphs, we provide a motivational SPA from the transportation domain.

In this scenario, the provider of a worldwide operating taxi fleet wants to analyze the rides of its taxis to optimize the operational planning in real-time. The taxi provider maintains several taxi fleets in different cities across the world, whereas the operational and analytics center is located in Europe.

To realize a recommendation system for increasing the taxi usage rate as depicted in Figure 7.1, it is required to process the constant stream of location information from the taxis to extract metrics such as the *average speed* or *distance* of each single ride. These metrics are then *analyzed* to obtain optimization measures. However, before this can happen, several preprocessing activities are needed, e.g., the *speed* calculation between two geographic locations and an *aggregation* of all recorded locations for a single ride to derive the distance. Each processing step, i.e., an operator, is depicted as a single entity in Figure 7.1 whereas the *monitor* operator is only used to calculate the processing duration for a single data item.

Figure 7.1 also considers two different types of data flow connections among operators, which are depicted by fine respectively bold arrows. A fine arrow represents operations, where the operator emits the same volume of data as it receives. Bold arrows represent aggregation operations, which aggregate the streaming data over a given time window and emit fewer data after processing compared to the incoming data.

Since the taxis emit a new location information every second, it is necessary to process the data as close as possible next to the data providers, i.e., taxis, to reduce the data transfer duration between the data sources and the stream processing operators. Furthermore, there are also other limitations for the deployment of the operators, e.g., the Analysis operator must only be deployed within a private cloud on the premises of the taxi operator to protect the intellectual property of the algorithm.

For the concrete implementation we consider the following operators:

Speed: This operator calculates the movement speed between the last known location and the current location and forwards the speed information to the *Average Speed* operator.

Average Speed: This operator aggregates all individual speed information and calculates the average speed. This operator forwards only one item for each ride to the *Analysis* operator.

Aggregation: This operator aggregates all location information and forwards them as a single item to the *Distance* operator.

Distance: This operator calculates the total distance of the ride, based on the individual location information and forwards the distance information to the *Analysis* operator.

Analysis: The Analysis operator receives the average speed as well as the distance and compiles a report to be forwarded to the *Monitor*.

Monitor: The *Monitor* receives all individual location information as well as the report from the *Analysis* operator and logs the arrival times of these items. This operator then calculates the total time, which is required to create the report for a specific ride, after the last location information was submitted to the SPA.

7.3.2 Evaluation Configuration

Streaming Data

For our evaluation, we selected 75 rides from the T-drive trajectory data sample [124], which provides GPS-based trajectories within Beijing. Each of these rides consists of multiple location recordings with a timestamp and a unique identifier for each ride. This allows us to replay the rides over a given time span to create a data stream. The streaming data exposes changing data rates, since the rides were recorded across the course of a week and require a reconfiguration of processing capabilities of the SPE to process the data in near real-time. For our evaluation, we replayed the data according to the actual timely sequence of events within 110 minutes for each evaluation run. The streaming data is provided to the Speed, Aggregation, and Monitor operators at the same time and each of these operators processes the data according to their internal implementation.

Thresholds

For our evaluation, we chose the following concrete values for the elastic resource provisioning strategy: First, we enabled the VISP Runtime to allocate a maximum amount (R) of 50 VMs for each operator, whereas the BTU for each VM is 60 minutes based on the pricing model of Amazon EC2 [105]. The scaling thresholds for the incoming queue are set to 5 for scaling down (Q_{min}) and 150 for scaling up (Q_{max}), respectively 90% (CPU_{max}) and 10% (CPU_{min}) for the thresholds for the average CPU usage of the Operator Instances.

Baselines

To evaluate the elasticity aspects of the VISP Runtimes, we selected two baselines. These two baselines represent an under-provisioning as well as an over-provisioning scenario with a fixed amount of Operator Instances, as listed in Table 7.1. These fixed resource baselines represent the deployment approach for current state-of-the-art SPEs. Each operator has an individual amount of Operator Instances, due to different workloads within the SPA as well as the complexity of the stream processing operator. The baseline configuration has been selected based on the minimal respectively maximal usage within the elastic scenario based on our resource provisioning strategy.

Testbed

The evaluation was carried out within Amazon EC2 as well as a private OpenStack-based cloud. The selection of the Amazon EC2 regions is based on the evaluation scenario, as one can see in Figure 7.1 indicated by the flags. The *Speed* and *Aggregation* operators are deployed in the Singapore and the Sydney region of Amazon EC2 respectively. Furthermore, the *Average Speed* operator is deployed in the Tokyo region, the *Distance* operator is deployed in the Ireland region, and the *Monitor* operator is hosted in the Oregon region. Finally, the *Analysis* operator is deployed on a private OpenStack-based cloud within Europe. In terms of size, we use *t2.micro* instances for the Operator Instances on Amazon EC2 as well as on the private OpenStack-based cloud.

Metrics

To assess the functionality as well as the total cost for our evaluation scenario, we define different metrics. Since the VISP Ecosystem aims not only at providing a distributed runtime environment for SPAs, but also at reducing the total operational cost by applying elastic provisioning strategies, we have assessed the *Cost for Operator Instances*. This metric aggregates the number of Operator Instances for all operators for the SPA throughout the whole evaluation run, where one Operator Instance amounts for one cost unit for each minute running. Furthermore, we assess the *Total Makespan in Seconds*, which represents the time span between the first location recorded by the Monitor operator until the last report is recorded. This allows us to assess the overall stream processing performance.

The *Average Duration for the Report Generation in Seconds* describes the duration which passes between the last location information of a single ride and the issuing of the report. This metric allows us to assess the real-time processing capabilities of the VISP Runtime. Based on the previous metric we also assess the QoS, i.e., *Total Delays*, by applying a SLA to the report generation process. We assume that the report needs to be finished within 60 seconds after the last location is recorded, i.e., the *Average Duration for the Report Generation in Seconds* has to be lower than 60. This further results in the *SLA Adherence* that describes how many delays were recorded in relation to the total amount of rides.

Table 7.1: Resource Setup - Number of VMs per Operator

	Speed	*Average Speed*	*Aggregation*	*Distance*	*Analysis*	*Monitor*
Under-provisioning	5	6	2	1	1	1
Over-provisioning	8	10	3	1	1	1

Table 7.2: Evaluation Results

	Elastic Provisioning	Over-Provisioning	Under-Provisioning
Number of Total Rides	75	75	75
Number of Location Information Items	50742	50742	50742
Cost for Operator Instances	2160.66 ($\sigma = 13.61$)	2664 ($\sigma = 0.00$)	1856 ($\sigma = 0.00$)
Total Makespan in Seconds	6653 ($\sigma = 9.60$)	6655 ($\sigma = 0.00$)	6975 ($\sigma = 1.00$)
Average Duration for the Report Generation in Seconds	77 ($\sigma = 10.69$)	35 ($\sigma = 0.00$)	355 ($\sigma = 0.57$)
Total Number of Delays	21 ($\sigma = 5.29$)	0 ($\sigma = 0.00$)	75 ($\sigma = 0.00$)
SLA Adherence in %	28.00 ($\sigma = 7.40$)	100.00 ($\sigma = 0.00$)	0.00 ($\sigma = 0.00$)

7.4 Discussion

To evaluate our approach, each provisioning scenario was executed three times over the course of two days. This was done to reduce the risk of any corruption of the results, which may occur due to different system loads as well as communication channels among the different regions in Amazon EC2 and the OpenStack-based cloud.

Table 7.2 lists the average values for all three evaluation runs alongside with the standard deviation σ. Figure 7.2 presents the number of Operator Instances across the evaluation alongside with the incoming rate of streaming data. While Figure 7.2 presents the resource usage of all operators combined, Figure 7.3 presents the resource usage as well as the load of the incoming queue for the Speed Operator Node for the elastic provisioning scenario. In both figures, the horizontal axis represents the time in minutes.

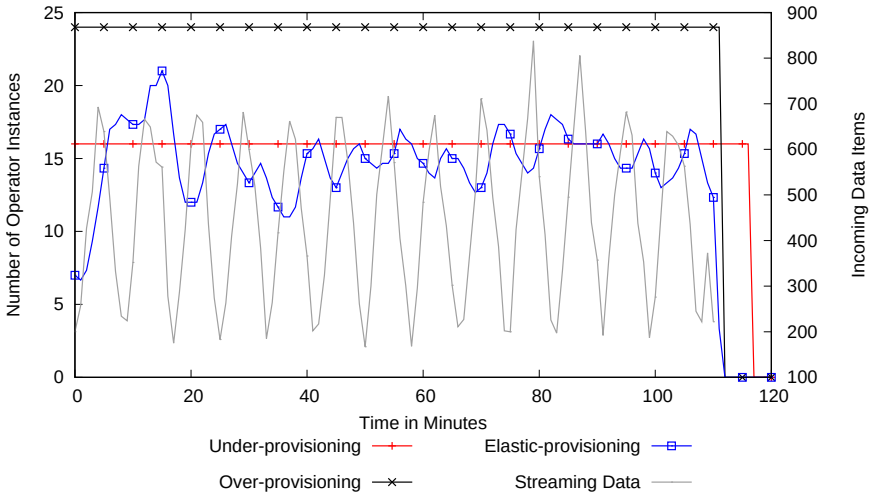


Figure 7.2: Resource Usage

The vertical axis represents the total number of Operator Instances on the left side and the amount of incoming streaming data, as well as the buffered streaming data for Figure 7.3 on the right side.

The evaluation shows that the system behavior follows a predictable outcome as required by Stonebraker [125]. This outcome is derived from the low standard deviations across the different evaluation runs, despite the fact that the evaluation was carried out in a cloud environment.

Furthermore, the evaluation also shows the relation between the amount of computational resources, i.e., Operator Instances, and the total makespan. The under-provisioning scenario exposes the longest makespan, while the total makespan for the over-provisioning and the elastic provisioning scenarios are almost the same. This additional required time can be explained by a shortage of Operator Instances in the under-provisioning scenario compared to the over-provisioning one. In comparison, the elastic provisioning scenario only requires 15% more resources than the under-provisioning scenario, while performing as fast as the over-provisioning one. This fact can be attributed to the elastic scaling mechanism, which only allocates additional Operator Instances when they are required. This can be also observed in Figure 7.2, which represents the total amount of Operator Instances over time. Figure 7.3 provides an even better representation of the relation between the streaming data rates and the amount of Operator Instances. Each time the streaming data rates rise, the amount of Operator Instances also increases to cope with the increased system load.

Since our approach applies a threshold-based scaling approach, it takes some time, i.e., 60-200 seconds, for the VISP Runtime to fully adapt to the increased system load. This delay is caused by reasoning the resource provisioning as well as the setup time of the Operator Instances, i.e., system startup of the VM.

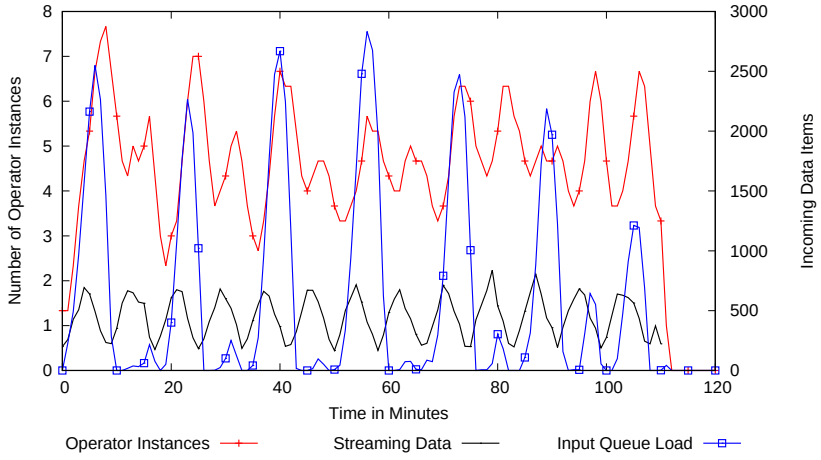


Figure 7.3: Resource Usage of the Speed Operator Node

Although the resource provisioning represents an optimization problem, its solution amounts only for a small fraction of the overall delay due to the typically small number of operators for an SPA. This can be observed in Figure 7.3 around minute 7, where the incoming queue buffers the streaming data, until our approach updates the number of available Operator Instances to the changing streaming data rates. Nevertheless, it takes our approach only twice as long to issue a report compared to the over-provisioning scenario, whereas the under-provisioning scenario requires ten times as much time. This observation is also supported by the total delay as well as the SLA adherence metric shown in Table 7.2, where we can see that the over-provisioning scenario has no issues with complying with the applied SLAs. Notably, the elastic provisioning scenario has an SLA adherence rate of 28%, although the report generation took on average only 17 seconds longer than required by the SLA. In the under-provisioning scenario, we can observe an SLA adherence of 0%, i.e., no report was issued in time, which is consistent with the other observations.

Our evaluation shows that our elasticity mechanism allows for a cost-efficient realization of an SPA, while being compliant with given SLAs.

7.5 Summary

Within this chapter, we have introduced a resource elasticity mechanism to deal with changing rates of streaming data. This allows us to operate SPAs in a cost-efficient manner due to a flexible adoption of Operator Instances at run time within the VISP Ecosystem. Furthermore, we have shown in a testbed-based evaluation that our approach yields a cost reduction compared to an over-provisioning scenario and a SLA compliance improvement compared to an under-provisioning scenario.

CHAPTER 8

Cost-efficient Data Stream Processing

In this chapter we present a novel resource provisioning approach with a specific focus on the efficient use of computational resources. This approach assigns new computational resources on demand to operators whenever an increase of the data volume requires additional processing capabilities. Besides the on-demand upscaling capabilities, our approach also optimizes the releasing of computational resources whenever suitable and economically feasible. In contrast to the rather coarse granular resource allocation approach presented in the previous chapter, this approach applies a more fine grained one to obtain a higher resource usage which results in lower operational cost. To evaluate our approach, we implemented this approach within the VISP Ecosystem and conduced a series of evaluations with different data volume patterns based on our motivational scenario. These evaluations show that our approach allows us to improve the SLA compliance by up to 25% and a reduction for the operational cost of up to 36% in contrast to a threshold based one.

8.1 Overview

Due to the high data volatility provided by IoT devices, it is essential that today's SPEs adopt elastic resource provisioning strategies, as has already been discussed in the previous chapter. Up to now, most elastic provisioning approaches only consider VMs as the smallest entity for leasing and releasing of computational resources. This approach is feasible for private clouds, where the main objective of resource provisioning is resource-efficiency without considering any billing aspects or BTUs. To address this shortcoming, this chapter considers an additional resource abstraction layer on top of VMs, to allow for more fine-grained elastic provisioning strategies with the goal to ensure 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 operators [126]. The containerization provides several advantages regarding deployment and management of computational resources. Besides the smaller granularity compared to VMs, containerized operators also allow for a faster adaption of the SPA on already running computational resources [127]. An additional layer of containers also enables reusing already paid computational resources, i.e., resources can be utilized for the full BTU [127].

Today, frameworks like Apache Mesos [128], Apache YARN [129], Kubernetes [130] or Docker Swarm [101] provide the functionality to deploy containerized applications on computational resources. These frameworks rely on simple principles like random deployment, bin-packing, or equal distribution to deploy containers across multiple hosts. Although these approaches work well for most use cases, the resource usage for the underlying VMs in terms of their BTUs can be improved as we are going to show in the remainder of this chapter.

Therefore, we propose an elastic resource provisioning approach which ensures an SLA-compliant enactment of SPAs while maximizing the resource usage of computational resources and thus minimizing the operational cost, i.e., cost for computational resources and penalty cost for delayed processing. 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 QoS.

The remainder of this chapter is structured as follows: First, we discuss the enactment scenario for the SPA given in the motivational scenario and derive several requirements for an optimization approach in Section 8.2. Based on these requirements we then provide the problem definition for the optimization problem in Section 8.3, which leads to our optimization approach presented in Section 8.4. In Section 8.5, we describe our evaluation setup and in Section 8.6, we present the evaluation results and their discussion before we summarize the chapter in Section 8.7.

8.2 Enactment Scenario and Requirements

8.2.1 Enactment Scenario

For the enactment scenario in this chapter, we consider a deployment of the motivational SPA (see Chapter 4) in one geographic location. During the enactment, the operators need to deal with streaming data from a varying amount of manufacturing machines, as shown in Figure 8.1 at the bottom. This varying data volume requires the SPA to adapt its processing capabilities on-demand, i.e., the number of Operator Instances for specific operators, which are hosted on an arbitrary amount of hosts, e.g., H1 – H4 in Figure 8.1, to comply with the SLAs. The SPE aims at minimizing the needed number of hosts by using an optimal deployment, since each host amounts for additional cost.

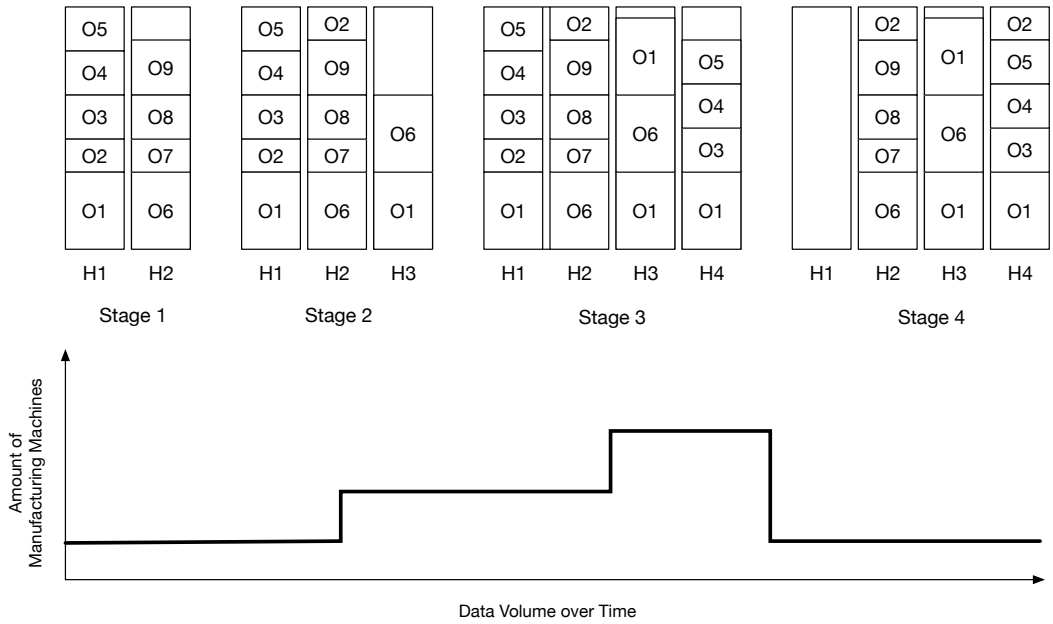


Figure 8.1: Enactment Scenario

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 H1 and H2. Since the data volume increases after some time, the SPA needs to update the processing capabilities by deploying replicas of the operators 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. In Stage 3, the data volume increases again, the SPE needs to create further Operator Instances to comply with the SLAs. Although the second replication of operator 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, H1 and H2 meet the end of their BTUs. Therefore, the SPE evaluates whether some of the replicated operators can be removed again without violating the SLAs. Due to the decreasing data volume after Stage 3, the system can remove (O1, O3, O4, and O5) or migrate (O2) some of the Operator Instances to other hosts. This leads to the situation that no Operator Instances are running on host H1 at the end of its BTU and the SPE can release H1, while host H2 needs to be leased for another BTU.

8.2.2 Requirements

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

SLA Compliance The first requirement is SLA compliance in terms of maximum processing duration, for data that is processed by the SPA. 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 computational 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 8.3. Because this optimization problem is NP-hard (see Section 8.3.2), 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 SPA because each reconfiguration activity has a negative performance impact on the data processing capabilities.

8.3 Problem Definition

8.3.1 System Model and Notation

The system model is used to describe the system state of the individual operators that form the SPA as well as the used computational resources. The individual operators are represented by $O = \{1, \ldots, o^{\#}\}$, where $o \in O$ represents a specific operator. Each operator 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 run time, each operator 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 $o \in O$.

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 run time. 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 is assigned a specific image, whose identifier is identical to the name of the operator.

Besides the fundamental operator attributes for instantiating operators, there is also a set of attributes which is used to ensure the SLA compliance for data processing. Each operator 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 according to the topology of the SPA. The o_{slo} value is recorded in an optimal processing scenario, where no data item needs to be queued for 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 for processing.

In addition to the current o_d, the system model also considers previous processing durations. Here, we consider for each operator 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. The attribute $o_{\#}$ represents all current instances, i.e., the sum of all instances of the operator o, and the attribute o_s represents all already executed scaling operations, both upscaling and downscaling, for a specific operator. Last, we also denote the current incoming amount of data items as DR.

8.3.2 Optimization Problem

Based on the identified requirements in Section 8.2.2, we can formulate an optimization problem as shown in Equation 8.1. 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.

$$
\begin{aligned}
\text{Min} \quad & h^{\#} \cdot h_{cost} \\
& + \frac{\sum_{h \in H} h_{cpu} - \sum_{i \in I \cap i_{type}=o} o_{cpu}}{\sum_{h \in H} h_{cpu}} \\
& + \frac{\sum_{h \in H} o_{memory} - \sum_{i \in I \cap i_{type}=o} o_{memory}}{\sum_{h \in H} h_{memory}} \\
& + \sum_{v \in V} v \cdot P
\end{aligned}
\tag{8.1}
$$

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 [131], which is known to be NP-hard.

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 A' 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 operator has the same memory requirements o_{memory}. Furthermore, we define the value of the specific operator by the amount of data items o_{queue} that are queued for a specific operator, i.e., the more items need to be processed, the higher is the value for creating an Operator Instance for this 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 queue o_{queue} for the specific operator.

Due to the fact that 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 approach which can obtain an optimal solution in polynomial time. Since this conclusion conflicts with the third requirement given in Section 8.2.2, we decided to realize a heuristic-based optimization approach, which can be solved in polynomial time.

8.4 Optimization Approach

The overall goal of our optimization approach is to minimize the cost for computational resources and maximize the usage of already leased VMs while being compliant to given SLAs. Therefore, we apply an on-demand approach with an emphasis on reducing the deployment and configuration overhead, i.e., instantiating and removing additional Operator Instances, as well as minimizing the computational 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 chapter.

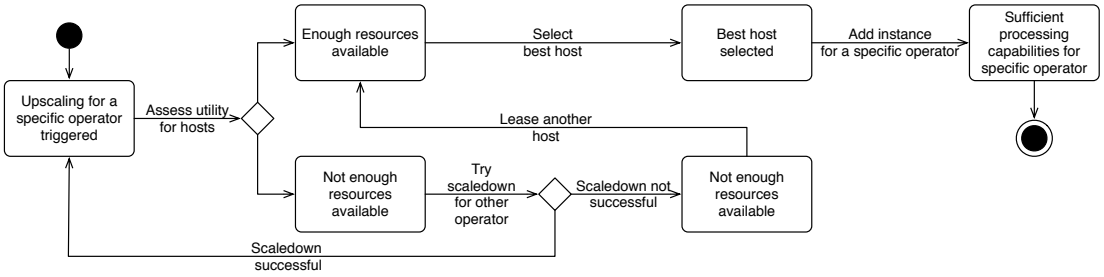


Figure 8.2: Upscaling Procedure for a Specific Operator

8.4.1 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 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 is triggered. This upscaling procedure consists of several steps, as depicted in Figure 8.2. 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 host offers enough computational resources to instantiate a new instance for the required operator, 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 Algorithm 3 and discussed in Section 8.4.3. If there is any operator that can be scaled down, an Operator Instance of this operator will be scaled down to free computational resources for the upscaling operation. When there are no operators which can be scaled down, i.e., all operators are needed for SLA-compliant data stream processing, the second possibility is applied where the SPE leases a new host. As soon as the resources are either provided by scaling down another operator or the new host is running, the SPE deploys the required Operator Instance and finishes the upscaling procedure.

8.4.2 Optimize Resource Usage

To minimize the operational cost, 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 evaluates 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.

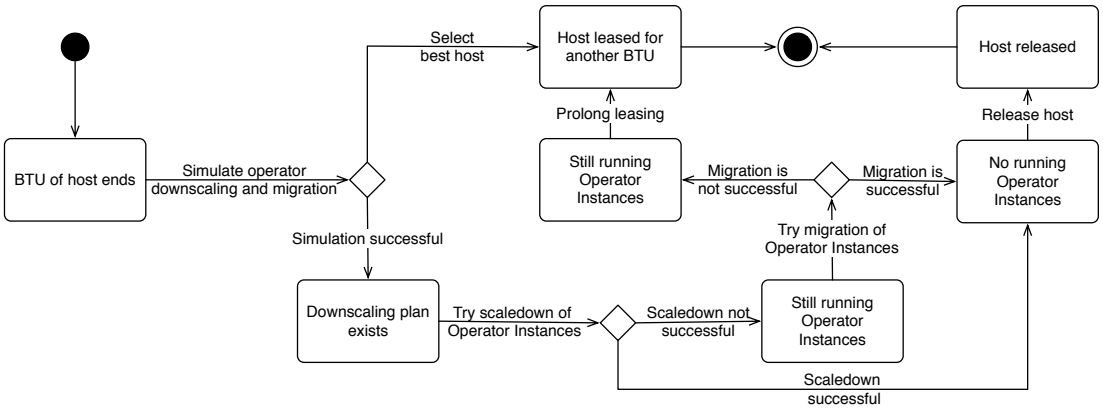


Figure 8.3: Downscaling Procedure for a Host

To release hosts, as shown in Figure 8.3, all Operator Instances running on the designated host which is targeted to be shut down, need to be either released or migrated to other hosts. This releasing procedure consists of three phases. The first phase is a simulation phase, where the optimization approach creates a downscaling plan to evaluate whether the downscaling and migration is actually feasible. Hereby, the optimization approach applies the *Operator Selection Algorithm* (Algorithm 3) for all operators, which have running instances on this host and obtain their utility value. If any of the operators has a positive utility value, all Operator Instances (up to 20% of all Operator Instances for the specific operator) running on this host are marked to be released. The 20%-threshold for the Operator Instances is in place to avoid any major reconfigurations for a single operator, since it may be the case that all Operator Instances for the operator are running on this host and after the downscaling there would be not sufficient Operator Instances left which would trigger again the upscaling procedure.

For those Operator Instances which cannot be shut down, the procedure simulates whether they can be migrated to other hosts. This simulation uses the upscaling procedure for operators, as described in Section 8.4.1. The only difference is that the host which is targeted to be shut down, is omitted as a suitable target host. If the simulation renders no feasible downscaling plan, the host is leased for another BTU and the downscaling procedure is finished.

In case there is a downscaling plan, the operators are released in the second phase and if any migration is required, the upscaling procedure for operators is triggered based on the simulation in phase three. When all Operator Instances are successfully removed (scaled down or migrated), the shut down of the host is initialized. In the unlikely event that the downscaling plan could not be executed, i.e., the Operator Instance migrations fail, the host also needs to be leased for another BTU.

8.4.3 Algorithms

To realize our BTU-based provisioning approach, we have devised three algorithms, which are discussed in detail in this section. These three algorithms realize individual tasks for the upscaling and downscaling procedures as shown in Figure 8.2 and Figure 8.3. Algorithm 1 ensures the SLA compliance of the individual operators on a regular basis by interpreting the monitoring information provided by the VISP Runtime. The other two algorithms are only triggered if a new Operator Instance needs to be started or when there is a shortage of free computational resources. These two algorithms analyze the SLA compliance and resource usage on demand at specific points in time and identify the most suitable host for upscaling (Algorithm 2) or potential operators, which can be scaled down (Algorithm 3). Although these algorithms do not represent the core functionality of the resource provisioning approach, they are still essential to identify required upscaling operations and choose the optimal degree of parallelism per operator whereas the overall cost-reduction and reconfiguration is represented by the downscaling procedure shown in Figure 8.3. The remainder of this section discusses the structure and rationale of these three algorithms in detail.

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 for each operator o and either returns 0, if the current stream processing capabilities are enough to comply with the SLAs, or 1 if the operator 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 to be scaled up (Line 10).

Before we trigger the upscaling operation, we apply an additional check if the upscaling operation is actually required. The SPA may retrieve short-term data volume peaks, e.g., due to short network disruptions. These peaks would not require any additional computational resources, because they would be dealt with after a short time with the already available processing capabilities. Nevertheless, the upscaling algorithm would trigger the upscaling procedure, because it would detect the processing delay. 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 specific $scalingThreshold$ (Lines 16 – 18).

Algorithm 2, i.e., the *Host Selection Algorithm*, is used to rank all currently leased hosts according to their suitability to host a new Operator Instance of a particular operator o. Therefore, the algorithm evaluates for each host h whether a new instance of the required operator 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.

Algorithm 1 Upscaling Algorithm

1: **function** upTrigger(o, N)
2: **if** $o_d > o_{slo}$ **then**
3: upscaling = 1
4: **end if**
5: observationMean = $\frac{1}{N} * \sum_{i=1}^{N} i$
6: durationMean = $\frac{1}{N} * \sum_{i=1}^{N} o_{d_i}$
7: $\beta = \frac{\sum_{i=1}^{N}(i-\text{observationMean})*(o_{d_i}*\text{durationMean})}{\sum_{i=1}^{N}(i-\text{observationMean})^2}$
8: $\alpha = \text{durationMean} - \beta * \text{observationMean}$
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 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 also use currently unused resources of the hosts based on the first-come, first-serve 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 volume 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. Here, we prefer those hosts which have already a cached copy of the Operator Image.

While such Operator Images may be rather small for SPEs which operate on a process or thread level, like Apache Storm, these images can reach up to 100 MB for containerized operators. Therefore, a download from the external repository of these Operator Images requires some time. In order to distinguish hosts which have a cached copy of the Operator Image from those hosts that do not have a cached copy of the Operator Image, we multiply the *suitability* with a constant factor CF to create two different groups of hosts for the overall selection (Lines 10 – 12). For this constant factor, we recommend to use the value 0.01 which was also used in the remainder of our work. The value 0.01 was chosen to clearly distinguish these two groups, since the actual suitability values are always in the range of 0 to 1 based on the structure of the algorithm. Each of these groups maintains their resource-based ordering, but we prioritize those hosts that provide a faster startup time due to the cached image, i.e., the group with lower values. 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**

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

First, the algorithm ensures that there is at least one Operator Instance for the given operator (Lines 2 – 4). Second, the function considers the amount of all currently running instances for the specific operator and normalizes it to obtain a value between 0 and 1 (Line 5). This normalization is carried out based on the maximum respectively minimum amount of instances for all operators.

This value represents the aspect that it is better to scale down an operator with numerous Operator Instances because the scale down operation removes a smaller percentage of processing power compared to an operator with fewer Operator Instances.

Furthermore, we also consider the SLA compliance of the particular operator. Here, we consider the actual compliance for the processing duration and multiply it with the penalty cost as a weighting factor (Line 6). Since the penalty cost for the violation of a single data item is typically lower than 1, we add 1 to the penalty cost P. Whenever the processing duration o_d takes longer than the SLO o_{slo}, the delay value will be less than 1, 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 all scaling operations (Line 7). Here, we penalize previous scaling operations because we want to avoid any oscillating effects, 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, as soon as no data items are waiting in the incoming queue o_{queue} (Lines 8 – 12). For QL we recommend the use of the value 100 to clearly indicate that the operator can be scaled down, regardless of the other values which are in the range of 0 – 1 for the *instances* and *scalings* value or significantly lower than 100 for the *delay* value. This value was selected based on a number of preliminary experiments prior to the actual evaluation where the data processing never took longer than 50 times the given SLO o_{slo}.

Algorithm 3 Operator Selection Algorithm

```
1:  function DOWN(o)
2:      if o_# < 2 then
3:          return -1
4:      end if
5:      instances = (o_# − min(o_# ∈ O)) / (max(o_# ∈ O) − min(o_# ∈ O))
6:      delay = o_d / o_slo * (1 + P)
7:      scalings = o_s / Σ_{o_s ∈ O} o_s
8:      if o_queue < 1 then
9:          queueLoad = QL
10:     else
11:         queueLoad = 0
12:     end if
13:     return 1 + W1 * instances + W2 * queueLoad − W3 * delay − W4 * scalings
14: end function
```

Finally, we join the distinct aspects to obtain the overall utility value. While the number of instances and *queueLoad* represent a positive aspect to scale down an operator, all other aspects discourage a scaling operation. The instances and scalings value are normalized between 0 and 1 whereas the scalings value can exceed 1 if the data processing is delayed. Therefore, we introduce optional weights $W1, W2, W3$, and $W4$ for the different aspects. However, the default value for each of these weights is 1 to treat all aspects with the same significance. 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).

8.5 Evaluation

8.5.1 Evaluation Setup

For our evaluation, we revisit the motivational scenario (see Chapter 4) and discuss the concrete implementation of this SPA.

Sensors

First, we are going to discuss the sensors which emit the data items for our SPA. In this SPA, we consider three different sensors, as listed in Table 8.1. Each of these sensors generates a data item with a particular structure, which can be only processed by a dedicated operator, e.g., O1 for sensor S1. Due to the different structure, the size of the data items also differs. The first and the last sensor (S1 and S3) encode the information in plain text. This results in rather small data items with a size of 90 to 95 Bytes. The second sensor encodes the information with an image and is therefore much larger, i.e., around 12500 Bytes.

Stream Processing Operators

The second important implementation aspect for the SPA are the operators. Each of these operators performs a specific task with specific resource requirements and specific processing durations. Table 8.2 lists all operators which are used for the evaluation and each operator is assigned a number of different performance as well as resource metrics. The resource metrics represent mean values across several SPA executions.

Table 8.1: Sensors

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

Table 8.2: Stream Processing Operators

	Processing Duration (ms)	CPU Shares	Memory (MB)	Storage (MB)	State	Outgoing Ratio
O1	600	131	524	68	✓	50:1
O2	600	65	440	68	✓	100:1
O3	1500	660	452	89		1:3
O4	750	100	430	68		1:1
O5	750	83	502	68	✓	1:1
O6	750	77	527	68	✓	1:1
O7	700	46	464	68	✓	3:1
O8	1300	47	452	70	✓	300:1
O9	500	74	466	68		1:0

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 values in the table are 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 SPA is processing constant data volume in a fixed over-provisioning scenario to avoid any waiting durations for the recordings.

For the storage operator, we have three different sizes. Due to the fact that the majority of the processing operators only implement processing logic, the size of the images is the same. The only two exceptions are the Generate Report (O8) image, which also contains a PDF generation library and the Parse and Distribute Data (O3) Operator Image, which also contains the Tesseract binary [132], 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. The outgoing ratio describes whether a particular operator 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., O3 distributes the production information to three other operators.

For our scenario, we have implemented nine different operators [108] as Spring Boot [133] applications, which are discussed in detail in the remainder of this section.

Filter Availability (O1) 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 warning for each new defect notification and filters all other data items.

Monitor Temperature (O2) The Monitor Temperature operator filters all temperatures below a predefined threshold and issues a notification to the human operator for each new temperature violation.

Parse and Distribute Data (O3) The Parse and Distribute Data operator 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 [132] to parse the image and then the Spring Boot-based application [133] forwards the machine-readable production data to the downstream operators.

Calculate Performance (O4) The Calculate Performance operator 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 8.2.

$$performance = \frac{producedItems \cdot idealProductionTime}{reportingCycle} \tag{8.2}$$

Calculate Availability (O5) The Calculate Availability operator 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 8.3.

$$availability = \frac{totalTime - scheduledDowntime - unscheduledDowntime}{totalTime} \tag{8.3}$$

Calculate Quality (O6) The Calculate Quality operator 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 8.4.

$$quality = \frac{totalProducedGoods - totalDefectiveGoods}{totalProducedGoods} \tag{8.4}$$

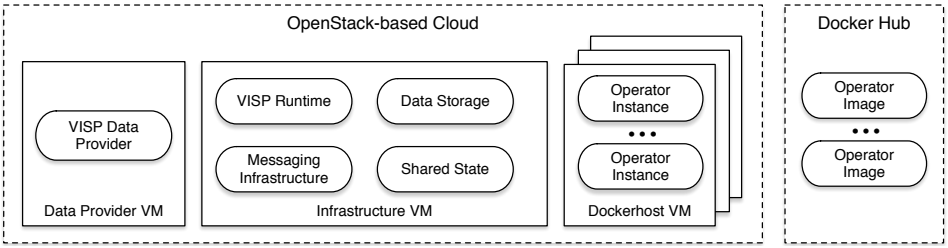


Figure 8.4: Deployment for the Evaluation Scenario

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 8.5.

$$oee = availability \cdot performance \cdot quality \tag{8.5}$$

Generate Report (O8) The Generate Report operator aggregates multiple OEE values and generates a PDF report. This report is then forwarded to the user for further manual inspection.

Inform User (O9) The Inform User operator forwards the notifications to a human user. In our evaluation scenario, this operator implementation only serves as a monitoring endpoint and all incoming data items are discarded at this operator.

8.5.2 Evaluation Deployment

For our evaluation, we make use of the VISP Testbed [4], 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. Furthermore, the Data Provider also allows defining different arrival patterns (see Section 8.5.5) to evaluate the adaptation possibilities of the VISP Runtime, in particular of its scaling mechanisms. The evaluation runs are carried out in a private cloud running OpenStack [106], whereas the components are deployed on different VMs, as depicted in Figure 8.4. 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 [110], the Shared State, i.e., Redis [104] and the Data Storage, i.e., a MySQL [134] 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 run time. These Dockerhost VMs are used to run the Operator Instances, which take care of the actual data processing. The Operator Images, which are required to run the Operator Instances, are hosted on an external service, i.e., Dockerhub [135]. Finally, the Data Provider VM is in charge of simulating the data streams from the sensors, as described in Section 8.5.1.

8.5.3 Evaluation Configuration

For the *scalingThreshold* in Algorithm 1, we use the value 50. This value was selected to be high enough to allow for minimal hardware disturbances, e.g., moving data from memory to the hard drive, but low enough to react to small changes of the data volume. The concrete value was identified on a number of preliminary experiments, evaluating different thresholds in the range of 10 to 1000 items, whereas the threshold 50 was identified as the most suitable value for our purpose. Regarding the individual weights $W1 - W4$ used in Algorithm 3, we use the default value of 1 to evaluate the base design of our BTU-based provisioning approach without any specific emphasis on either the number of instances, scaling operations, queue load or the processing delay.

Besides the configuration aspects for Algorithms 1 and 3, there are also several other configuration aspects for the VISP Runtime. We chose a monitoring timespan of 15 seconds, i.e., the queue load and resource usage of the system is recorded every 15 seconds. The resource provisioning interval is set to 60 seconds. This interval has been selected to update the resource configuration for the SPA as well as SPE in short time intervals while ensuring enough time to download Operator Images from the external repository within one resource provisioning interval.

Regarding the BTU, we make use of three different BTU durations. The first duration is 60 minutes (BTU60), which used to be the predominant BTU on Amazon EC2 [105]. The second duration is 10 minutes (BTU10), which represented the minimal BTU for the Google Compute Engine [136] and the last duration is 30 minutes (BTU30), which has been selected to present a middle ground between the other two. Furthermore, we assume a linear pricing model for the BTUs, i.e., one leasing duration for the BTU10 model results in 1 cost, one leasing duration for the BTU30 model results in 3 cost and the leasing duration for the BTU60 model results in 6 cost. For each data item which is delayed, we accrue 0.0001 penalty cost, i.e., 10000 delayed items render the same cost as leasing a VM for 10 minutes. These penalty cost have been chosen to impose little cost for delayed processing compared to penalty cost in other domains, e.g., for business processes [137]. Also, preliminary experiments have shown that higher penalty cost, e.g., 0.001 or 0.01, would render unreasonable high penalty cost compared to the actual resource cost even for a high SLA compliance. Finally, each Dockerhost VM has the same computational resources with 4 virtual CPU cores and 7 GB RAM.

8.5.4 Baseline

To evaluate our BTU-based provisioning approach, we have selected a threshold-based baseline provisioning approach. The baseline implements a commonly used provisioning approach which already achieves very good results in terms of cost reduction against an over-provisioning scenario as shown in Chapter 7. The baseline approach considers only the amount of data items waiting on the incoming queue for processing as a scaling trigger. As soon as the variable o_{queue} exceeds an upper threshold according to Algorithm 1, the SPE triggers an upscaling operation for this operator.

The same applies when 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 a second time if o_{queue} surpasses a second upper threshold of 250 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.

8.5.5 Data Arrival Pattern

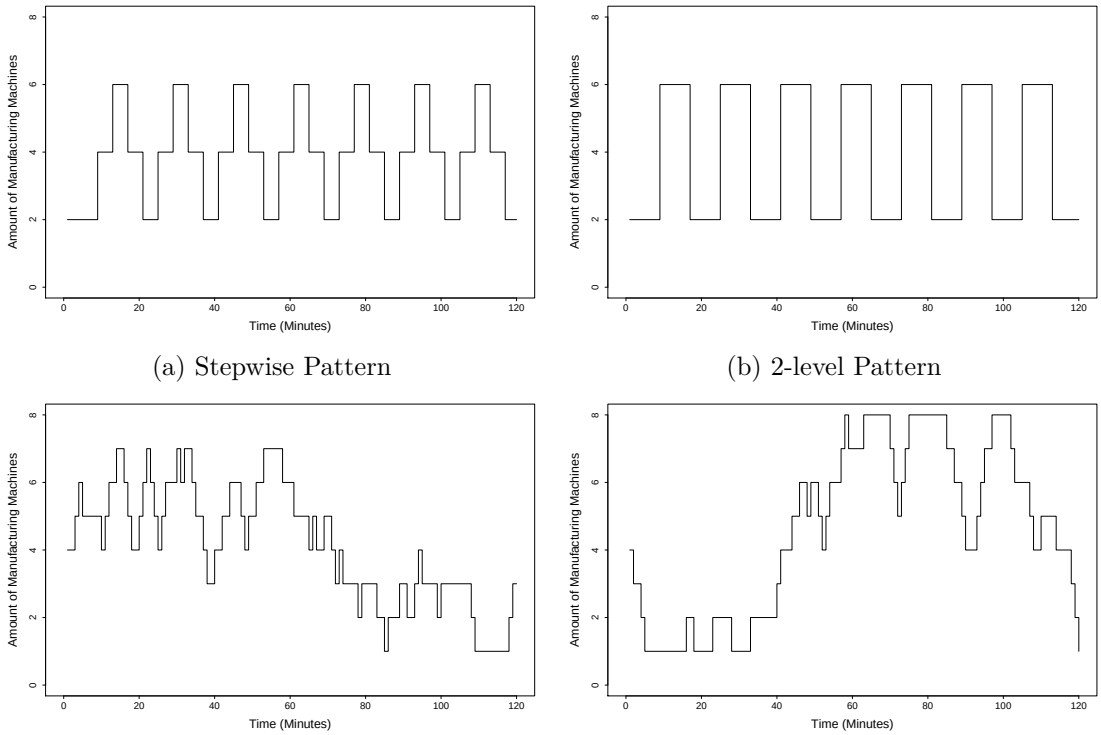
For our evaluation, we have selected four different arrival patterns which simulate different load scenarios for the SPA by submitting varying data volume to the SPA. The first arrival pattern has three different data volume levels, which are changed stepwise, so that the resulting arrival pattern could be approximated to a sinus curve, as shown in Figure 8.5a. These three different volume levels simulate different amounts of manufacturing machines ranging from two to eight machines that emit different amounts of data items, as shown in Table 8.1. To speed up the evaluation, we simulate the real-time data emissions shown in Table 8.1 every 480 milliseconds. This enables us to simulate 500 real-time minutes within only four minutes in the course of our evaluation and therefore increases the load on the SPA and as a consequence also on the SPE. In addition 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 SPA with more drastic volume changes, as shown in Figure 8.5b. 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.

The third and the fourth pattern represent random walks as defined by Equation 8.6, whereas R represents a random number between 0 and 1. This random walk is initialized with $machine = 4$ and we have selected two random walk patterns which stay between one and eight machines. The results of this random walk can be seen in in Figure 8.5c for the first random walk and in Figure 8.5d for the second one. Due to the random characteristic of the pattern generation, this pattern exhibits more changes of the data volume in short time compared to the first two data arrival patterns.

$$machine_n = \begin{cases} machine_{n-1} - 1 & R < 0.4 \\ machine_{n-1} & 0.4 \leq R \geq 0.6 \\ machine_{n-1} + 1 & R > 0.6 \end{cases} \tag{8.6}$$

All four patterns are continuously generated by the VISP Data Provider [108] throughout the whole evaluation duration of 120 minutes.



(a) Stepwise Pattern

(b) 2-level Pattern

(c) Random Walk Pattern 1

(d) Random Walk Pattern 2

Figure 8.5: Data Arrival Patterns

8.5.6 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 the SLA compliance. After each evaluation run, these metrics are extracted by the VISP Reporting Utility [108]. 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 SLA compliance of the SPA. Each 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 for a short time from under-provisioning, which results in longer processing durations. To assess the overall compliance of the processing durations, we define three different SLA compliance levels. 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 a *near real-time* requirement, 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 scaling decisions 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. The *Time To Adapt* metric states the arithmetic mean duration, which is required until the delayed processing for an operator 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.

8.6 Results and Discussion

For our evaluation we consider four different provisioning approaches. The first approach is the BTU-agnostic threshold-based approach while the other three approaches are BTU-based approaches with three different BTU configurations as discussed in Section 8.5.2. To obtain reliable numbers, we have conducted three evaluation runs for each provisioning approach and data arrival pattern, which results in 48 evaluation runs. These evaluations have been executed over the time span of four weeks on a private OpenStack-based cloud.

The discussion of our evaluation is divided in four subsections based on the four data arrival patterns. Each subsection features a table which lists the average numbers of the three evaluation runs alongside with their standard deviations. Hereby it must be noted that the evaluations for the threshold-based approach have been conducted once, which results in only one compliance scenario. The actual cost depending on the BTU have been calculated after the evaluation, depending on the actual leasing duration of the VM. Additionally, we also provide a figure which represents the resource configurations of the Operator Instances and VMs over the course of the evaluation for each data arrival pattern.

For the discussion we are going to analyze the differences between the BTU-based and the threshold-based approach in detail only for the stepwise data arrival pattern because this arrival pattern allows us to isolate specific aspects of the BTU-based approach. Nevertheless, our evaluations show that the overall trend regarding the SLA compliance and total cost is the same for all four data arrival patterns. For the other arrival patterns we only highlight specific aspects of the individual patterns and refer for all other effects to the discussion of the stepwise data arrival pattern.

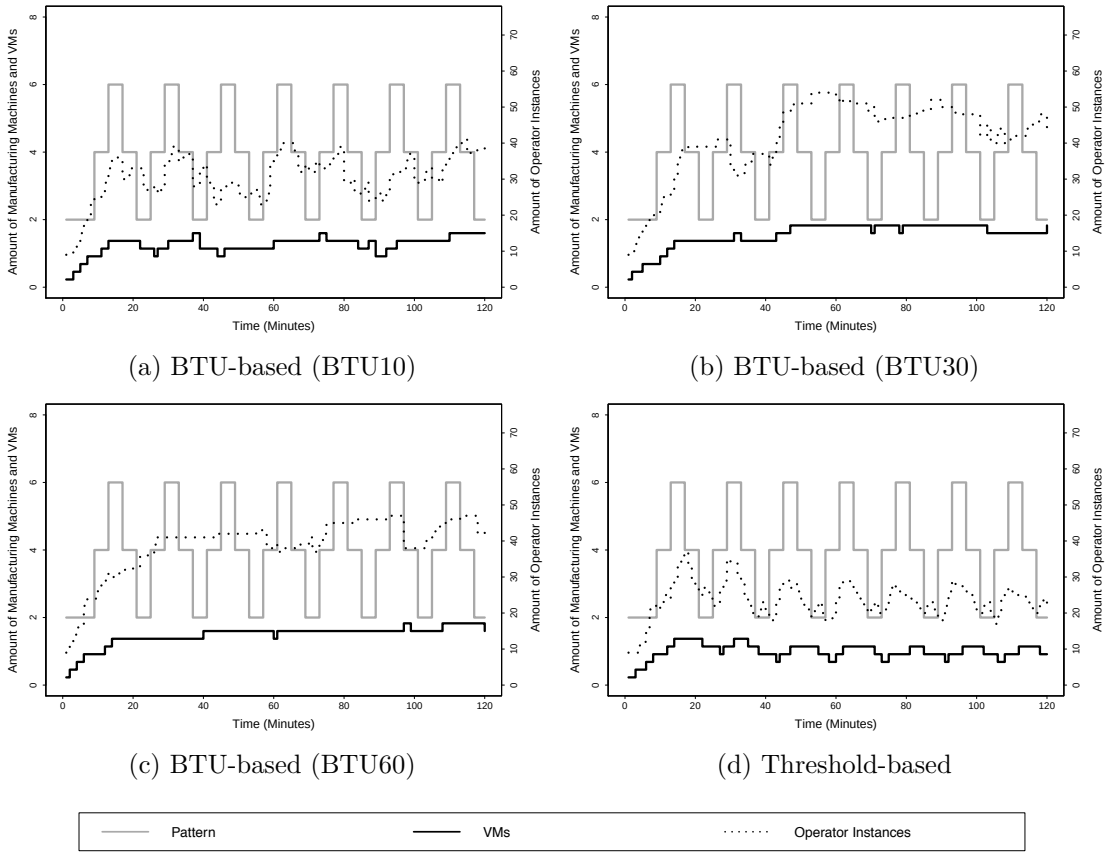
Table 8.3: Evaluation Results for Stepwise Scenario

	BTU-based			Threshold-based		
	BTU10	BTU30	BTU60	BTU10	BTU30	BTU60
Real-time Compliance	49% (σ = 1%)	52% (σ = 1 %)	53% (σ = 1%)	40% (σ = 1%)		
Near Real-time Compliance	85% (σ = 2%)	90% (σ = 1%)	93% (σ = 1%)	67% (σ = 1%)		
Relaxed Compliance	89% (σ = 1%)	93% (σ = 1%)	95% (σ = 1%)	71% (σ = 1%)		
Resource Cost	72.33 (σ = 3.79)	92.00 (σ = 1.73)	98.00 (σ = 3.84)	58.00 (σ = 1.73)	79.00 (σ = 4.58)	120.00 (σ = 6.00)
Real-time Total Cost	158.91 (σ = 0.82)	173.39 (σ = 0.68)	174.69 (σ = 4.25)	151.83 (σ = 1.95)	172.83 (σ = 4.93)	213.83 (σ = 6.45)
Near Real-time Total Cost	96.85 (σ = 0.39)	108.24 (σ = 0.50)	108.88 (σ = 3.17)	109.59 (σ = 1.77)	130.59 (σ = 4.73)	171.59 (σ = 6.35)
Relaxed Total Cost	90.96 (σ = 1.84)	103.03 (σ = 1.04)	105.41 (σ = 2.91)	102.97 (σ = 1.57)	123.97 (σ = 4.49)	164.97 (σ = 6.12)

8.6.1 Stepwise Data Arrival Pattern

For the stepwise pattern, we can see that the overall SLA compliance is higher for the BTU-based approach for all three SLA compliance scenarios as shown in Table 8.3. This compliance benefit ranges from 9% for the BTU10 configuration in the real-time compliance scenario, up to 24% in the relaxed compliance scenario for the BTU60 configuration. The SLA compliance gain can be explained due to the downscaling strategy of the BTU-based approach in contrast to the on-demand one for the threshold-based approach. The threshold-based approach only considers the amount of data items that need to be processed by each operator for the scaling decisions, which can be observed in Figure 8.6d. This figure shows that the line for the Operator Instances follows the data volume very closely with a short delay. The delay is due to the fact that the threshold-based approach can only react to the changes of the data volume. On closer inspection, one can also identify smaller increases after the downscaling phase, e.g., around minutes 40, 55 or 70. These smaller bumps indicate that the downscaling approach was too eager and the SPE has to compensate it by scaling up again. Throughout this time span, i.e., between the detection of a lack of processing capabilities and the successful upscaling for the operator, the SPA is very likely to violate the SLA compliance, especially in the real-time scenario.

The BTU-based approach does not exhibit such a strongly coupled relationship between the Operator Instances and the data volume. While the upscaling trigger is the same for both scenarios, there are clear differences in the downscaling behavior. The BTU-based approach only considers downscaling activities briefly before the end of a BTU.



(a) BTU-based (BTU10)

(b) BTU-based (BTU30)

(c) BTU-based (BTU60)

(d) Threshold-based

Figure 8.6: Stepwise Pattern

This can be seen around minutes 20 or 40 for the BTU10 scenario, around minute 30 for the BTU30 scenario and around minute 60 for the BTU60 scenario in Figures 8.6b and 8.6c. The result of this lazy downscaling strategy is a decrease of scaling activities, especially for the BTU30 and BTU60 scenario. This decrease in scaling activities results in a better SLA compliance since the SPA already has the processing capabilities for future data volume peaks as this is the case for the stepwise data arrival pattern. This results in high SLA compliance values of over 90% for the BTU30 and BTU60 scenarios. It also needs to be noted that the lack of active downscaling activities does not increase the cost for computational resources since these resources have already been paid at the beginning of their BTU.

The BTU-based downscaling operations are often triggered at suitable times, e.g., around minutes 20 and 38 for the BTU10 configuration or minute 70 for the BTU30 configuration, where the downscaling activities do not impact the SLA compliance. Nevertheless, there are also points in time when the BTU of a VM coincides with a peak of the data volume, e.g., at minute 30 for the BTU30 configuration.

In these situations, the BTU-based approach will initialize the downscaling procedure to release a VM shortly before the end of its BTU. In the specific case around minute 30 for the BTU30 scenario, the downscaling procedure is successful because monitoring does not report any delays for processing based on Algorithm 3 and the VM is triggered to be shut down. But in the next reasoning cycle, the SPE realizes the lack of processing capabilities and leases another VM to compensate the resource requirements. Although these non-efficient scaling operations result in a measurable overhead as well as an SLA compliance reduction, the BTU-based approach still achieves a better SLA compliance than the threshold-based approach.

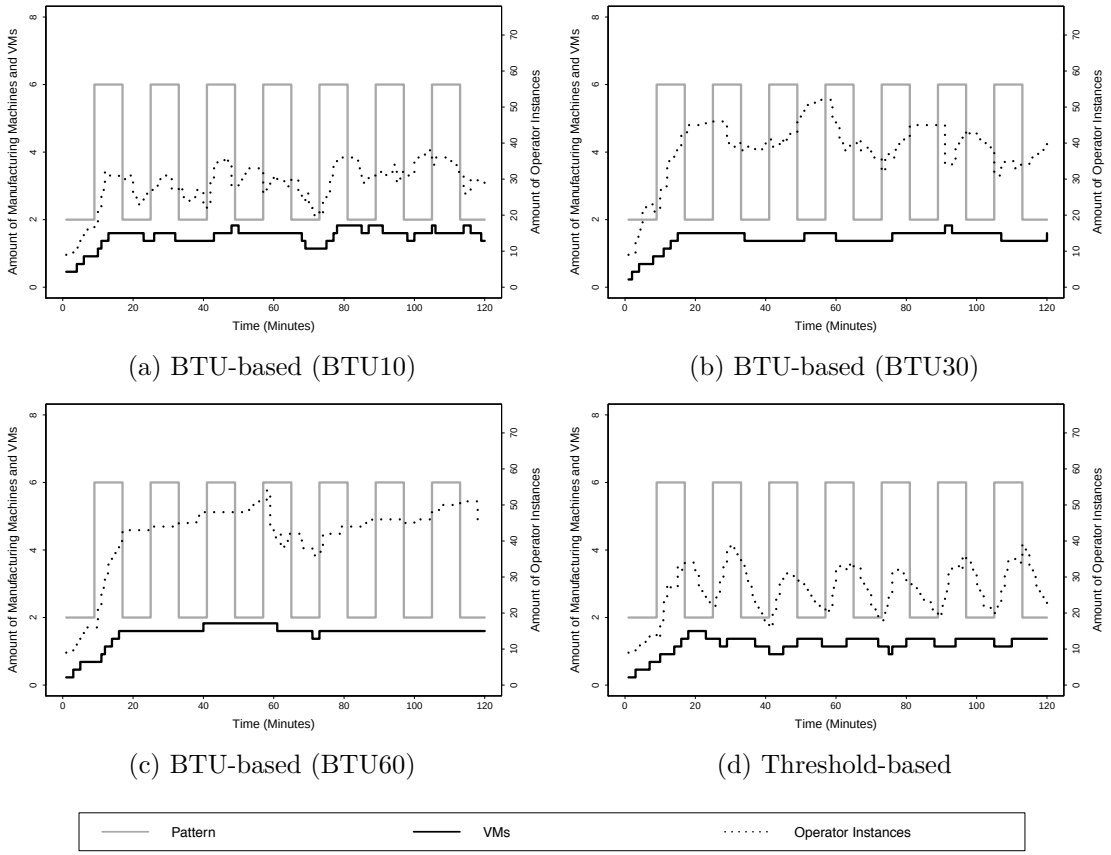
Furthermore, it can be seen that the amount of scaling activities for the Operator Instances is inverse to the length of the BTU. For the BTU10 configuration, it can be observed in Figure 8.6a that the level of scaling activities is similar to those of the threshold-based scenario. This results in a rather low SLA compliance, but for the BTU30 and especially the BTU60 there are less downscaling events, i.e., BTU ends, which reduces the need to scale up again to comply with future data volume peaks.

Besides the SLA compliance, we also consider the operational cost for data processing. These cost are composed of the resource cost, i.e., the cost for leasing VMs and the penalty cost, which accrue for delayed data processing. In Table 8.3, it can be seen that the resource cost for the BTU10 and BTU30 configuration are higher than the ones for the threshold-based ones. These higher cost can be explained due to the defensive approach of releasing VMs for the BTU-based approach, which often results in leasing the VM for another BTU based on Algorithm 3. For the BTU60 configuration, the resource cost are around 19% lower than those for the threshold-based configuration. Although the BTU60 configuration uses more computational resources, as shown in Figure 8.6c, the overall cost are lower, because the threshold-based approach releases the VMs often prematurely before the end of their BTU, which results in a waste of already paid resources.

When we consider only the resource cost, we can see that the BTU-based approach only outperforms the threshold-based approach for the BTU60 configuration. Nevertheless, this situation changes when we also consider the penalty cost, i.e., 1 cost for 10000 delayed items. After adding the penalty cost and analyzing the total cost for the different compliance scenarios, we can see that only the real-time total cost for the BTU10 configuration is higher than the threshold-based approach. All other scenarios result in slightly less cost for the BTU30 configuration in the real-time scenario and up to a 36% cost-reduction for the near real-time one for the BTU60 configuration.

8.6.2 2-level Data Arrival Pattern

The 2-level data arrival pattern exhibits the same trend for the SLA compliance and cost as the stepwise data arrival pattern as shown in Table 8.4. When we analyze the Figures 8.7a, 8.7b, 8.7c and 8.7d, we can also see a similar scaling behavior compared to the stepwise data arrival pattern. Nevertheless, there is one notable effect for the BTU60 configuration in Figure 8.7c.



(a) BTU-based (BTU10)

(b) BTU-based (BTU30)

(c) BTU-based (BTU60)

(d) Threshold-based

Figure 8.7: 2-level Pattern

The BTU-based provisioning approach tends to start more and more Operator Instances throughout the evaluation run. We can see that after minute 20, when the SPE has enough processing capabilities, the upscaling trigger requests new Operator Instances from time to time to cope with the data volume. These upscaling operations are most likely due to minor external events, e.g., short network delays induced by other applications running on the same physical hardware, which cause the SPA to require additional processing capabilities. The result of this slow increase of Operator Instances over time is that the SPA is likely to have more processing capabilities than it actually needs. Nevertheless, at the end of the BTU of a VM, the necessity of these processing capabilities is evaluated, and for example in the BTU60 configuration, the Operator Instances are cut back around minute 60. After a short recalibration phase between minutes 65 and 75, the SPE follows the same pattern again until the resources are cut back again around minute 120. This mechanism allows the SPE to use the already leased resources, i.e., no additional VMs are leased from minute 80 until 120, to achieve a high resource usage.

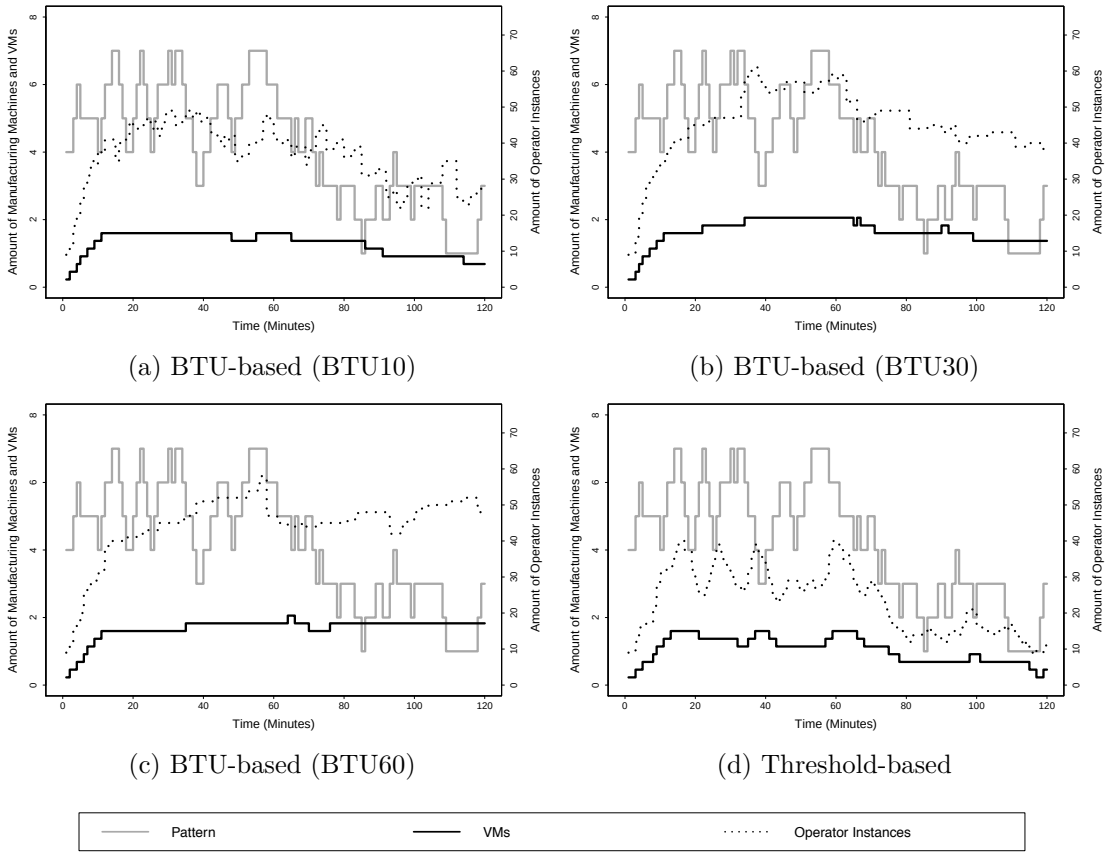
Table 8.4: Evaluation Results for 2-level Scenario

	BTU-based			Threshold-based		
	BTU10	BTU30	BTU60	BTU10	BTU30	BTU60
Real-time Compliance	48% ($\sigma = 1\%$)	50% ($\sigma = 1\%$)	55% ($\sigma = 1\%$)	40% ($\sigma = 2\%$)		
Near Real-Time Compliance	84% ($\sigma = 2\%$)	88% ($\sigma = 0\%$)	93% ($\sigma = 2\%$)	68% ($\sigma = 2\%$)		
Relaxed Compliance	88% ($\sigma = 2\%$)	91% ($\sigma = 0\%$)	95% ($\sigma = 1\%$)	72% ($\sigma = 2\%$)		
Resource Cost	82.33 ($\sigma = 5.13$)	96.00 ($\sigma = 7.94$)	104.00 ($\sigma = 6.93$)	66.00 ($\sigma = 0.00$)	86.00 ($\sigma = 1.73$)	122.00 ($\sigma = 3.46$)
Real-time Total Cost	169.17 ($\sigma = 6.83$)	177.62 ($\sigma = 6.82$)	175.90 ($\sigma = 4.77$)	157.88 ($\sigma = 2.64$)	177.88 ($\sigma = 4.17$)	213.88 ($\sigma = 4.16$)
Near Real-time Total Cost	108.35 ($\sigma = 6.74$)	155.43 ($\sigma = 8.18$)	114.50 ($\sigma = 6.28$)	114.62 ($\sigma = 3.21$)	134.62 ($\sigma = 4.19$)	170.62 ($\sigma = 2.94$)
Relaxed Total Cost	102.37 ($\sigma = 6.74$)	110.62 ($\sigma = 8.18$)	111.73 ($\sigma = 6.28$)	108.83 ($\sigma = 2.40$)	128.83 ($\sigma = 3.37$)	164.83 ($\sigma = 2.59$)

8.6.3 Random Walk 1 Data Arrival Pattern

Based on the numbers in Table 8.5, we can see that the random walk 1 data arrival pattern follows the same trend for the SLA compliance as well as total cost as the stepwise data arrival pattern. At closer inspection we can see that the SLA compliance is very similar with a difference of less than 3%. This aspect shows that the baseline as well as the BTU-based provisioning approach have similar characteristics for the simple data arrival pattern, like the stepwise or 2-level one, as well as random ones.

Based on Figures 8.8a, 8.8b, 8.8c and 8.8d, we can identify one notable difference between the BTU-based and the threshold-based resource provisioning approach. While the Operator Instance curve and the data volume curve are well-aligned for the threshold-based and the BTU10 configuration, we can identify a clear gap for the BTU30 in Figure 8.8b and especially for the BTU60 configuration (Figure 8.8c). For the latter two configurations, the Operator Instance curve remains high although the data volume decreases over time. This behavior can be explained due to the optimal resource usage of the already paid resources, which enables the BTU30 and BTU60 configuration to keep the running Operator Instances without any additional cost. Although this behavior may seem to be a waste of resources at first sight due to the deviation of the actual data volume and the Operator Instances, it becomes beneficial for the SPA in terms of SLA compliance when the volume rises again, e.g., around minutes 85 or 120.



(a) BTU-based (BTU10)

(b) BTU-based (BTU30)

(c) BTU-based (BTU60)

(d) Threshold-based

Figure 8.8: Randomwalk Pattern 1

8.6.4 Random Walk 2 Data Arrival Pattern

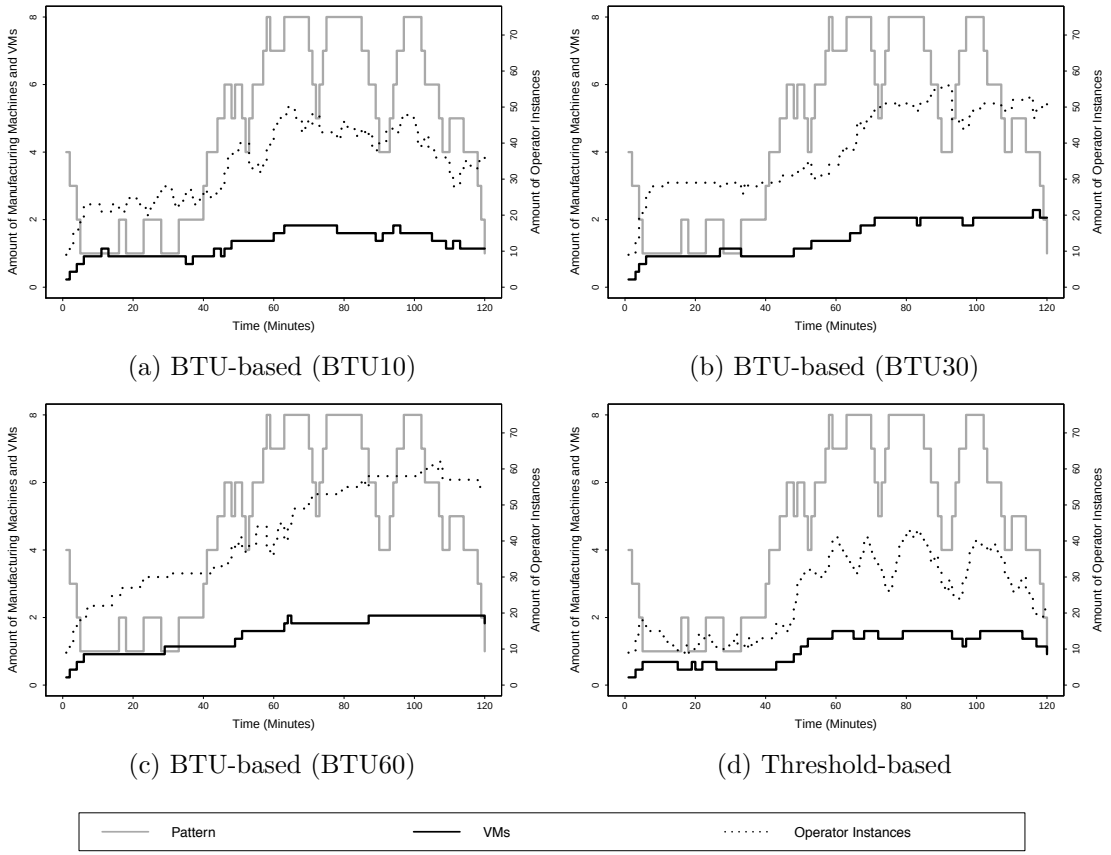
The numerical results in terms of the SLA compliance and total cost follow the same trends as for the stepwise data arrival pattern, based on the numbers in Table 8.6. For this data arrival pattern also only the BTU10 configuration requires more cost than the threshold-based baseline for the real-time scenario. All other configurations and scenarios result in lower cost than the baseline. When we analyze the graphical representation of Figures 8.9a, 8.9b, 8.9c and 8.9d for the random walk 2, the most prominent difference in contrast to the random walk 1 is the even better alignment of the Operator Instance and data volume curves. This is due to the fact that the data volume is rising after minute 40, and the already paid resources can be actively used for data processing instead of only serving as free backup processing capabilities. Furthermore, it can be seen that the BTU-based approach requires less scaling activities between minute 60 and 120 in contrast to the threshold-based approach in Figure 8.9d. This is again due to the lazy release characteristics of the BTU-based approach, which result in a higher SLA compliance compared to the threshold-based approach.

Table 8.5: Evaluation Results for Random Walk 1

	BTU-based			Threshold-based		
	BTU10	BTU30	BTU60	BTU10	BTU30	BTU60
Real-time Compliance	49% ($\sigma = 1\%$)	52% ($\sigma = 2\%$)	54% ($\sigma = 0\%$)	39% ($\sigma = 0\%$)		
Near Real-time Compliance	85% ($\sigma = 2\%$)	90% ($\sigma = 3\%$)	93% ($\sigma = 0\%$)	66% ($\sigma = 1\%$)		
Relaxed Compliance	89% ($\sigma = 2\%$)	93% ($\sigma = 2\%$)	95% ($\sigma = 1\%$)	71% ($\sigma = 1\%$)		
Resource Cost	69.33 ($\sigma = 3.51$)	95.00 ($\sigma = 1.73$)	110.00 ($\sigma = 3.46$)	61.33 ($\sigma = 1.53$)	86.00 ($\sigma = 1.73$)	128.00 ($\sigma = 6.93$)
Real-time Total Cost	158.19 ($\sigma = 4.99$)	176.94 ($\sigma = 4.70$)	185.95 ($\sigma = 3.40$)	158.68 ($\sigma = 1.54$)	183.35 ($\sigma = 1.40$)	225.35 ($\sigma = 6.56$)
Near Real-time Total Cost	94.44 ($\sigma = 6.14$)	111.43 ($\sigma = 5.65$)	121.61 ($\sigma = 2.89$)	115.55 ($\sigma = 1.14$)	140.22 ($\sigma = 1.77$)	182.22 ($\sigma = 6.88$)
Relaxed Total Cost	88.11 ($\sigma = 5.86$)	106.67 ($\sigma = 4.69$)	117.91 ($\sigma = 3.16$)	107.54 ($\sigma = 1.47$)	132.21 ($\sigma = 2.14$)	174.21 ($\sigma = 7.25$)

Table 8.6: Evaluation Results for Random Walk 2

	BTU-based			Threshold-based		
	BTU10	BTU30	BTU60	BTU10	BTU30	BTU60
Real-time Compliance	49% ($\sigma = 2\%$)	51% ($\sigma = 1\%$)	53% ($\sigma = 0\%$)	41% ($\sigma = 1\%$)		
Near Real-time Compliance	87% ($\sigma = 2\%$)	90% ($\sigma = 1\%$)	92% ($\sigma = 1\%$)	70% ($\sigma = 1\%$)		
Relaxed Compliance	90% ($\sigma = 2\%$)	94% ($\sigma = 1\%$)	95% ($\sigma = 0\%$)	75% ($\sigma = 75\%$)		
Resource Cost	74.00 ($\sigma = 6.08$)	90.00 ($\sigma = 0.00$)	106.00 ($\sigma = 3.46$)	59.67 ($\sigma = 4.04$)	82.00 ($\sigma = 9.17$)	118.00 ($\sigma = 9.17$)
Real-time Total Cost	172.67 ($\sigma = 5.35$)	184.03 ($\sigma = 0.99$)	195.73 ($\sigma = 3.33$)	164.18 ($\sigma = 3.84$)	187.41 ($\sigma = 9.88$)	223.41 ($\sigma = 9.88$)
Near Real-time Total Cost	100.17 ($\sigma = 6.65$)	108.91 ($\sigma = 2.47$)	120.59 ($\sigma = 3.59$)	113.41 ($\sigma = 3.47$)	135.98 ($\sigma = 9.93$)	171.98 ($\sigma = 9.93$)
Relaxed Total Cost	92.67 ($\sigma = 7.15$)	101.96 ($\sigma = 1.61$)	115.67 ($\sigma = 3.48$)	104,43 ($\sigma = 2.49$)	127.19 ($\sigma = 8.29$)	163.19 ($\sigma = 8.29$)



(a) BTU-based (BTU10)

(b) BTU-based (BTU30)

(c) BTU-based (BTU60)

(d) Threshold-based

Figure 8.9: Randomwalk Pattern 2

8.6.5 Evaluation Conclusion

When we compare the evaluation results of the four different data arrival patterns, we can see that they all share the same trend. Regarding the SLA compliance, we can see that the BTU-based approach achieves a better SLA compliance for all configurations for all compliance scenarios. Furthermore, the SLA values are roughly the same (with a maximum difference of 3%) across all data arrival patterns despite their different characteristics. For the total cost, we can observe that only the BTU10 configuration for the real-time scenario results in higher cost in contrast to the baseline. All other configurations and scenarios for the BTU-based approach exhibit a cost reduction. In addition, it must be noted that the resource cost are always lower for the BTU60 configuration than for the threshold-based approach.

We can also observe that the compliance for real-time data processing on cloud infrastructures is rather low, i.e., around 40% for the baseline and around 50%-55% for the BTU-based approach. This is mainly due to the fact that cloud environments are often influenced by other applications running on the same physical hardware. This can result in minor data transmissions or processing delays that have a severe impact on the SLA compliance. Nevertheless, we can see that for the near real-time and relaxed time scenarios, the SLA compliance ranges from 84% to 95% for the BTU-based approach, which meets the requirements of our motivational scenario discussed in Section 8.2.2.

8.6.6 Threats to Applicability

Although the presented system model builds on top of real-world observations, it cannot be guaranteed that all external aspects are adequately considered in our system model which may result in a non-optimal performance in real-world deployments. However we consider this risk as rather small, since we conducted our evaluations in a cloud-based testbed which already considers external influences by other applications running on the same physical hardware. To consider such external effects for the evaluation, we executed each evaluation scenario and configuration three times on different days (including the weekend) to cover different usage scenarios on the OpenStack-based cloud due to other stakeholders on the same physical hardware.

8.7 Summary

Within this chapter, we have discussed the most important requirements for optimizing data stream processing in volatile environments. Based on these requirements, we developed a system model for which we have presented a BTU-based optimization approach. This optimization approach has been evaluated based on four different data arrival pattern against a threshold-based approach, which was introduced in Chapter 7, which already provides a significant cost reduction compared to an only threshold-based one as presented in the previous Chapter 7. Furthermore, the evaluation shows that the BTU-based approach results in a better SLA compliance which also achieves a better overall cost structure compared to the threshold-based approach.

CHAPTER 9

Conclusions

In this final chapter, we summarize the contributions of this thesis and revisit our research questions formulated in Section 1.2 to put our contributions in perspective. Finally, we also present future research directions to conclude this thesis.

9.1 Summary of Contributions

Within this thesis, we introduce and combine several novel concepts that are required to create a data stream processing ecosystem with an emphasis on the cost-efficient operation of SPAs in distributed environments. To achieve this goal, this thesis takes a holistic viewpoint to address the challenges that have been introduced in Section 1.1 from different angles ranging from a novel system design for SPEs over an extensible topology definition approach for SPAs to cost-efficient resource provisioning strategies. Each of these angles addresses a specific challenge and these three major contributions combined form the foundation for the VISP Ecosystem.

First, we propose the creation of an ecosystem for data stream processing to improve the usability and efficiency of creating and operating SPAs by combining the established concept of SPEs with operator marketplaces. Based on a literature survey, we then identify that established SPEs often cannot cope with challenges originating from the IoT. These challenges mainly consider the geographic distributed locations of data sources and computational resources that process the data, but also the volatile data volume that change due to external events [32]. To address this shortcoming, we introduce the VISP Runtime, which represents a novel system design for SPEs. The VISP Runtime builds on top of cloud computing techniques that allow the SPE to react on-demand to the volatile data volume in contrast to established SPEs that are hardly able to change the resource configuration for SPAs at run time. In addition, the VISP Runtime also supports the autonomous coordination among individual VISP Runtime instances in different geographic locations based on a dedicated coordination protocol.

This allows users to deploy SPAs in a network-efficient manner by using only a single user interface to manage the operational aspects of SPAs as they are operated with today's centralized SPEs. The main outcome of this contribution is not only the proposed system design for the VISP Runtime but also a reference implementation thereof. This reference implementation serves two major tasks: On the one hand, it allows the validation of our requirements towards the VISP Ecosystem, and on the other hand, it serves as a customizable platform to implement and evaluate further aspects within this thesis, but also for related research projects, e.g., the Master theses by Hiessl [138] and Knasmüller [139].

Second, we introduce a novel topology definition approach for SPAs that facilitates the design for SPAs in distributed environments as well as the description of non-functional operational aspects like SLAs. Up to now, most topology definition approaches only consider a centralized runtime environment that does not support any aspects of a distributed deployment. Furthermore, most SPEs manage the SPAs based on a best-effort principle for all operators of an SPA and do not enable SPA users to apply non-functional requirements, like SLAs on an operator level. To address this issue, we have identified several non-functional requirements for SPAs based on a literature survey. These requirements are integrated into the established SPL to create the VTDL, which has been introduced within this thesis. In addition, we also design and implement several coordination mechanisms to support the VTDL within the VISP Runtime and conducted several experiments to evaluate both the feasibility as well as the reduced management overhead by using the VTDL in contrast to other established topology definition approaches.

The third major contribution focuses on the cost-efficiency of operating SPAs that are exposed to volatile data volume. To address this aspect, we leverage the resource elasticity of cloud-based computational resources. Therefore, we propose two dynamic resource provisioning approaches that update the resource configuration of individual operators on demand, depending on different KPIs, like resource usage or SLA compliance. To minimize the operational cost, we create an optimization model that represents the topology of the SPA, all available computational resources, and the KPIs. This model is then optimized based on one of the two proposed resource provisioning approaches and this optimized model can be applied to update the resource configuration of the SPA in a recurring interval. In order to evaluate our approaches, we have also created the VISP Testbed, which allows us to compare our approaches against baseline approaches provided by the literature. The results of our evaluation show that both approaches are capable of reducing the operational costs while guaranteeing a high level of QoS.

These three major contributions provide the foundation for the VISP Ecosystem and ensure that SPAs can be operated in a cost-efficient manner. All of the above-mentioned implementations, like the VISP Runtime, the reference implementation for the VTDL as well as the reference implementations for both resource provisioning approaches are available as open source [108] and have been evaluated rigorously based on different scenarios from the manufacturing domain.

9.2 Research Questions Revisited

In Section 1.2, we have introduced three research questions, which guide the work in this thesis. Therefore, we are going to revisit these research questions in this section, to summarize how these questions have been answered within this thesis and outline possible limitations of our work.

Research Question I

How can geographically distributed data streams be processed efficiently?

We have addressed this question in Chapter 5 by introducing the VISP Runtime as a novel system design for SPEs. Up to now, SPEs often only support a centralized deployment. This however renders these SPEs very inefficient in terms of data processing of data streams originating from different geographic locations because all the data needs to be transferred to a single location and this data transfer poses a high load on the network. The VISP Runtime uses a distributed deployment of multiple instances in different geographic locations to partition the load across several geographic regions and to reduce the load to a single location. The first notable improvement of the VISP Runtime in contrast to established other SPEs, is a dedicated communication protocol, which allows the VISP Runtime instances to synchronize the state among them autonomously. The second notable enhancement is that the VISP Runtime is capable of updating both the resource configuration as well as the reconfiguration of the topology of an SPA at run time. This reconfiguration feature allows the SPE to react on any changes of the data streams to improve the overall data throughput and therefore also the efficiency of the data processing. These reconfigurations can either consider the spawning of additional replicas for a specific operator, which is possible due to its cloud-native design or the relocation of specific operators to other geographic locations to reduce the load.

Research Question II

How can stream processing applications be described to allow a distributed deployment model and respect service level objectives?

We have addressed this question in Chapter 6 by introducing VTDL, a novel topology definition approach for SPAs. This topology definition approach represents an evolution of established ones that only focus on the deployment of SPAs in a single geographic location and a best-effort approach for assigning computational resources to operators.

Before designing the system model for our approach, we identified several functional as well as non-functional aspects of designing and operating SPAs from the literature and combined them to form the system model of our approach.

This model has then been implemented based on the established SPL topology definition approach to become the VTDL. The VTDL allows users to describe functional as well as non-functional aspects for SPAs, like the structure of SPAs, the explicit distributed deployment, or SLOs on an abstract level. Nevertheless, it must be noted that the sole description of those aspects for SPAs is not sufficient because they also need to be supported by SPEs that interpret these instructions and apply them accordingly. Therefore, we have integrated a reference implementation for the VTDL into the VISP Runtime to evaluate the system model of the VTDL. Although our contributions provided in this thesis answer this research question, it would also be interesting how and especially which aspects of the VTDL model can be integrated into other established SPEs.

Research Question III

How can stream processing applications be optimally executed on computational resources?

We have addressed this question in Chapter 7 and Chapter 8 by proposing two novel resource provisioning approaches to optimize the execution of SPAs. In order to develop these approaches, we have first identified different KPIs that indicate the need for a reconfiguration of the computational resources for SPAs or individual operators. These KPIs can either describe internal aspects, like an increased CPU load or the number of data items waiting on the messaging infrastructure, or external ones, like the end of an BTU or the failure of an Operator Instance. Based on these KPIs, we have then devised two resource provisioning approaches to optimize the resource usage of computational resources and therefore minimize the operational cost for running SPAs. The first approach (introduced in Chapter 7) represents a basic approach, which updates the resource configuration whenever a specific KPI reaches a predefined threshold. Although this approach reduces the overall need for computational resources, it also introduces a number of reconfiguration tasks for the SPEs which in turn require additional computational resources. These reconfigurations have been addressed by the second approach (introduced in Chapter 8), which evaluates whether the suggested reconfigurations actually improve the optimal execution of SPAs significantly or only achieve very minimal resource optimizations. This approach therefore only applies those reconfigurations that are essential to achieving a better resource usage and as a result reduces the management overhead for SPEs compared to the first approach. Nevertheless, it must be noted that both approaches improve the usage of computational resources dramatically in contrast to fixed provisioning strategies, which represent the state-of-the-art for established SPEs. Our work has improved the resource usage within a single geographical location. We also identified that the migration of specific operators within a distributed deployment could also reduce the overall resource usage and lead to an optimal execution of SPAs. This aspect has been addressed by Hiessl [138], who proposes an optimization approach which also relocates individual operators if required across different VISP Runtime instances.

9.3 Future Work

In this thesis, we introduced the foundation for establishing a cost-efficient data stream processing ecosystem. However, in the course of our thesis we have also identified several aspects which could be optimized. In the remainder of this section, we therefore outline several open challenges and discuss possible future research areas.

Optimal Deployment of Operators across Different Geographic Locations One of the most promising research directions is an extension of the third research question, namely the optimal resource provisioning of SPAs on computational resources. In this thesis, we focus on the resource optimization within one geographic location and hence only achieved local optimas. However, it would be desirable to design optimization approaches that find a global optimum across all geographic locations. In close collaboration with this thesis, there have already been some advances in this direction. Hiessl [138] proposes an iterative global optimization approach, which constantly evaluates basic KPIs like data throughput to calculate an optimal deployment for SPAs and to trigger operator migrations if required. This approach has been implemented within the VISP Ecosystem and indicates that such a global optimization approach could improve the data processing efficiency even further. Nevertheless, there still are several aspects that need to be investigated for future approaches, like the structure of the topologies, e.g., the critical path within topologies, or potential negative side-effects of scaling or migration operations. Furthermore, it would also be desirable to implement more resource provisioning approaches from the literature that can be used as baselines for the VISP Testbed and promote it to a benchmarking infrastructure for future approaches.

Development of Complex KPIs One of the most important aspects for operating and especially optimizing the operation of SPAs are KPIs that can be used by both resource optimization as well as failure compensation mechanisms. Up to now, the VISP Runtime relies on basic KPIs, like resource usage, throughput or SLA compliance for individual operators. Although these KPIs cover most of the operational aspects, there still are some edge cases where these basic KPIs may be misleading. Therefore, it is required to investigate towards the combination of simple KPIs, e.g., resource usage in relation to the data item throughput, as already proposed for cloud-based services by Moldovan et al [140]. These KPIs can then be used by optimization approaches to maybe obtain even better optimization plans than those proposed in this thesis.

Business Models for IoT Marketplaces Another promising research direction is the investigation towards feasible business models for both IoT-based data as well as operators to improve the monetization on the VISP Marketplace. Although there is already some preliminary work for the monetization of data by Tien-Dung et al. [48] as well as by Biasion [141] for arbitrary services for the IoT, there still are several open research challenges. Here, the most important challenge is to model the volatile structure and network effects of IoT-based environments [142] to derive sustainable business models.

Complex Failure Compensation Strategies for SPAs Up to now, the VISP Runtime and the VTDL only consider rather basic failure compensation strategies, like re-deploying single Operator Instances or redeploying the complete SPA. Although these approaches are sufficient for reestablishing the functionality of SPAs after incidents, they still require some time for recovery which results in processing delays. Therefore, it is essential to investigate towards more sophisticated compensation strategies, like the one presented by Knasmüller [139] who proposes the usage of the circuit-breaker pattern for SPAs. This approach already reduces the compensation time within the VISP Ecosystem but there are still several open research challenges like the investigation towards reliable failure indicators to apply predictive compensation measures.

Data Privacy and Data Ownership The last important challenge is the consideration of data ownership and data privacy aspects in distributed stream processing systems [143]. Although the distributed nature of individual VISP Runtimes in contrast to a single centralized SPE already allows to filter or preprocess data, e.g., anonymize, within the jurisdiction of the data source, there are still several open research challenges. The most important challenge is to design solid anonymization strategies that ensure the privacy of the data without rendering the data useless. While this challenge is universal for all software systems, it is necessary to also optimize these strategies in terms of data throughput to facilitate a data processing without delay.

Acronyms

API Application Programming Interface.

BPMS Business Process Management Systems.

BTU Billing Time Unit.

CQL Continuous Query Language.

CREMA Cloud-based Rapid Elastic Manufacturing.

FIFO First-In, First-Out.

IoT Internet of Things.

IP Internet Protocol.

KPI Key Performance Indicator.

OCR Optical Character Recognition.

OEE Overall Equipment Effectiveness.

PaaS Platform as a Service.

QoS Quality of Service.

SaaS Software as a Service.

SLA Service Level Agreement.

SLO Service Level Objective.

SPA Stream Processing Application.

SPE Stream Processing Engine.

SPL Stream Processing Language.

SQL Structured Query Language.

URL Uniform Resource Locator.

VISP VIenna ecosystem for Stream Processing.

VM Virtual Machine.

VTDL Vienna Topology Description Language.

Bibliography

[1] C. Hochreiner, S. Schulte, S. Dustdar, and F. Lecue, "Elastic Stream Processing for Distributed Environments", *IEEE Internet Computing*, vol. 19, no. 6, pp. 54–59, 2015. DOI: 10.1109/mic.2015.118.

[2] C. Hochreiner, M. Vögler, S. Schulte, and S. Dustdar, "Elastic Stream Processing for the Internet of Things", in *9th International Conference on Cloud Computing (CLOUD)*, IEEE, 2016, pp. 100–107. DOI: 10.1109/cloud.2016.0023.

[3] C. Hochreiner, M. Vögler, P. Waibel, and S. Dustdar, "VISP: An Ecosystem for Elastic Data Stream Processing for the Internet of Things", in *20th International Enterprise Distributed Object Computing Conference (EDOC)*, IEEE, 2016, pp. 19–29. DOI: 10.1109/edoc.2016.7579390.

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

[5] C. Hochreiner, M. Vögler, S. Schulte, and S. Dustdar, "Nomadic Applications Traveling in the Fog", in *2nd EAI International Conference on Cloud Networking for Internet of Things Systems (CN4IoT)*, vol. 189, Springer International Publishing, 2018, pp. 151–161. DOI: 10.1007/978-3-319-67636-4_17.

[6] C. Hochreiner, M. Vögler, S. Schulte, and S. Dustdar, "Cost-efficient enactment of stream processing topologies", *PeerJ Computer Science*, vol. 3, e141, 2017. DOI: 10.7717/peerj-cs.141.

[7] C. Hochreiner, M. Nardelli, B. Knasmüller, S. Schulte, and S. Dustdar, "VTDL: A Notation for Stream Processing Applications (accepted for publication)", in *12th International Symposium on Service-Oriented System Engineering (SOSE)*, IEEE, 2018, NN–NN.

[8] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A Survey", *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, 2010. DOI: 10.1016/j.comnet.2010.05.010.

[9] Gartner, *Gartner Says 8.4 Billion Connected Things Will Be in Use in 2017, Up 31 Percent From 2016*, https://www.gartner.com/newsroom/id/3598917, Retrieved December 30, 2017.

[10] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo, "SPADE: The System S Declarative Stream Processing Engine", in *International Conference on Management of Data (SIGMOD)*, ACM, 2008, pp. 1123–1134. DOI: 10.1145/1376616.1376729.

[11] H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. Zdonik, "Retrospective on Aurora", *Proceedings of the VLDB Endowment*, vol. 13, no. 4, pp. 370–383, 2004. DOI: 10.1007/s00778-004-0133-5.

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

[13] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani, "SPC: A distributed, scalable platform for data mining", in *4th International Workshop on Data mining standards, services and platforms*, 2006, pp. 27–37.

[14] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter Heron: Stream processing at scale", in *International Conference on Management of Data (SIGMOD)*, ACM, 2015, pp. 239–250. DOI: 10.1145/2723372.2742788.

[15] M. Fu, S. Mittal, V. Kedigehalli, K. Ramasamy, M. Barry, A. Jorgensen, C. Kellogg, N. Lu, B. Graham, and J. Wu, "Streaming@Twitter", *IEEE Data Engineering*, pp. 15–27, 2015.

[16] Z. Zhuang, T. Feng, Y. Pan, H. Ramachandra, and B. Sridharan, "Effective Multi-stream Joining in Apache Samza Framework", in *International Congress on Big Data*, IEEE, 2016, pp. 267–274. DOI: 10.1109/bigdatacongress.2016.41.

[17] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@twitter", in *International Conference on Management of Data (SIGMOD)*, ACM, 2014, pp. 147–156. DOI: 10.1145/2588555.2595641.

[18] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets", *HotCloud*, vol. 10, pp. 10–17, 2010.

[19] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: stateful scalable stream processing at LinkedIn", *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1634–1645, 2017. DOI: 10.14778/3137765.3137770.

[20] M. Swan, "Sensor mania! the internet of things, wearable computing, objective metrics, and the quantified self 2.0", *Journal of Sensor and Actuator Networks*, vol. 1, no. 3, pp. 217–253, 2012. DOI: 10.3390/jsan1030217.

[21] R. K. Kodali and S. Mandal, "IoT based weather station", in *International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*, IEEE, 2016, pp. 680–683. DOI: 10.1109/iccicct.2016.7988038.

[22] Z. Shen, V. Kumaran, M. J. Franklin, S. Krishnamurthy, A. Bhat, M. Kumar, R. Lerche, and K. Macpherson, "CSA: Streaming Engine for Internet of Things", *IEEE Data Engineering*, pp. 39–50, 2015.

[23] A. Botta, W. de Donato, V. Persico, and A. Pescapé, "Integration of Cloud Computing and Internet of Things: A Survey", *Future Generation Computer Systems*, vol. 56, pp. 684–700, 2016. DOI: 10.1016/j.future.2015.09.021.

[24] A. Clemm, M. Chandramouli, and S. Krishnamurthy, "DNA: An SDN framework for distributed network analytics", in *International Symposium on Integrated Network Management (IM)*, IFIP/IEEE, 2015, pp. 9–17. DOI: 10.1109/inm.2015.7140271.

[25] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, "Network-aware operator placement for stream-processing systems", in *22nd International Conference on Data Engineering, 2006 (ICDE)*, 2006, pp. 49–49. DOI: 10.1109/icde.2006.105.

[26] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer, "Auto-scaling techniques for elastic data stream processing", in *30th International Conference on Data Engineering Workshops (ICDEW)*, 2014, pp. 296–302. DOI: 10.1109/icdew.2014.6818344.

[27] C. W. Krueger, "Software reuse", *ACM Computing Surveys (CSUR)*, vol. 24, no. 2, pp. 131–183, 1992. DOI: 10.1145/130844.130856.

[28] M. Hirzel, S. Schneider, and B. Gedik, "SPL: An Extensible Language for Distributed Stream Processing", *ACM Transactions Programming Languages and Systems*, vol. 39, no. 1, 5:1–5:39, Mar. 2017. DOI: 10.1145/3039207.

[29] H. C. Andrade, B. Gedik, and D. S. Turaga, *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge University Press, 2014.

[30] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing", *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010. DOI: 10.1145/1721654.1721672.

[31] V. Mathew, R. K. Sitaraman, and P. Shenoy, "Energy-aware load balancing in content delivery networks", in *Proceedings of 2012 INFOCOM*, IEEE, 2012, pp. 954–962. DOI: 10.1109/INFCOM.2012.6195846.

[32] F. Lécué, S. Tallevi-Diotallevi, J. Hayes, R. Tucker, V. Bicer, M. L. Sbodio, and P. Tommasi, "Star-city: semantic traffic analytics and reasoning for city", in *19th International Conference on Intelligent User Interfaces*, 2014, pp. 179–188. DOI: 10.1145/2557500.2557537.

[33] J. Jung, B. Krishnamurthy, and M. Rabinovich, "Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs and Web Sites", in *11th International Conference on World Wide Web*, ACM, 2002, pp. 293–304. DOI: 10.1145/511446.511485.

[34] S. Chaisiri, B.-S. Lee, and D. Niyato, "Optimization of resource provisioning cost in cloud computing", *IEEE Transactions on Services Computing*, vol. 5, no. 2, pp. 164–177, 2012. DOI: 10.1109/TSC.2011.7.

[35] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker, "Load shedding in a data stream manager", in *Proceedings of the VLDB Endowment*, VLDB Endowment, 2003, pp. 309–320.

[36] N. Tatbul, U. Çetintemel, and S. Zdonik, "Staying fit: Efficient load shedding techniques for distributed stream processing", in *Proceedings of the VLDB Endowment*, VLDB Endowment, 2007, pp. 159–170.

[37] B. Babcock, M. Datar, and R. Motwani, "Load shedding for aggregation queries over data streams", in *20th International Conference on Data Engineering*, IEEE, 2004, pp. 350–361. DOI: 10.1109/ICDE.2004.1320010.

[38] P. Mell and T. Grance, "The NIST definition of cloud computing", 2011. DOI: 10.6028/NIST.SP.800-145.

[39] Z. Shelby, "Embedded web services", *IEEE Wireless Communications*, vol. 17, no. 6, 2010. DOI: 10.1109/mwc.2010.5675778.

[40] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions", *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, 2013. DOI: 10.1016/j.future.2013.01.010.

[41] J. Mineraud, O. Mazhelis, X. Su, and S. Tarkoma, "A gap analysis of Internet-of-Things platforms", *Computer Communications*, vol. 89, pp. 5–16, 2016. DOI: 10.1016/j.comcom.2016.03.015.

[42] H. Derhamy, J. Eliasson, J. Delsing, and P. Priller, "A survey of commercial frameworks for the Internet of Things", in *20th Conference on Emerging Technologies & Factory Automation (ETFA)*, IEEE, 2015, pp. 1–8. DOI: 10.1109/etfa.2015.7301661.

[43] X. Su, R. M. Svendsen, H. N. Castejón, E. Berg, and J. Zoric, "Towards an integrated solution to internet of things-a technical and economical proposal", in *15th International Conference on Intelligence in Next Generation Networks (ICIN)*, IEEE, 2011, pp. 46–51. DOI: 10.1109/icin.2011.6081101.

[44] A. Castellani, N. Bui, P. Casari, M. Rossi, Z. Shelby, and M. Zorzi, "Architecture and protocols for the internet of things: A case study", in *8th International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, IEEE, 2010, pp. 678–683. DOI: 10.1109/percomw.2010.5470520.

[45] S. De, P. Barnaghi, M. Bauer, and S. Meissner, "Service modelling for the Internet of Things", in *Federated Conference on Computer Science and Information Systems (FedCSIS)*, IEEE, 2011, pp. 949–955.

[46] D. Munjin and J Morin, "Toward internet of things application markets", in *International Conference on Green Computing and Communications (GreenCom)*, IEEE, 2012, pp. 156–162. DOI: 10.1109/greencom.2012.33.

[47] K. Akpınar, K. A. Hua, and K. Li, "ThingStore: A Platform for Internet-of-Things Application Development and Deployment", in *9th International Conference on Distributed Event-Based Systems (DEBS)*, ACM, 2015, pp. 162–173. DOI: 10.1145/2675743.2771833.

[48] C. Tien-Dung, T.-V. Pham, V. Quang-Hieu, H.-L. Truong, D.-H. Le, and S. Dustdar, "MARSA: A Marketplace for Realtime Human-Sensing Data", *ACM Transactions on Internet Technology (TOIT)*, vol. 16, no. 3, pp. 1–21, 2016. DOI: 10.1145/2883611.

[49] GroveStreams, *IoT Platform*, https://grovestreams.com, Retrieved December 30, 2017.

[50] D. Guinard and V. Trifa, "Towards the web of things: Web mashups for embedded devices", in *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in International World Wide Web Conferences (WWW)*, 2009, p. 15.

[51] ThingWorx, *Foundation*, https://www.thingworx.com, Retrieved December 30, 2017.

[52] Amazon, *Web Services IoT Platform*, https://aws.amazon.com/iot-platform/, Retrieved December 30, 2017.

[53] Google, *Cloud Dataflow*, https://cloud.google.com/dataflow/, Retrieved December 30, 2017.

[54] Microsoft, *Azure Stream Analytics*, https://azure.microsoft.com/en-us/services/stream-analytics/, Retrieved December 30, 2017.

[55] Apache Software Foundation, *Edgent*, https://edgent.apache.org, Retrieved December 30, 2017.

[56] Y. Qin, Q. Z. Sheng, N. J. Falkner, S. Dustdar, H. Wang, and A. V. Vasilakos, "When Things Matter: A Survey on Data-Centric Internet of Things", *Journal of Network and Computer Applications*, 2016. DOI: 10.1016/j.jnca.2015.12.016.

[57] N. Tatbul, Y. Ahmad, U. Çetintemel, J.-H. Hwang, Y. Xing, and S. Zdonik, "Load Management and High Availability in the Borealis Distributed Stream Processing Engine", in *GeoSensor Networks*, 2008, pp. 66–85. DOI: 10.1007/978-3-540-79996-2_5.

[58] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom, "STREAM: the stanford stream data manager (demonstration description)", in *International Conference on Management of Data (SIGMOD)*, 2003, pp. 665–665. DOI: 10.1145/872757.872854.

[59] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "MillWheel: fault-tolerant stream processing at internet scale", *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013. DOI: 10.14778/2536222.2536229.

[60] W. Hummer, B. Satzger, and S. Dustdar, "Elastic stream processing in the cloud", *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 3, no. 5, pp. 333–345, 2013. DOI: 10.1002/widm.1100.

[61] B. Satzger, W. Hummer, P. Leitner, and S. Dustdar, "Esc: Towards an elastic stream computing platform for the cloud", in *International Conference on Cloud Computing (CLOUD)*, 2011, pp. 348–355. DOI: 10.1109/cloud.2011.27.

[62] T. Heinze, "Elastic complex event processing", in *8th Middleware Doctoral Symposium (MDS)*, ACM, 2011, 4:1–4:6. DOI: 10.1145/2093190.2093194.

[63] A. Ishii and T. Suzumura, "Elastic stream computing with clouds", in *International Conference on Cloud Computing (CLOUD)*, IEEE, 2011, pp. 195–202. DOI: 10.1109/cloud.2011.11.

[64] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink™: Stream and Batch Processing in a Single Engine", *IEEE Data Engineering Bulletin*, vol. 38, no. 4, pp. 28–38, 2015.

[65] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Distributed QoS-aware scheduling in storm", in *9th International Conference on Distributed Event-Based Systems (DEBS)*, ACM, 2015, pp. 344–347. DOI: 10.1145/2675743.2776766.

[66] Spring, *Cloud Data Flow*, http://cloud.spring.io/spring-cloud-dataflow/, Retrieved December 30, 2017.

[67] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. B. Zdonik, "Scalable Distributed Stream Processing", in *Conference on Innovative Data Systems Research (CIDR)*, vol. 3, 2003, pp. 257–268.

[68] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management", in *International Conference on Management of Data (SIGMOD)*, 2013, pp. 725–736. DOI: 10.1145/2463676.2465282.

[69] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing", *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 25, no. 6, pp. 1447–1463, 2014. DOI: 10.1109/tpds.2013.295.

[70] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg", in *10th European Conference on Computer Systems (EuroSys)*, 2015, 18:1–18:17. DOI: 10.1145/2741948.2741964.

[71] V. Setty, R. Vitenberg, G. Kreitz, G. Urdaneta, and M. Van Steen, "Cost-effective resource allocation for deploying pub/sub on cloud", in *34th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 2014, pp. 555–566. DOI: 10.1109/icdcs.2014.63.

[72] D. Florescu and D. Kossmann, "Rethinking cost and performance of database systems", *ACM Sigmod Record*, vol. 38, no. 1, pp. 43–48, 2009. DOI: 10.1145/1558334.1558339.

[73] H. Lim, Y. Han, and S. Babu, "How to Fit when No One Size Fits", in *Conference on Innovative Data Systems Research (CIDR)*, vol. 4, 2013, p. 35.

[74] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu, "Elastic scaling of data parallel operators in stream processing", in *International Symposium on Parallel & Distributed Processing (IPDPS)*, IEEE, 2009, pp. 1–12. DOI: 10.1109/ipdps.2009.5161036.

[75] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in Storm", in *7th International Conference on Distributed Event-based Systems (DEBS)*, ACM, 2013, pp. 207–218. DOI: 10.1145/2488222.2488267.

[76] J. Xu, Z. Chen, J. Tang, and S. Su, "T-Storm: traffic-aware online scheduling in storm", in *34th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 2014, pp. 535–544. DOI: 10.1109/icdcs.2014.61.

[77] L. Xu, B. Peng, and I. Gupta, "Stela: Enabling Stream Processing Systems to Scale-in and Scale-out On-demand", in *International Conference on Cloud Engineering (IC2E)*, IEEE, 2016.

[78] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, and C. Fetzer, "Online parameter optimization for elastic data stream processing", in *6th Symposium on Cloud Computing*, ACM, 2015, pp. 276–287. DOI: 10.1145/2806777.2806847.

[79] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, "Streamcloud: An elastic and scalable data streaming system", *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 12, pp. 2351–2365, 2012. DOI: 10.1109/tpds.2012.24.

[80] T. De Matteis and G. Mencagli, "Keep calm and react with foresight: strategies for low-latency and energy-efficient elastic data stream processing", in *21st Symposium on Principles and Practice of Parallel Programming (SIGPLAN)*, ACM, 2016, pp. 1–12. DOI: 10.1145/3016078.2851148.

[81] F. Hanna, L. Marchal, J.-M. Nicod, L. Philippe, V. Rehn-Sonigo, and H. Sabbah, "Minimizing Rental Cost for Multiple Recipe Applications in the Cloud", in *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, 2016, pp. 28–37. DOI: 10.1109/ipdpsw.2016.71.

[82] Apache Software Foundation, *Beam*, https://beam.apache.org, Retrieved December 30, 2017.

[83] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing", *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1792–1803, 2015. DOI: 10.14778/2824032.2824076.

[84] M. Hirzel, H. Andrade, B. Gedik, V. Kumar, G. Losa, M. Nasgaard, R Soule, and K. Wu, "SPL Stream Processing Language Specification", *IBM Journal Research Division*, 2009. DOI: 10.1147/JRD.2013.2243535.

[85] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: semantic foundations and query execution", *Proceedings of the VLDB Endowment*, vol. 15, no. 2, pp. 121–142, 2006.

[86] R. Soulé, M. Hirzel, B. Gedik, and R. Grimm, "River: an intermediate language for stream processing", *Software: Practice and Experience*, pp. 891–929, 2016. DOI: 10.1002/spe.2338.

[87] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications", in *11th International Conference on Compiler Construction*, Springer, 2002, pp. 179–196. DOI: 10.1007/3-540-45937-5_14.

[88] Apache Software Foundation, *Apex*, https://apex.apache.org, Retrieved December 30, 2017.

[89] Apache Software Foundation, *Kafka*, https://kafka.apache.org/documentation/streams/, Retrieved December 30, 2017.

[90] *Flux Framework for Apache Storm*, http://storm.apache.org/releases/2.0.0-SNAPSHOT/flux.html, Retrieved December 30, 2017, 2017.

[91] S. Schulte, P. Hoenisch, C. Hochreiner, S. Dustdar, M. Klusch, and D. Schuller, "Towards process support for cloud manufacturing", in *18th International Enterprise Distributed Object Computing Conference (EDOC)*, IEEE, 2014, pp. 142–149. DOI: 10.1109/edoc.2014.28.

[92] S. Nakajima, *Introduction to TPM: Total Productive Maintenance.* 11th Edition. Productivity Press, 1988, ISBN: 0915299232.

[93] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac, "Internet of things: Vision, applications and research challenges", *Ad Hoc Networks*, vol. 10, no. 7, pp. 1497–1516, 2012. DOI: 10.1016/j.adhoc.2012.02.016.

[94] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the Internet of Things", in *1st edition of the MCC workshop on Mobile Cloud Computing*, ACM, 2012, pp. 13–16. DOI: 10.1145/2342509.2342513.

[95] J. Liang, S. Y. Ko, I. Gupta, and K. Nahrstedt, "MON: On-Demand Overlays for Distributed System Management", in *WORLDS*, vol. 5, USENIX Association Berkeley, 2005, pp. 13–18.

[96] A. L. Davis and R. M. Keller, "Data Flow Program Graphs", *Computer*, vol. 15, no. 2, pp. 26–41, Feb. 1982. DOI: 10.1109/MC.1982.1653939.

[97] Y. Ai, M. Peng, and K. Zhang, "Edge Cloud Computing Technologies for Internet of Things: A Primer", *Digital Communications and Networks*, 2017. DOI: 10.1016/j.dcan.2017.07.001.

[98] N. Mohan and J. Kangasharju, "Edge-fog cloud: A distributed cloud for internet of things computations", in *Cloudification of the Internet of Things (CIoT)*, IEEE, 2016, pp. 1–6. DOI: 10.1109/ciot.2016.7872914.

[99] Ø. Hauge, C. Ayala, and R. Conradi, "Adoption of open source software in software-intensive publishers–A systematic literature review", *Information and Software Technology*, vol. 52, no. 11, pp. 1133–1154, 2010. DOI: 10.1016/j.infsof.2010.05.008.

[100] S. Turber, J. vom Brocke, O. Gassmann, and E. Fleisch, "Designing Business Models in the Era of Internet of Things", in *Advancing the Impact of Design Science: Moving from Theory to Practice*, Springer, 2014, pp. 17–31. DOI: 10.1007/978-3-319-06701-8_2.

[101] Docker, *Software Containerization Platform*, https://www.docker.com, Retrieved December 30, 2017.

[102] S. Vinoski, "Advanced message queuing protocol", *IEEE Internet Computing*, no. 6, pp. 87–89, 2006. DOI: 10.1109/mic.2006.116.

[103] ISO/IEC, *ISO/IEC 20922. Software engineering – Information technology – Message Queuing Telemetry Transport (MQTT) v3.1.1.* ISO/IEC, under development.

[104] redislabs, *Redis*, https://redis.io, Retrieved December 30, 2017.

[105] Amazon, *Elastic Compute Cloud (EC2)*, https://aws.amazon.com/ec2/, Retrieved December 30, 2017.

[106] O. Foundation, *OpenStack*, https://www.openstack.org, Retrieved December 30, 2017.

[107] F. Keller and S. Wendt, "FMC: An approach towards architecture-centric system development", in *10th International Conference and Workshop on the Engineering of Computer-Based Systems*, IEEE, 2003, pp. 173–182. DOI: 10.1109/ECBS.2003.1194797.

[108] C. Hochreiner, *Source Code Repositories for VISP*, https://github.com/visp-streaming, Retrieved December 30, 2017.

[109] Pivotal, *Spring Cloud*, http://projects.spring.io/spring-cloud/, Retrieved December 30, 2017.

[110] Pivotal, *RabbitMQ*, https://www.rabbitmq.com, Retrieved December 30, 2017.

[111] Docker, *Docker Cloud Hosting*, https://cloud.docker.com, Retrieved December 30, 2017.

[112] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized Streams: An Efficient and Fault-tolerant Model for Stream Processing on Large Clusters", in *4th Conference on Hot Topics in Cloud Computing*, ser. HotCloud'12, USENIX Association Berkeley, 2012, pp. 10–10.

[113] L. M. Vaquero and L. Rodero-Merino, "Finding your way in the fog: Towards a comprehensive definition of fog computing", *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 5, pp. 27–32, 2014. DOI: 10.1145/2677046.2677052.

[114] E. Comission, "Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC", *Official Journal of the European Union*, vol. 119, pp. 1–88, 2016-05-04.

[115] M. Villamizar, O. Garces, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, and M. Lang, "Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures", in *16th International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, ACM/IEEE, 2016, pp. 179–182. DOI: 10.1109/ccgrid.2016.37.

[116] T. Heinze, L. Aniello, L. Querzoni, and Z. Jerzak, "Cloud-based data stream processing", in *8th International Conference on Distributed Event-Based Systems (DEBS)*, ACM, 2014, pp. 238–245. DOI: 10.1145/2611286.2611309.

[117] A. Rodriguez, R. McGrath, Y. Liu, and J. Myers, "Semantic management of streaming data", in *2nd International Conference on Semantic Sensor Networks*, vol. 522, CEUR-WS, 2009, pp. 80–95.

[118] W. Wang, P. Barnaghi, G. Cassar, F. Ganz, and P. Navaratnam, "Semantic sensor service networks", in *IEEE Sensors*, IEEE, 2012, pp. 1–4. DOI: 10.1109/icsens.2012.6411490.

[119] G. Jacques-Silva, B. Gedik, R. Wagle, K.-L. Wu, and V. Kumar, "Building user-defined runtime adaptation routines for stream processing applications", *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1826–1837, 2012. DOI: 10.14778/2367502.2367521.

[120] F. Baude, L. El Beze, and M. Oliva, "Towards a flexible data stream analytics platform based on the GCM autonomous software component technology", in *International Conference on High Performance Computing & Simulation (HPCS)*, IEEE, 2016, pp. 34–41. DOI: 10.1109/hpcsim.2016.7568313.

[121] J.-H. Hwang, Y. Xing, U. Cetintemel, and S. Zdonik, "A cooperative, self-configuring high-availability solution for stream processing", in *23rd International Conference on Data Engineering (ICDE)*, IEEE, 2007, pp. 176–185. DOI: 10.1109/icde.2007.367863.

[122] SeleniumHQ, *Browser Automation*, http://www.seleniumhq.org, Retrieved December 30, 2017.

[123] S. Genaud and J. Gossa, "Cost-wait trade-offs in client-side resource provisioning with elastic clouds", in *International Conference on Cloud computing (CLOUD)*, IEEE, 2011, pp. 1–8. DOI: 10.1109/cloud.2011.23.

[124] J. Yuan, Y. Zheng, C. Zhang, W. Xie, X. Xie, G. Sun, and Y. Huang, "T-drive: driving directions based on taxi trajectories", in *18th International Conference on advances in geographic information systems (SIGSPATIAL)*, ACM, 2010, pp. 99–108. DOI: 10.1109/TKDE.2011.200.

[125] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing", *International Conference on Management of Data (SIGMOD)*, vol. 34, no. 4, pp. 42–47, 2005. DOI: 10.1145/1107499.1107504.

[126] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes", *IEEE Cloud Computing*, no. 3, pp. 81–84, 2014. DOI: 10.1109/mcc.2014.51.

[127] R. Zhang, M. Li, and D. Hildebrand, "Finding the big data sweet spot: Towards automatically recommending configurations for hadoop clusters on docker containers", in *International Conference on Cloud Engineering (IC2E)*, IEEE, 2015, pp. 365–368. DOI: 10.1109/IC2E.2015.101.

[128] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center", in *8th Conference on Networked Systems Design and Implementation (NSDI)*, vol. 11, USENIX Association Berkeley, 2011, pp. 22–22.

[129] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet another resource negotiator", in *4th Annual Symposium on Cloud Computing*, ACM, 2013, p. 5. DOI: 10.1145/2523616.2523633.

[130] Google, *Kubernetes*, https://kubernetes.io, Retrieved December 30, 2017.

[131] R. Andonov, V. Poirriez, and S. Rajopadhye, "Unbounded knapsack problem: Dynamic programming revisited", *European Journal of Operational Research*, vol. 123, no. 2, pp. 394–407, 2000. DOI: 10.1016/s0377-2217(99)00265-9.

[132] Google, *Tesseract Open Source OCR Engine*, https://github.com/tesseract-ocr/tesseract, Retrieved December 30, 2017.

[133] Pivotal, *Spring Boot*, https://projects.spring.io/spring-boot/, Retrieved December 30, 2017.

[134] Oracle, *MySQL Database*, https://www.mysql.com, Retrieved December 30, 2017.

[135] Docker, *Docker Public Repository*, https://hub.docker.com, Retrieved December 30, 2017.

[136] Google, *Compute Engine*, https://cloud.google.com/compute/, Retrieved December 30, 2017.

[137] P. Hoenisch, D. Schuller, S. Schulte, C. Hochreiner, and S. Dustdar, "Optimization of Complex Elastic Processes", *IEEE Transactions on Services Computing (TSC)*, vol. 9, no. 5, pp. 700–713, 2016. DOI: 10.1109/tsc.2015.2428246.

[138] T. Hiessl, "Optimizing the Placement of Stream Processing Operators in the Fog", Master's thesis, TU Wien, 2017.

[139] B. Knasmüller, "On Fault Tolerance in Stream Processing Systems", Master's thesis, TU Wien, 2018.

[140] D. Moldovan, G. Copil, H.-L. Truong, and S. Dustdar, "MELA: elasticity analytics for cloud services", *International Journal of Big Data Intelligence*, vol. 2, no. 1, pp. 45–62, 2015. DOI: 10.1504/IJBDI.2015.067569.

[141] T. Biasion, "A Network-based Business Model Framework for the Internet of Things", Master's thesis, TU Wien, 2017.

[142] M. E. Porter and J. E. Heppelmann, "How smart, connected products are transforming competition", *Harvard Business Review*, vol. 92, no. 11, pp. 64–88, 2014.

[143] Z. Yan, P. Zhang, and A. V. Vasilakos, "A survey on trust management for Internet of Things", *Journal of network and computer applications*, vol. 42, pp. 120–134, 2014. DOI: 10.1016/j.jnca.2014.01.014.

[144] P. Waibel, J. Matt, C. Hochreiner, O. Skarlat, R. Hans, and S. Schulte, "Cost-optimized redundant Data Storage in the Cloud", *Service Oriented Computing and Applications (SOCA)*, vol. 11, no. 4, pp. 411–426, 2017. DOI: 10.1007/s11761-017-0218-9.

[145] C. Prybila, S. Schulte, C. Hochreiner, and I. Weber, "Runtime verification for business processes utilizing the Bitcoin blockchain", *Future Generation Computer Systems*, 2017. DOI: 10.1016/j.future.2017.08.024.

[146] C. Hochreiner, S. Schulte, and O. Kopp, "Bericht zum 8. ZEUS Workshop", *Softwaretechnik-Trends*, vol. 36, no. 2, pp. 61–62, 2016.

[147] C. Hochreiner, P. Frühwirt, Z. Ma, P. Kieseberg, S. Schrittwieser, and E. R. Weippl, "Genie in a Model? Why Model Driven Security will not secure your Web Application.", *JoWUA*, vol. 5, no. 3, pp. 44–62, 2014. DOI: 10.22667/JOWUA.2014.09.31.044.

[148] M. Borkowski, C. Hochreiner, and S. Schulte, "Moderated Resource Elasticity for Stream Processing Applications", in *International Workshop on Autonomic Solutions for Parallel and Distributed Data Stream Processing (Auto-DaSP 2017) at 23rd International European Conference on Parallel and Distributed Computing (Euro-Par 2017)*, 2017, NN–NN.

[149] M. Nardelli, C. Hochreiner, and S. Schulte, "Elastic Provisioning of Virtual Machines for Container Deployment", in *International Workshop on Autonomous Control for Performance and Reliability Trade-Offs in Internet of Services (ACPROSS 2017) at 8th International Conference on Performance Engineering (ICPE 2017)*, ACM/SPEC, 2017, pp. 5–10. DOI: 10.1145/3053600.3053602.

[150] L. Mazzola, P. Kapahnke, P. Waibel, C. Hochreiner, and M. Klusch, "FCE4BPMN: On-demand QoS-based Optimised Process Model Execution in the Cloud", in *International Conference on Engineering, Technology and Innovation / International Technology Management Conference (ICE/ITMC)*, IEEE, 2017, pp. 319–328. DOI: 978-1-5386-0774-9/17.

[151] P. Waibel, C. Hochreiner, and S. Schulte, "Cost-Efficient Data Redundancy in the Cloud", in *9th International Conference on Service-Oriented Computing and Applications (SOCA)*, IEEE, 2016, pp. 1–9. DOI: 10.1109/SOCA.2016.12.

[152] M. Borkowski, S. Schulte, and C. Hochreiner, "Predicting Cloud Resource Utilization", in *9th International Conference on Utility and Cloud Computing (UCC)*, IEEE/ACM, 2016, pp. 37–42. DOI: 10.1145/2996890.2996907.

[153] C. Hochreiner, P. Waibel, and M. Borkowski, "Bridging gaps in cloud manufacturing with 3D printing", in *Informatik 2016, 46. Jahrestagung der Gesellschaft für Informatik*, 2016, pp. 1623–1626.

[154] S. Schulte, M. Borkowski, C. Hochreiner, M. Klusch, A. Murguzur, O. Skarlat, and P. Waibel, "Bringing Cloud-based Rapid Elastic Manufacturing to Reality with CREMA", in *Workshop on Intelligent Systems Configuration Services for Flexible Dynamic Global Production Networks (FLEXINET) at the 8th International Conference on Interoperability for Enterprise Systems and Applications (I-ESA 2016)*, 2016, pp. 407–413.

[155] P. Hoenisch, C. Hochreiner, D. Schuller, S. Schulte, J. Mendling, and S. Dustdar, "Cost-Efficient Scheduling of Elastic Processes in Hybrid Clouds", in *8th International Conference on Cloud Computing (CLOUD)*, IEEE, 2015, pp. 17–24. DOI: 10.1109/CLOUD.2015.13.

[156] C. Hochreiner, "Privacy-Aware Scheduling for Inter-Organizational Processes", in *7th Central European Workshop on Services and their Composition (ZEUS)*, 2015, pp. 63–68.

[157] C. Hochreiner, M. Huber, G. Merzdovnik, and E. Weippl, "Towards Practical Methods to Protect the Privacy of Location Information with Mobile Devices", in *7th International Conference on Security of Information and Networks (SIN)*, ACM, 2014, pp. 17–24. DOI: 10.1145/2659651.2659680.

[158] C. Hochreiner, Z. Ma, P. Kieseberg, S. Schrittwieser, and E. R. Weippl, "Using Model Driven Security Approaches in Web Application Development", in *2nd TC5/8 International Conference*, IFIP/Springer, 2014, pp. 419–431. DOI: 10.1007/978-3-642-55032-4_42.

[159] P. Frühwirt, P. Kieseberg, C. Hochreiner, S. Schrittwieser, and E. R. Weippl, "InnoDB Datenbank Forensik Rekonstruktion von Abfragen über Datenbank-interne Logfiles", in *Sicherheit 2014: Sicherheit, Schutz und Zuverlässigkeit, Beiträge der 7. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik*, 2014, pp. 363–374.

[160] P. Hoenisch, D. Schuller, C. Hochreiner, S. Schulte, and S. Dustdar, "Elastic Process Optimization - The Service Instance Placement Problem", Distributed Systems Group, Vienna University of Technology, Tech. Rep. TUV-1841-2014-01, 2014.

[161] C. Hochreiner and S. Schulte, Eds., *8th ZEUS Workshop, Vienna, Austria, January 27-28, 2016*, vol. 1562, CEUR Workshop Proceedings, CEUR-WS.org, 2016.

APPENDIX A

VTDL-based Description for the Stream Processing Application

```
$availabilityUK        = Source() {
  concreteLocation     : ":::::ffff:8083:c001/general",
  type                 : "availabilitySensor",
  outputFormat         : "availability"
}

$filterAvailabilityUK = Operator($availabilityUK) {
  allowedLocations     : ":::::ffff:8083:c001",
  poolPreferences      : "cpu",
  concreteLocation     : ":::::ffff:8083:c001/cpu",
  inputFormat          : "availability",
  type                 : "filterAvailability",
  outputFormat         : "alert",
  stateful             : "false",
  replicationAllowed   : "true",
  responseTime         : "0.5"
}

$productiondataUK      = Source() {
  concreteLocation     : ":::::ffff:8083:c001/general",
  type                 : "productivitySensor",
  outputFormat         : "productivityImage"
}

$parsedataUK           = Operator($productiondataUK) {
  allowedLocations     : ":::::ffff:8083:c001",
  poolPreferences      : "gpu",
  concreteLocation     : ":::::ffff:8083:c001/gpu",
  inputFormat          : "productivityImage",
  type                 : "parsedata",
  outputFormat         : "productivityData",
  stateful             : "false",
  replicationAllowed   : "true",
  responseTime         : "1.5"
}
```

```
$performanceUK          = Operator($parsedataUK) {
  allowedLocations      : ":::::ffff:8083:c001",
  poolPreferences       : "cpu",
  concreteLocation      : ":::::ffff:8083:c001/cpu",
  inputFormat           : "productivityData",
  type                  : "performanceCalculation",
  outputFormat          : "performanceMetrics",
  stateful              : "true",
  replicationAllowed    : "false",
  responseTime          : "0.5"
}

$availabilityoeeUK      = Operator($parsedataUK) {
  allowedLocations      : ":::::ffff:8083:c001",
  poolPreferences       : "cpu",
  concreteLocation      : ":::::ffff:8083:c001/cpu",
  inputFormat           : "productivityData",
  type                  : "availabilityCalculation",
  outputFormat          : "availabilityMetrics",
  stateful              : "true",
  replicationAllowed    : "false",
  responseTime          : "0.5"
}

$qualityUK              = Operator($parsedataUK) {
  allowedLocations      : ":::::ffff:8083:c001",
  poolPreferences       : "cpu",
  concreteLocation      : ":::::ffff:8083:c001/cpu",
  inputFormat           : "productivityData",
  type                  : "qualityCalculation",
  outputFormat          : "qualityMetrics",
  stateful              : "true",
  replicationAllowed    : "false",
  responseTime          : "0.5"
}

$oeeUK                  = Operator($performanceUK, $availabilityoeeUK, $qualityUK) {
  allowedLocations      : ":::::ffff:8083:c001",
  poolPreferences       : "cpu",
  concreteLocation      : ":::::ffff:8083:c001/cpu",
  inputFormat           : "performanceMetrics, availabilityMetrics, qualityMetrics",
  type                  : "oeeCalculation",
  outputFormat          : "oeeMetrics",
  stateful              : "true",
  replicationAllowed    : "false",
  responseTime          : "0.5"
}

$reportUK               = Operator($oeeMetrics) {
  allowedLocations      : ":::::ffff:8083:c001",
  poolPreferences       : "cpu",
  concreteLocation      : ":::::ffff:8083:c001/cpu",
  inputFormat           : "oeeMetrics",
  type                  : "reportgeneration",
  outputFormat          : "report",
  stateful              : "true",
  replicationAllowed    : "false",
  responseTime          : "2.5"
}
```

```
$temperatureUK         = Source() {
  concreteLocation     : "::::ffff:8083:c001/general",
  type                 : "temperatureSensor",
  outputFormat         : "temperature"
}

$monitorTemperatureUK  = Operator($temperatureUK) {
  allowedLocations     : "::::ffff:8083:c001",
  poolPreferences      : "cpu",
  concreteLocation     : "::::ffff:8083:c001/cpu",
  inputFormat          : "temperature",
  type                 : "monitorTemperature",
  outputFormat         : "alert",
  stateful             : "false",
  replicationAllowed   : "true",
  responseTime         : "0.5"
}

$availabilitySE        = Source() {
  concreteLocation     : "::::ffff:8083:c002/general",
  type                 : "availabilitySensor",
  outputFormat         : "availability"
}

$filterAvailabilitySE  = Operator($availabilitySE) {
  allowedLocations     : "::::ffff:8083:c002",
  poolPreferences      : "cpu",
  concreteLocation     : "::::ffff:8083:c002/cpu",
  inputFormat          : "availability",
  type                 : "filterAvailability",
  outputFormat         : "alert",
  stateful             : "false",
  replicationAllowed   : "true",
  responseTime         : "0.5"
}

$temperatureES         = Source() {
  concreteLocation     : "::::ffff:8083:c003/general",
  type                 : "temperatureSensor",
  outputFormat         : "temperature"
}

$monitorTemperatureSE  = Operator($temperatureSE) {
  allowedLocations     : "::::ffff:8083:c002",
  poolPreferences      : "cpu",
  concreteLocation     : "::::ffff:8083:c002/cpu",
  inputFormat          : "temperature",
  type                 : "monitorTemperature",
  outputFormat         : "alert",
  stateful             : "false",
  replicationAllowed   : "true",
  responseTime         : "0.5"
}

$availabilityES        = Source() {
  concreteLocation     : "::::ffff:8083:c003/general",
  type                 : "availabilitySensor",
  outputFormat         : "availability"
}
```

```
$filterAvailabilityES = Operator($availabilityES) {
  allowedLocations    : ":::::ffff:8083:c003",
  poolPreferences     : "cpu",
  concreteLocation    : ":::::ffff:8083:c003/cpu",
  inputFormat         : "availability",
  type                : "filterAvailability",
  outputFormat        : "alert",
  stateful            : "false",
  replicationAllowed  : "true",
  responseTime        : "0.5"
}

$productiondataES     = Source() {
  concreteLocation    : ":::::ffff:8083:c003/general",
  type                : "productivitySensor",
  outputFormat        : "productivityImage"
}

$parsedataES          = Operator($productiondataES) {
  allowedLocations    : ":::::ffff:8083:c003",
  poolPreferences     : "gpu",
  concreteLocation    : ":::::ffff:8083:c003/gpu",
  inputFormat         : "productivityImage",
  type                : "parsedata",
  outputFormat        : "productivityData",
  stateful            : "false",
  replicationAllowed  : "true",
  responseTime        : "1.5"
}

$performanceES        = Operator($parsedataES) {
  allowedLocations    : ":::::ffff:8083:c003",
  poolPreferences     : "cpu",
  concreteLocation    : ":::::ffff:8083:c003/cpu",
  inputFormat         : "productivityData",
  type                : "performanceCalculation",
  outputFormat        : "performanceMetrics",
  stateful            : "true",
  replicationAllowed  : "false"
}

$availabilityoeeES    = Operator($parsedataES) {
  allowedLocations    : ":::::ffff:8083:c003",
  poolPreferences     : "cpu",
  concreteLocation    : ":::::ffff:8083:c003/cpu",
  inputFormat         : "productivityData",
  type                : "availabilityCalculation",
  outputFormat        : "availabilityMetrics",
  stateful            : "true",
  replicationAllowed  : "false"
}

$qualityES            = Operator($parsedataES) {
  allowedLocations    : ":::::ffff:8083:c003",
  poolPreferences     : "cpu",
  concreteLocation    : ":::::ffff:8083:c003/cpu",
  inputFormat         : "productivityData",
  type                : "qualityCalculation",
  outputFormat        : "qualityMetrics",
  stateful            : "true",
  replicationAllowed  : "false"
}
```

```
$oeeES                = Operator($performanceES, $availabilityoeeES, $qualityES) {
  allowedLocations    : ":::::ffff:8083:c003",
  poolPreferences     : "cpu",
  concreteLocation    : ":::::ffff:8083:c003/cpu",
  inputFormat         : "performanceMetrics, availabilityMetrics, qualityMetrics",
  type                : "oeeCalculation",
  outputFormat        : "oeeMetrics",
  stateful            : "true",
  replicationAllowed  : "false"
}

$temperatureES        = Source() {
  concreteLocation    : ":::::ffff:8083:c003/general",
  type                : "temperatureSensor",
  outputFormat        : "temperature"
}

$monitorTemperatureES = Operator($temperatureES) {
  allowedLocations    : ":::::ffff:8083:c003",
  poolPreferences     : "cpu",
  concreteLocation    : ":::::ffff:8083:c003/cpu",
  inputFormat         : "temperature",
  type                : "monitorTemperature",
  outputFormat        : "alert",
  stateful            : "false",
  replicationAllowed  : "true",
  responseTime        : "0.5"
}

$availabilityDE       = Source() {
  concreteLocation    : ":::::ffff:8083:c004/general",
  type                : "availabilitySensor",
  outputFormat        : "availability"
}

$filterAvailabilityDE = Operator($availabilityDE) {
  allowedLocations    : ":::::ffff:8083:c004",
  poolPreferences     : "cpu",
  concreteLocation    : ":::::ffff:8083:c004/cpu",
  inputFormat         : "availability",
  type                : "filterAvailability",
  outputFormat        : "alert",
  stateful            : "false",
  replicationAllowed  : "true"
}

$productiondataDE     = Source() {
  concreteLocation    : ":::::ffff:8083:c004/general",
  type                : "productivitySensor",
  outputFormat        : "productivityImage"
}

$parsedataDE          = Operator($productiondataDE) {
  allowedLocations    : ":::::ffff:8083:c004",
  poolPreferences     : "gpu",
  concreteLocation    : ":::::ffff:8083:c004/gpu",
  inputFormat         : "productivityImage",
  type                : "parsedata",
  outputFormat        : "productivityData",
  stateful            : "false",
  replicationAllowed  : "true"
}
```

```
$performanceDE          = Operator($parsedataDE) {
  allowedLocations      : ":::::ffff:8083:c004",
  poolPreferences       : "cpu",
  concreteLocation      : ":::::ffff:8083:c004/cpu",
  inputFormat           : "productivityData",
  type                  : "performanceCalculation",
  outputFormat          : "performanceMetrics",
  stateful              : "true",
  replicationAllowed    : "false",
  responseTime          : "0.5"
}

$availabilityoeeDE        = Operator($parsedataDE) {
  allowedLocations      : ":::::ffff:8083:c004",
  poolPreferences       : "cpu",
  concreteLocation      : ":::::ffff:8083:c004/cpu",
  inputFormat           : "productivityData",
  type                  : "availabilityCalculation",
  outputFormat          : "availabilityMetrics",
  stateful              : "true",
  replicationAllowed    : "false",
  responseTime          : "0.5"
}

$qualityDE              = Operator($parsedataDE) {
  allowedLocations      : ":::::ffff:8083:c004",
  poolPreferences       : "cpu",
  concreteLocation      : ":::::ffff:8083:c004/cpu",
  inputFormat           : "productivityData",
  type                  : "qualityCalculation",
  outputFormat          : "qualityMetrics",
  stateful              : "true",
  replicationAllowed    : "false",
  responseTime          : "0.5"
}

$oeeDE                  = Operator($performanceDE, $availabilityoeeDE, $qualityDE) {
  allowedLocations      : ":::::ffff:8083:c004",
  poolPreferences       : "cpu",
  concreteLocation      : ":::::ffff:8083:c004/cpu",
  inputFormat           : "performanceMetrics, availabilityMetrics, qualityMetrics",
  type                  : "oeeCalculation",
  outputFormat          : "oeeMetrics",
  stateful              : "true",
  replicationAllowed    : "false",
  responseTime          : "0.5"
}

$reportDE               = Operator($oeeUK, $oeeES, $oeeDE) {
  allowedLocations      : ":::::ffff:8083:c004",
  poolPreferences       : "cpu",
  concreteLocation      : ":::::ffff:8083:c004/cpu",
  inputFormat           : "oeeMetrics",
  type                  : "reportgeneration",
  outputFormat          : "report",
  stateful              : "true",
  replicationAllowed    : "false",
  responseTime          : "2.5"
}
```

```
$temperatureDE         = Source() {
  concreteLocation     : "::::ffff:8083:c004/general",
  type                 : "temperatureSensor",
  outputFormat         : "temperature"
}

$monitorTemperatureDE = Operator($temperatureDE) {
  allowedLocations     : "::::ffff:8083:c004",
  poolPreferences      : "cpu",
  concreteLocation     : "::::ffff:8083:c004/cpu",
  inputFormat          : "temperature",
  type                 : "monitorTemperature",
  outputFormat         : "alert",
  stateful             : "false",
  replicationAllowed   : "true",
  responseTime         : "0.5"
}

$informUser            = Operator($filterAvailabilityUK, $filterAvailabilitySE,
                           $filterAvailabilityES, $filterAvailabilityDE, $temperatureUK,
                           $temperatureSE, $temperatureES, $reportDE) {
  allowedLocations     : "::::ffff:8083:c004",
  concreteLocation     : "::::ffff:8083:c004/general",
  inputFormat          : "alert, report",
  type                 : "informUser",
  outputFormat         : "message",
  queueLength          : "200"
}

$user                  = Sink($informUser) {
  concreteLocation     : "::::ffff:8083:c004/general",
  inputFormat          : "message",
  type                 : "user"
}
```

APPENDIX B

Curriculum Vitæ

Christoph Hochreiner
Vorgartenstrasse 129-143/1/6/18
1020 Wien

Born	January, 11th 1988
Email	hochreiner@infosys.tuwien.ac.at
Web	https://www.infosys.tuwien.ac.at/staff/hochreiner/

Education

09/2014 – 01/2018	Doctoral Program in Technical Sciences (Dr.tech.), TU Wien
10/2011 – 09/2014	Information Systems (Mag.rer.soc.oec), WU Wien
04/2011 – 04/2014	Software Engineering and Internet Computing (Dipl.Ing.), TU Wien
02/2008 – 04/2011	Business Informatics (Bsc), TU Wien
10/2007 – 02/2011	Software and Information Engineering (Bsc), TU Wien

Work Experience

09/2017 – 12/2017	Project Assistant, TU Wien
09/2014 – 08/2017	University Assistant, TU Wien
05/2012 – 08/2014	Researcher, SBA Research
09/2009 – 04/2012	Software Developer, SE-Flex-AS (Christian Doppler Labor)
07/2011 – 08/2011	Internship, TU Wien
03/2012 – 03/2014	Teaching Assistant, TU Wien
03/2010 – 06/2011	Teaching Assistant, TU Wien

Teaching

Lecturer

184.269 Advanced Internet Computing

Teaching Assistant

184.269 Advanced Internet Computing
188.403 Software Engineering and Projectmanagement
188.405 Advanced Software Engineering
188.407 Management of Software Projects
188.409 Requirements Engineering and Specification

Co-Supervised Master Theses

Thomas Biasion
A Network-based Business Model Framework for the Internet of Things
Thomas Hießl
Optimizing the Placement of Stream Processing Operators in the Fog
Bernhard Knasmüller
On Fault Tolerance in Stream Processing Systems

Co-Supervised Bachelor Theses

Andres Carrasco
Scalability for Data Stream Processing Frameworks
Roland Ganaus
Deployment Techniques for Dynamic Application Scenarios

Scientific Services

Organizing Committees

- 10th ZEUS Workshop (ZEUS 2018), Steering Committee
- 9th ZEUS Workshop (ZEUS 2017), Steering Committee
- 8th ZEUS Workshop (ZEUS 2017), Local Chair

Program Committees

- 6th International Conference on Emerging Internet, Data & Web Technologies (EIDWT-2018)
- International Workshop on Autonomic Solutions for Parallel and Distributed Data Stream Processing (Auto-DaSP 2017) at the 23rd International European Conference on Parallel and Distributed Computing (Euro-Par 2017)
- Workshop on IT-Governance und Strategisches Informationsmanagement (ITG-SIM) at the 47th Annual Meeting of the German Gesellschaft für Informatik 2017
- International Workshop on Performance and Conformance of Workflow Engines at the 5th European Conference on Service-Oriented and Cloud Computing (ESOCC 2016)
- Workshop on IT-Governance und Strategisches Informationsmanagement (ITG-SIM) at the 46th Annual Meeting of the German Gesellschaft für Informatik 2016
- Workshop on Additive Fertigung/3D Druck - Technologie, Auswirkungen und Chancen at the 46th Annual Meeting of the German Gesellschaft für Informatik 2016
- Workshop on IT-Governance und Strategisches Informationsmanagement (ITG-SIM) at the 45th Annual Meeting of the German Gesellschaft für Informatik 2015

Reviewer

- ACM Transactions on Internet Technology
- ACM Transactions on the Web
- Future Generation Computer Systems
- IEEE Computer
- IEEE Internet Computing
- IEEE Transactions on Services Computing
- Journal of Web Semantics
- PeerJ

Publications

Journal Papers

- C. Hochreiner, M. Vögler, S. Schulte, and S. Dustdar, "Cost-efficient enactment of stream processing topologies", *PeerJ Computer Science*, vol. 3, e141, 2017. DOI: 10.7717/peerj-cs.141
- P. Waibel, J. Matt, C. Hochreiner, O. Skarlat, R. Hans, and S. Schulte, "Cost-optimized redundant Data Storage in the Cloud", *Service Oriented Computing and Applications (SOCA)*, vol. 11, no. 4, pp. 411–426, 2017. DOI: 10.1007/s11761-017-0218-9
- C. Prybila, S. Schulte, C. Hochreiner, and I. Weber, "Runtime verification for business processes utilizing the Bitcoin blockchain", *Future Generation Computer Systems*, 2017. DOI: 10.1016/j.future.2017.08.024
- P. Hoenisch, D. Schuller, S. Schulte, C. Hochreiner, and S. Dustdar, "Optimization of Complex Elastic Processes", *IEEE Transactions on Services Computing (TSC)*, vol. 9, no. 5, pp. 700–713, 2016. DOI: 10.1109/tsc.2015.2428246
- C. Hochreiner, S. Schulte, S. Dustdar, and F. Lecue, "Elastic Stream Processing for Distributed Environments", *IEEE Internet Computing*, vol. 19, no. 6, pp. 54–59, 2015. DOI: 10.1109/mic.2015.118
- C. Hochreiner, S. Schulte, and O. Kopp, "Bericht zum 8. ZEUS Workshop", *Softwaretechnik-Trends*, vol. 36, no. 2, pp. 61–62, 2016
- C. Hochreiner, P. Frühwirt, Z. Ma, P. Kieseberg, S. Schrittwieser, and E. R. Weippl, "Genie in a Model? Why Model Driven Security will not secure your Web Application.", *JoWUA*, vol. 5, no. 3, pp. 44–62, 2014. DOI: 10.22667/JOWUA.2014.09.31.044

Conference/Workshop Proceedings

- C. Hochreiner, M. Nardelli, B. Knasmüller, S. Schulte, and S. Dustdar, "VTDL: A Notation for Stream Processing Applications (accepted for publication)", in *12th International Symposium on Service-Oriented System Engineering (SOSE)*, IEEE, 2018, NN–NN
- M. Borkowski, C. Hochreiner, and S. Schulte, "Moderated Resource Elasticity for Stream Processing Applications", in *International Workshop on Autonomic Solutions for Parallel and Distributed Data Stream Processing (Auto-DaSP 2017) at 23rd International European Conference on Parallel and Distributed Computing (Euro-Par 2017)*, 2017, NN–NN

- M. Nardelli, C. Hochreiner, and S. Schulte, "Elastic Provisioning of Virtual Machines for Container Deployment", in *International Workshop on Autonomous Control for Performance and Reliability Trade-Offs in Internet of Services (ACPROSS 2017) at 8th International Conference on Performance Engineering (ICPE 2017)*, ACM/SPEC, 2017, pp. 5–10. DOI: 10.1145/3053600.3053602

- L. Mazzola, P. Kapahnke, P. Waibel, C. Hochreiner, and M. Klusch, "FCE4BPMN: On-demand QoS-based Optimised Process Model Execution in the Cloud", in *International Conference on Engineering, Technology and Innovation / International Technology Management Conference (ICE/ITMC)*, IEEE, 2017, pp. 319–328. DOI: 978-1-5386-0774-9/17

- C. Hochreiner, M. Vögler, S. Schulte, and S. Dustdar, "Nomadic Applications Traveling in the Fog", in *2nd EAI International Conference on Cloud Networking for Internet of Things Systems (CN4IoT)*, vol. 189, Springer International Publishing, 2018, pp. 151–161. DOI: 10.1007/978-3-319-67636-4_17

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

- P. Waibel, C. Hochreiner, and S. Schulte, "Cost-Efficient Data Redundancy in the Cloud", in *9th International Conference on Service-Oriented Computing and Applications (SOCA)*, IEEE, 2016, pp. 1–9. DOI: 10.1109/SOCA.2016.12

- M. Borkowski, S. Schulte, and C. Hochreiner, "Predicting Cloud Resource Utilization", in *9th International Conference on Utility and Cloud Computing (UCC)*, IEEE/ACM, 2016, pp. 37–42. DOI: 10.1145/2996890.2996907

- C. Hochreiner, P. Waibel, and M. Borkowski, "Bridging gaps in cloud manufacturing with 3D printing", in *Informatik 2016, 46. Jahrestagung der Gesellschaft für Informatik*, 2016, pp. 1623–1626

- C. Hochreiner, M. Vögler, P. Waibel, and S. Dustdar, "VISP: An Ecosystem for Elastic Data Stream Processing for the Internet of Things", in *20th International Enterprise Distributed Object Computing Conference (EDOC)*, IEEE, 2016, pp. 19–29. DOI: 10.1109/edoc.2016.7579390

- S. Schulte, M. Borkowski, C. Hochreiner, M. Klusch, A. Murguzur, O. Skarlat, and P. Waibel, "Bringing Cloud-based Rapid Elastic Manufacturing to Reality with CREMA", in *Workshop on Intelligent Systems Configuration Services for Flexible Dynamic Global Production Networks (FLEXINET) at the 8th International Conference on Interoperability for Enterprise Systems and Applications (I-ESA 2016)*, 2016, pp. 407–413

- C. Hochreiner, M. Vögler, S. Schulte, and S. Dustdar, "Elastic Stream Processing for the Internet of Things", in *9th International Conference on Cloud Computing (CLOUD)*, IEEE, 2016, pp. 100–107. DOI: 10.1109/cloud.2016.0023
- P. Hoenisch, C. Hochreiner, D. Schuller, S. Schulte, J. Mendling, and S. Dustdar, "Cost-Efficient Scheduling of Elastic Processes in Hybrid Clouds", in *8th International Conference on Cloud Computing (CLOUD)*, IEEE, 2015, pp. 17–24. DOI: 10.1109/CLOUD.2015.13
- C. Hochreiner, "Privacy-Aware Scheduling for Inter-Organizational Processes", in *7th Central European Workshop on Services and their Composition (ZEUS)*, 2015, pp. 63–68
- S. Schulte, P. Hoenisch, C. Hochreiner, S. Dustdar, M. Klusch, and D. Schuller, "Towards process support for cloud manufacturing", in *18th International Enterprise Distributed Object Computing Conference (EDOC)*, IEEE, 2014, pp. 142–149. DOI: 10.1109/edoc.2014.28
- C. Hochreiner, M. Huber, G. Merzdovnik, and E. Weippl, "Towards Practical Methods to Protect the Privacy of Location Information with Mobile Devices", in *7th International Conference on Security of Information and Networks (SIN)*, ACM, 2014, pp. 17–24. DOI: 10.1145/2659651.2659680
- C. Hochreiner, Z. Ma, P. Kieseberg, S. Schrittwieser, and E. R. Weippl, "Using Model Driven Security Approaches in Web Application Development", in *2nd TC5/8 International Conference*, IFIP/Springer, 2014, pp. 419–431. DOI: 10.1007/978-3-642-55032-4_42
- P. Frühwirt, P. Kieseberg, C. Hochreiner, S. Schrittwieser, and E. R. Weippl, "InnoDB Datenbank Forensik Rekonstruktion von Abfragen über Datenbank-interne Logfiles", in *Sicherheit 2014: Sicherheit, Schutz und Zuverlässigkeit, Beiträge der 7. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik*, 2014, pp. 363–374

Technical Reports

- P. Hoenisch, D. Schuller, C. Hochreiner, S. Schulte, and S. Dustdar, "Elastic Process Optimization - The Service Instance Placement Problem", Distributed Systems Group, Vienna University of Technology, Tech. Rep. TUV-1841-2014-01, 2014

Proceedings

- C. Hochreiner and S. Schulte, Eds., *8th ZEUS Workshop, Vienna, Austria, January 27-28, 2016*, vol. 1562, CEUR Workshop Proceedings, CEUR-WS.org, 2016