Adapting Stream Processing Pipelines in Fog Infrastructures

Bachelor Thesis

by

Daniel Gomm

Degree Course: Industrial Engineering and Management B.Sc.
Matriculation Number: 2055065

Institute of Applied Informatics and Formal Description Methods (AIFB)

KIT Department of Economics and Management

Advisor:	Prof. Dr. York Sure-Vetter
Second Advisor:	Prof. Dr. Andreas Oberweis
Supervisor:	M.Sc. Patrick Wiener
Submitted:	October 19, 2020

Abstract

As the number of Internet of Things (IoT) devices continues growing, means of effectively deploying Internet of Things (IoT) applications become increasingly important. Fog computing can provide an appropriate platform for many of these platforms. A commonly deployed application in such a setting is stream processing. This thesis thus addresses the issue of adapting stream processing pipelines while they are running. Stream processing pipelines perform computations through a sequence of Pipeline Elements (PEs). These PEs can be deployed across different fog nodes. Adapting a stream processing pipeline thus refers to the relocation (migration) of PEs between fog nodes. In the therefore developed conceptual framework, the two sub-problems migration of a correctly running PE and migration of an interrupted PE are handled separately and are addressed with respective procedures. These procedures were implemented into Apache StreamPipes (incubating) as a proof of concept. The evaluation of this proof of concept shows that the proposed procedures perform the migration consistently and with a low PE downtime.

Contents

List of Abbreviations

BLCR Berkeley Lab Checkpoint/Restart.

CPU Central Processing Unit.

CRIU Checkpoint and Restore in Userspace.

CSV Comma Separated Values.

DLAC Dual Level Asynchronous Checkpointing.

DRAM Dynamic Random Access Memory.

E Experiment.

GUI Graphical User Interface.

IO Input/Output.

IoT Internet of Things.

MiB Mebibyte (1024^2 Byte).

OS Operating System.

PE Pipeline Element.

REST Representational State Transfer.

RQ Research Question.

SDK Software Development Kit.

URL Uniform Resource Locator.

VAS virtual address spaces.

VM Virtual Machines.

List of Figures

List of Tables

1 Introduction

While the concept of the Internet of Things (IoT) has been around for about 20 years [22], mainstream attention and adoption have just recently seen a steep increase. Between 2012 and 2017, the annual search interest on the IoT has increased more than nine-fold, according to data from Google (see Appendix A.1). In the same timeframe, industrial adoption has been growing as well. According to an industry report commissioned by Microsoft [23], 91% of the companies contacted were in the process of adopting IoT technologies in 2019. In addition, 64% of the companies reported plans to further expand their IoT ventures in the next two years.

In parallel with this rise in interest and industrial adoption, the number of connected devices is expected to more than quadruple between 2015 and 2025, reaching more than 75 billion at the end of 2025 [2]. With this ever-growing number of connected devices, the significance of effectively deploying these devices in IoT applications is growing as well. For effective deployment, several requirements must be fulfilled. These prerequisites have been outlined by Yigitoglu et al. [43]. They are based on latency, bandwidth, resource constraints, mobility, dynamic workloads, multi-tenancy, and privacy. To deploy IoT devices and applications in line with these requirements, neither a solely cloud-based approach nor a solely edge-based one is appropriate [44]. Thus, a combined approach of edge and cloud computing is needed.
The fog computing paradigm addresses this combination, extending cloud computing to the network edge [4]. The capabilities and characteristics of edge computing (e.g., low latency, real-time computing, etc.) and cloud computing (e.g., an abundance of computing resources, etc.) are combined in fog computing, making it a well-suited approach for most IoT applications [44].
Many tasks in the IoT context run continuously and can be divided into several consecutive steps. One way to address such a continuously running task comprised of a sequence of steps is through stream processing. In stream-processing, an input data stream is processed in a sequence of actions, forming a so-called stream processing pipeline. Each of these actions is executed by an action-specific processor, referred to as Pipeline Element (PE). Besides the IoT context, stream processing is used in a variety of applications, including electronic trading, fraud control, and infrastructure monitoring [39].
Using fog computing and stream processing in combination can thus address commonly encountered tasks in the IoT within an IoT compatible frame. However, the deployment of stream processing applications in the fog comes with some challenges. To visualize these challenges, two exemplary cases are introduced. These cases also serve to illustrate the underlying technology, the addressed problems, and the developed conceptual framework in the subsequent chapters. While the first case is abstract and serves as an easy-to-follow example constructed for this thesis, the second case represents a real-world use case and

is a simplified adoption for stream processing in the fog of the work of Chowdhurry et al. [9]:

Case 1 The first case is a simple example of a stream processing pipeline, which aggregates values. As depicted in Figure 1, the stream processing pipeline gets its input data stream from a text file that contains numeric Comma Separated Values (CSV). The second PE aggregates incoming values over a predefined window of past events and passes on the result. In the last step, this data stream of values is written to a CSV output text file. Using this stream processing pipeline, the result's consistency can be monitored, and the functionality can be tested.

Case 2 The second case is based on a medical wearable device that monitors a patient's electrocardiogram. Early symptoms of heart attacks are detected using the acquired electrocardiogram data. Therefore, the sensory data needs to be classified using a neural network, which can trigger an alarm on the patient's wearable device. In detail, this application consists of 5 steps, as shown in Figure 1. First, sensors acquire the data. Next, the signal is amplified, then it is filtered before the classification happens. Finally, an alarm on the wearable device is triggered when symptoms are detected.

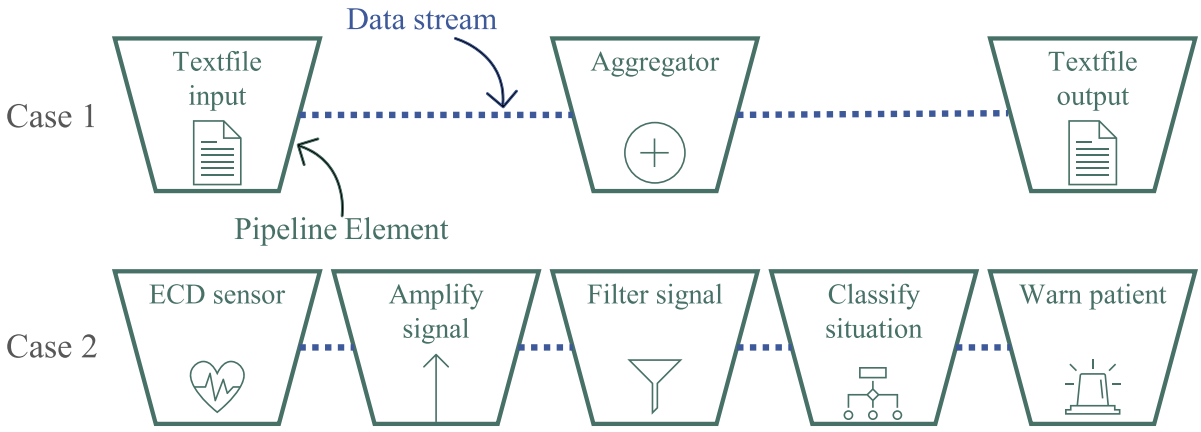


Figure 1: Stream processing pipelines of the exemplary cases

The application in Case 2 requires a notable amount of computing resources for the classification of the sensory information, while its overall computing time should be as small as possible to alarm the patient in time to take necessary actions. At the same time, the wearable device has only limited computational capabilities and cannot provide sufficient service quality in the classification step. To cater to these needs, the task is realized as a stream processing pipeline deployed in the fog. Fog computing is utilized by deploying the PEs individually and distributing them across fog nodes (different computing devices

in the fog). To achieve a low latency, the PEs for the computationally intensive steps are deployed on resourceful nodes with low latency in the close vicinity. The PE that inputs the electrocardiogram data into the stream processing pipeline, and the PE that triggers the alarm are deployed on the wearable device. Due to wearable devices' mobile nature, the computing nodes in the close vicinity are continually changing, thus requiring a possibility to relocate the pipeline's components while the pipeline is running. This dynamic relocation is referred to as workload-mobility [43]. Workload-mobility can also be important in the case of a resource scarcity at a node, which bottlenecks the application's computing time, and therefore imposes a significant toll on the application's usefulness. Additionally, in a situation in which a fog node hosting a PE becomes unreachable, for example, because its network connection is interrupted, the PE has to be relocated as well to facilitate the continuation of the pipeline's execution.
To address these challenges and, more broadly, all the previously introduced requirements outlined by Yigitoglu et al. [43], it is necessary to adapt stream processing pipelines while they are running by relocating PEs. This relocation is referred to as migration. Migration means that processing previously performed on a specific node is shifted to another node, which takes over processing.

This thesis focuses on developing a conceptual framework for the migration of stream processing applications in the fog. More precisely, this thesis focuses on the following research questions (RQ):

RQ1 How can stateless and stateful event-driven PEs be migrated between nodes in the fog?

RQ2 How can the data stream's consistent processing be facilitated when an event-driven PE is migrated between nodes in the fog?

By addressing these research questions, this thesis fills gaps in existing research. To the best of the author's knowledge, there has not been any prior research into the migration of stateful stream processing applications that considered the fog computing paradigm's specific challenges and opportunities.
Therefore, this thesis takes on this topic by developing a conceptual framework for the migration of PEs that also takes the specific challenges and opportunities of fog computing into account. Moreover, this thesis proposes separate migration tactics for different situations.

After this brief introduction and motivation, the second chapter investigates the enabling technologies for this thesis, giving a high-level overview of fog computing and stream processing, emphasizing the role of state in stream processing. Thereby, chapter 2 lays the groundwork for the subsequent chapters. The third chapter introduces related work. It

contemplates related works that assess the migration of applications in the fog and works that concern state in data processing. These works propose differing approaches for the migration of applications and for state management. In chapter 4, these approaches are compared, and the requirements for the migration of stream processing PEs in the fog are outlined. However, the chapter's main focus is on the development of a conceptual framework for the migration of PEs. The developed conceptual framework handles the migration of running and interrupted PEs distinctly. For running PEs, *PE live migration* is proposed, interrupted PEs are relocated using the *PE restoration*. The *PE restoration* relies on centrally available PE state checkpoints. To acquire these checkpoints, dual-level asynchronous checkpointing is introduced. As a proof of concept, the developed conceptual migration framework is then implemented into Apache StreamPipes (incubating) in chapter 5. This chapter provides an in-depth overview of the concept's implementation and outlines implementation details of modifications and extensions of StreamPipes, presupposed by the conceptual framework. In chapter 6, the proof of concept implementation is evaluated in several experiments to assess whether and to which extent the requirements for the conceptual framework are met. The chapter is closed by pointing out the limitations of the thesis. Finally, the last chapter summarizes the previous chapters' key points while outlining possible directions for further research complementing this thesis.

2 Background

This chapter introduces fog computing and stream processing. A profound understanding of both of these topics is needed as background for the subsequent chapters. Thus, fog computing is thoroughly introduced, defined, and its intricacies are pointed out. Furthermore, stream processing is introduced together with the concept of state in the stream processing context. State is then examined in detail, and relevant aspects in the handling of state are pointed out.

2.1 Fog Computing

There are various definitions of the fog computing paradigm. In this thesis, the definition of Bonomi et al. [4] is used. They state that "fog computing extends the cloud computing paradigm to the edge of the network" [4]. They further describe fog computing as a highly virtualized platform that connects end devices to traditional cloud services by providing computing, storage, and networking services in between.
This definition of fog computing includes all available connected devices, meaning that compared to cloud computing, which typically solely utilizes large data centers [44], fog computing takes advantage of small servers, routers, switches, gateways, set-top boxes, and other small computing devices [44] as well as the cloud infrastructure, which remains a core component.

To give a more comprehensive overview, the characteristics that portray fog computing are presented in the following listing:

- Since a lot of the devices used in fog computing occupy a relatively small space, this hardware can be located close to the network edge [44], allowing for **low latency** and **location-aware computing** [4, 43]. In Case 2, hardware close to the edge, and location-aware computing is needed to allow a low latency.

- The fog is comprised of many different devices with vastly varying capabilities. This makes the **heterogeneity of devices** [4] another characteristic of fog computing. For example, in Case 2, the pipeline is deployed on a wearable device with limited computational capabilities and devices in the close vicinity that possess significantly more computational capabilities. These edge near devices host the more computationally intensive PEs.

- A **large number of nodes** makes **privacy** a priority [43]. Since a large amount of data is collected (especially in the IoT) and is then distributed for processing, security must be provided at the edge to ensure that sensitive data is not shared with unwanted resources [43]. At the same time, fog computing can enhance privacy.

For example, production data in an industrial IoT application can be preprocessed in the company's local network before sending it to cloud servers performing the rest of the computation.

- **Interoperability** between fog nodes is needed since many applications are deployed across multiple fog nodes [4]. This is especially prevalent in stream processing applications since when a stream processing pipeline is deployed across multiple fog nodes, their correct interplay is crucial for a consistent operation.

- Devices in the fog are usually **geographically distributed** [4] and connected to the network predominantly through wireless **access** [31, 4, 3]. The geographic distribution is exploited in Case 2 to lower the computation time by only deploying PEs on fog nodes in the close vicinity with a low latency.

As presented, the heterogeneity of devices is a characteristic of fog computing. With the heterogeneity of devices comes a heterogeneity of resources, architectures, and operating systems [35]. Addressing this issue, virtualization technologies are utilized to provide a more generic environment for applications on fog nodes [35]. Hence, it is necessary to look at commonly deployed virtualization technologies.
The two main approaches taken for virtualization are hardware virtualization in the form of so-called Virtual Machines (VM) and operating system level virtualization, better known as containerization [35]. Sharing the kernel with the operating system makes containerization more lightweight than virtual machines, which leads to containerization being the more appropriate solution in regards to the overall footprint and support of workload mobility [35]. Containerization is particularly vital in fog computing since processing is not only performed on devices with an abundance of computing resources but also on smaller devices that only provide moderate computing resources at low power consumption [24]. In many cases, this makes it difficult or even impossible to deploy virtual machines. The choice between VMs and containers is context and need dependent, but due to the described factors, containerization has become the reference technology for fog computing in the last years [34].

Fog computing is used in a multitude of different use cases [44], mostly in combination with IoT devices. These use cases include connected vehicles, health care, smart city/smart grid, video surveillance, and industrial IoT [44]. In these cases, using fog computing can address volatile workloads, resource constraints, latency requirements, and privacy concerns [43].

2.2 Stream Processing

Stream processing links a collection of parallelly computing PEs through channels [38]. These systems of linked PEs are called stream processing systems [38] or stream processing pipelines. In stream processing pipelines, data is passed in between PEs in the form of a data stream, which is a list of distinct data elements sourced from a data set of interest with indefinite length [38]. The data in a stream could, for example, be stock prices, postings in social networks, or sensory data [20] like the electrocardiogram data in Case 2.
Typically, stream processing systems consist of three typically differentiated types of PEs. This differentiation is based on their in- and output configuration. Distinguished are:

- PEs that have no inputs. They are called sources [38] and mark an entry point for data into a stream processing pipeline. Sources pass data into the system where this data is further processed. In Case 2, the source is the PE that inputs the sensory electrocardiogram data into the system.
- Filters (also called agents or processors) that take input data, perform computation on it, and output the result [38]. Examples for these intermediary processors are the amplifying, filtering, and classifying PEs from Case 2.
- PEs that only possess inputs. They are called sinks [38] and mark the end of a pipeline. Sinks use the incoming data to interact with entities outside of the stream processing pipeline. For example, the final PE in Case 1 outputs events to a CSV file, and the final PE in Case 2 alarms the patient.

Filters can be further categorized into deterministic and non-deterministic filters [38]. While deterministic filters follow a functional specification that produces a predictable outcome, non-deterministic filters do not have a predictable outcome.
Furthermore, PEs can be categorized into stateful and stateless PEs [7]. The difference between these types of PEs is extensively discussed in chapter 2.2.2.

2.2.1 Event-Driven Stream Processing

An event-driven stream processing pipeline is a particular form of a stream processing pipeline. It consists of a number of event-driven PEs, which operate on particular data elements that are called events. In principle, every notable thing can be an event [27]. An event consists of an event header and an event body [27]. While the event header describes the occurrence of the element (with a timestamp, event creator, etc.), the event body represents what happened [27].

Looking at Case 2, an event passed into the pipeline by the source PE might look like the illustration in Figure 2.

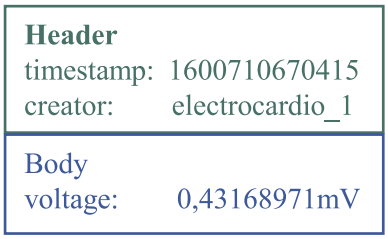


Figure 2: Exemplary visualization of an event form Case 2

Generally, in a system that follows an event-driven architecture, the occurrence of an event entails an action [27]. In event-driven stream processing, this action usually is the processing of the event in an event-driven pipeline consisting of event-driven PEs. This structure of event-driven components facilitates the loose coupling and distribution of components [27]. Event-driven PEs thus act as reusable, independent event-driven functions.
The following chapters address event-driven stream processing. For better readability, the event-driven PEs and stream processing pipelines are referred to as PEs and pipelines from here on.

2.2.2 State in Stream Processing

One major categorization for PEs is whether they possess a state, referred to as stateful, or if they do not possess a state, referred to stateless [7]. Usually, this differentiation refers to PEs whose outputs solely depend on the current event as stateless and to the PEs whose outputs additionally depend on past events as stateful. The concept of state is assessed to more precisely differentiate these categories of PEs.

To et al. [40] define the state as "the intermediate value of a specific computation that will be used in subsequent operations during the processing of a data flow". When considering stream processing, each PE can have its own state. The most commonly used abstraction of state is the so-called operator state [40], which describes the state from the PE's perspective [40]. The operator state consists of the processing, buffer, and routing state [40]. The *processing state* is present in all PEs, whose output depends on the current input and the past input [7]. It is composed of a set of values that have been obtained through past computation. To interact with the processing state from an outside perspective, it has to be exposed. Exposing the processing state can be achieved by storing it in mutable data structures [6], or by providing an interface to the in-memory processing state.
It is a common practice to keep output buffers between PEs in stream processing to

compensate for temporary network and stream rate problems[7]. The *buffer state* accounts for the events which have been processed by a PE but have yet to be processed by the ensuing PE [7]. In this thesis, the concept of buffer state is extended to not only account for the output buffers but also for the positional information about the event that has most recently been processed by the PE. Thus, the positional information represents how far processing has progressed in the event-stream, linking the operator state to an exact point in event-processing.
Finally, the *routing state* holds the information on how the PE's output events should be routed to the following PEs [7]. The routing state bears particular importance when the output destination can change at runtime.

Using the abstraction of operator state, so-called stateless PEs do possess a buffer and a routing state but lack a processing state. Only the so-called stateful PEs possess a processing state as well. This means that using the abstraction of state as operator state, the terms stateful and stateless specifically refer to the processing state.

Management of State

The effective management of state requires a multitude of operations on the state [40]. In the following, a few particularly essential operations are described.
A key aspect of state handling revolves around the question of how to *store and purge state* effectively. State can be stored either in-memory or in persistent storage [40]. While storing in-memory yields performance improvements [41], the stored state is not persistent and therefore does not survive system restarts. Somewhat of a middle way is taken by Apache Flink [1], when using RocksDB as so-called "State Backend", since RocksDB [15] inserts new writes into an in-memory data structure, which is flushed to a file on storage when it is filled up.
To prevent an expired state from taking up storage capacity, the data can be purged [40], which is another operation in state management that can be implemented into a system. The *migration* of the PE's state is an operation that is essential to the migration of the respective PE. This operation can be implemented in various ways with different strengths and weaknesses. The following chapter discusses these different tactics in greater detail.

Checkpointing State

In the context of stateful stream processing, a checkpoint refers to a backed-up intermediate result of a PE, from which processing can be recovered [45]. The checkpoint describes the state of the checkpointed entity at a specific point in time.

The checkpointing process can either be coordinated globally across the whole pipeline or independently for every PE [16]. Secondly, it can be distinguished whether checkpointing is performed synchronously or asynchronously [16]. These categorizations result in the following commonly used checkpointing tactics:

- In **Synchronous global checkpointing**, a consistent snapshot of all PEs involved in the pipeline is obtained. The checkpoint contains the individual states of the affected PEs, all representing the same point in time. This can be achieved, for example, by using a "stop-the-world" approach [28, 16], where incoming data is stopped, remaining data is processed, and after that, the states of all involved entities are captured.

- **Asynchronous global checkpointing** also achieves a snapshot across all involved nodes, but it allows the checkpointing of nodes at different times [16]. A famous example of this process is the Chandy-Lamport algorithm [8]. The thereby acquired checkpoints must, in addition to the PE state checkpoints, account for the processing related state changes that happened during checkpointing [8].

- In **asynchronous independent checkpointing**, the state of single PEs is captured independently of other PEs from the same pipeline. Thus, the checkpoints of the PEs comprising the pipeline do not necessarily provide a consistent global state [16].

3 Related Work

For a deep dive into related work, this chapter focuses on literature concerning the migration of applications in the fog and work regarding state-management in stream processing applications.

3.1 Migration of Applications in the Fog

Related works concerning the migration of applications in the fog can be categorized as depicted in Figure 3. While some publications perform the migration directly integrated into applications (application level), most focus on the migration on the virtualization level. On the virtualization level, either containers or VMs are migrated. The container migration approaches are further categorized into checkpoint/restore-based approaches, storage-based approaches, and other strategies.

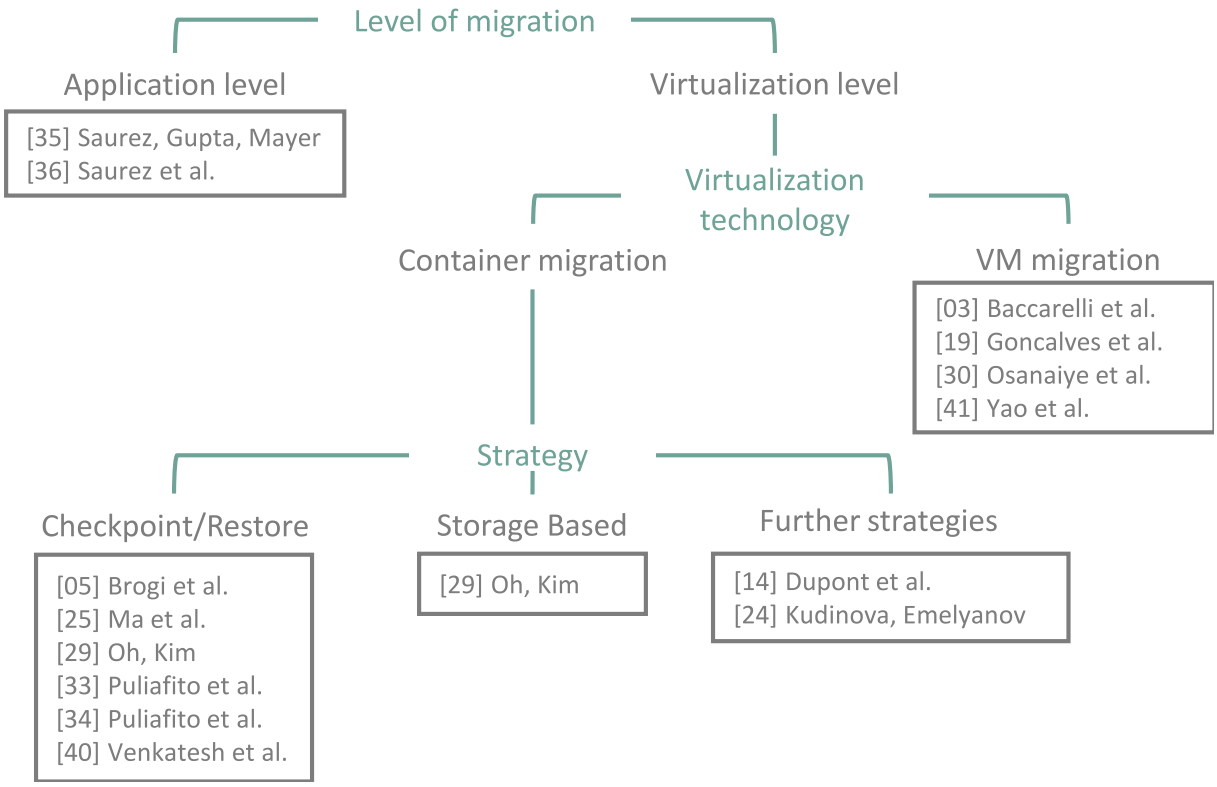


Figure 3: Overview of related works regarding application migration in the fog

3.1.1 Migration of Virtual Machines

Since the migration of VMs has already been extensively studied in traditional cloud computing [42], works with the subject of VM migration in the fog mainly focus on the optimization of already established migration procedures for fog-specific use cases.

Forsman et al. [17] give an overview of the three general approaches for VM migration. Cold migration shuts down the guest Operating System (OS) and moves it to a new host, where it is restarted. Hot migration does not shut down the guest OS but instead suspends it before moving it to another machine, where it is resumed. Hot migration thereby circumvents the termination of applications. Instead, they continue running on the new host. The third approach is the so-called live migration, which moves a VM while it is still running on the original host. The downtime of applications migrated between fog nodes is thereby vastly reduced. Because of the reduced downtime, live migration is the preferred approach for VM migration in the fog [3, 31]. Both Osanaiye et al. [31] and Baccarelli et al. [3] utilize so-called pre-copy live migration. Pre-copy live migration transfers memory contents in several iterations between the former and the new host [31]. In contrast, post-copy live migration initially sends the whole virtual processing unit and the device state to the new host, allowing the new host to fetch memory pages on demand [31]. While Osanaiye et al. [31] focus on minimizing the downtime by evaluating whether to proceed with the stop and copy phase of the migration, Baccarelli et al. [3] optimize the decision-making process over the question if a VM should be migrated to ensure low energy consumption over wireless 5G connections.
The work of Yao et al. [42] also focuses on the decision if the migration is desirable, while at the same time deciding on an appropriate target node for the migration, optimizing the decision in terms of low overall network cost.
Lastly, Goncalves et al. [19] explore a simulation-based mobility prediction strategy to proactively migrate VMs, with the intent of minimizing the latency. Using mobility prediction, they achieve a decrease in the number of performed migrations, resulting in a lower time of unavailability.

3.1.2 Migration of Containers

As shown in Figure 3, there are different approaches to the migration of containers. In [14] the straight-forward solution of an orchestrated container destruction on the origin node, followed by container re-deployment on the target node, is proposed. This approach, however, does not allow the migration of stateful applications.
To allow the re-deployment of containers with their respective states, many authors concentrate on concepts exploiting Checkpoint and Restore in Userspace (CRIU), which resembles the live migration approaches for VMs mentioned before. CRIU is a Linux software that can freeze, checkpoint, and restore running containers [12]. After an application's restoration, it resumes running precisely at the point it was frozen [12]. Puliafito et al. [35] conduct a detailed comparison of migration approaches that use CRIU. The compared container migration techniques are mostly similar to the approaches for VM migration. The first described possibility is cold migration, which they characterize as the

iterative approach of stopping the container, dumping its state, transferring the dump to the target node, and resuming the container there. In contrast, they also describe live migration techniques, where the container keeps running while most of the state is transferred. Pre-copy, post-copy, and hybrid migration are differentiated. They all use multiple transfer steps reducing the downtime to a fraction of the total migration time. The pre-copy technique uses an initial pre-dump that is transferred to the target node. After transmission, the container on the origin node is stopped, and the modified state is dumped and transferred to the target node, where the container is resumed. In comparison, post-copy directly stops the container on the origin node but transfers only the minimal task state required to start the container [11]. After the transfer, the container is resumed on the target node, where it requests missing memory pages from the origin node whenever the resumed container tries to access memory pages that have not yet been transferred. Lastly, hybrid migration combines pre-copy and post-copy migration techniques. According to the authors' performance evaluation, the post-copy and the hybrid approach have a comparably low downtime. However, these approaches also suffer from a degraded service performance because of the necessity to request missing memory pages from the origin node. Additionally, failures during the migration are more severe since the up-to-date container state is distributed between the origin and the target node. In practice, [5] and [30] show the general possibility of migrating a container using CRIU. Brogi et al. [5] additionally point out that the integration of CRIU into containerization technology can be a possible limitation. As of now, there are still compatibility issues between CRIU and the containerization software Docker [10], which highlights the actuality of the issue.

These results are complemented by publications that introduce frameworks for the dynamic triggering of the migration. The authors of [34] propose a platform that monitors position and resource constraints. Based on these factors, the platform triggers a container migration via CRIU.

Another focus of research is the improvement of the checkpoint and restore functionality itself. Ma et al. [26] approach the issue by leveraging the layered storage architecture of docker containers to reduce the transferred file systems size. Venkatesh et al. [41] take a different approach. By avoiding expensive file system writes through the usage of Linux kernel support for multiple individual virtual address spaces (VAS), they get a modified VAS-CRIU that stores memory snapshots in Dynamic Random Access Memory (DRAM), thereby vastly accelerating the snapshotting process.

An interesting contribution concerning the container live migration is by Kudinova and Emalyanov [25]. They address the problem of faulty process state restorations that can occur when a containers process tree is restored from a checkpoint. They state that this problem can also arise when restoring from a checkpoint using CRIU. In their paper, they

develop a mathematical model for the operation sequence needed to restore the process tree correctly.
Another approach to container migration presented by Oh and Kim [30] is the so-called storage-based migration. This approach uses a universally accessible storage volume on which the state of the origin container is stored. During the migration, the storage volume is detached from the origin node and attached to the target node, which uses the saved state on the persistent volume to restore the container's state after it is started. Since their paper focuses on the advantages of checkpoint-based migration (using CRIU) over storage-based migration, they also point out the limitations of storage-based migration. In storage-based migration, the deployed containerized applications need to be prepared to save and restore the state. Additionally, this migration type comes with a heavy storage access dependency and high storage provisioning cost, limiting their usefulness in fog computing.

3.1.3 Explicit State Management at Application-Level

Integrating explicit state management directly into deployed applications, the application level approaches fundamentally differ from the virtualization level approaches. Addressing the migration at application level allows specifically addressing application specific requirements at the expense of a less generalized solution.
In [37] and [36] the authors propose the programming infrastructure Foglets, which is aimed at geologically-distributed computing on fog and cloud nodes. Foglets allows for so-called Worker Processes, which execute the deployed applications computation, to be migrated between fog nodes. Each of the Workers running on a node implement two handlers that expose the state of the Worker. The first handler captures the volatile state on the origin node and assures that related computations are completed and saved. The second handler uses this volatile state as input to initialize the local state of the target node. Simultaneously with the execution of these two handlers, persistent data, stored in a RocksDB database, is migrated to the target node. These inner-workings of Foglets are outlined in the first publication of Saurez et al. [37]. In a follow-up publication on Foglets [36] the authors demonstrate Foglets capability to migrate a stream-processing application's state across nodes, supported by contextual information.

3.2 State in Data Processing

The second category of related works considered in this thesis are publications looking at the role of state in data- and, more specifically, stream-processing. The considered works are further categorized in Figure 4. While there are fundamental works that look at the grand topic of state management, most others focus on the actual usage of state in their

respective systems. In these publications, state is used in migration, for fault tolerance and/or for system scalability. Additionally, the paper of Ouyang et al. [32] follows an entirely different approach and therefore has to be distinctly classified.

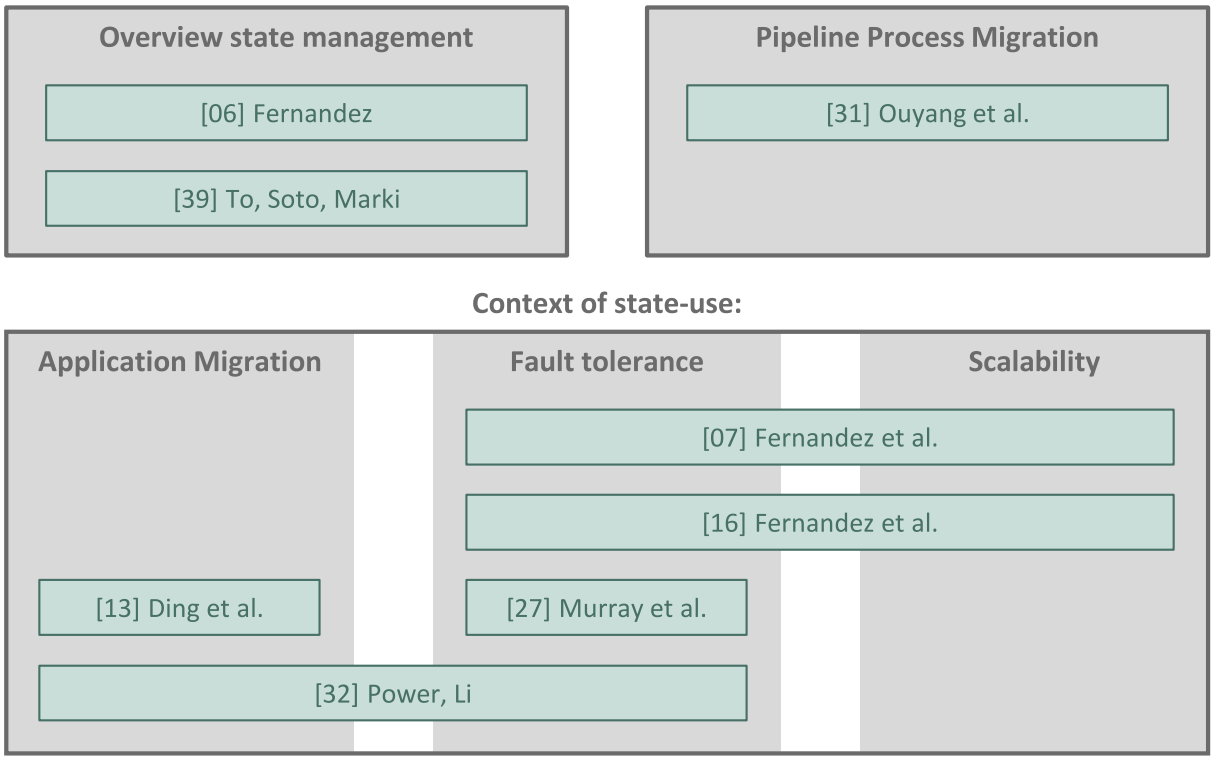


Figure 4: Overview over related works regarding state in data processing

In [6] and [40] state management is investigated on a general level. While [40] surveys state management in big data systems, [6] addresses insufficient state management as the main shortcoming of modern data-parallel processing systems, therefore diving deeply into the issue of state.

One operation that requires the correct management of an application's state is scaling. Scaling refers to the parallelization on operator level to improve throughput in computationally expensive operators to avoid bottlenecks [7]. Fernandez et al. [7] enable scaling by exposing the processing state. They model the processing state as a set of key/value pairs. To expose the state, developers have to implement a get-function for each developed operator, locking all internal data structures while taking a copy of them. In addition to scaling, the publication focuses on fault tolerance. Fault tolerance means the capability to recover from faults that occur during the application's execution. The authors achieve fault tolerance by asynchronously checkpointing the operator state of the operators to upstream nodes. These checkpoints are used to restore failed operators by setting the processing state, with the help of a developer implemented set-function. After the processing state is restored, unprocessed events are replayed from the buffer state, resulting in an up-to-date processing state. The same authors also published another paper [16]

taking a similar approach to achieve fault tolerance and scalability. In this work, they once again use asynchronous checkpointing. Furthermore, they minimize the disruption in processing during checkpointing by using so-called dirty state. The dirty state of a PE is acquired by checkpointing a PE while it continues processing events. In addition, they propose a concept for distributing state between processing nodes.
In contrast to this asynchronous checkpointing tactic, other authors utilize synchronous global checkpointing using a "stop-the-world" approach [28] and asynchronous global checkpointing using the Chandy-Lamport algorithm [8, 33] to provide checkpoints for fault-tolerance.
The third context in which state is used in related publications, is for the migration of PEs. Ding et al. [13] propose a migration mechanism that performs a progressive migration of the operator state. It allows processing while the operator state is migrated, achieving a reduction in downtime. Furthermore, they outline challenges arising due to the migration. They identify the impossibility of parallely executing and migrating a task, the possibility that the origin node might still receive inputs after the migration, and issues in the transmission of operator states as main challenges. In comparison to this work, the authors of [33] base their migration concept on shared and distributed state. This makes the migration of data processing applications trivial since a consistent state is prevalent at all nodes. However, this design comes at the expense of high networking and computation costs since all changes to the operator state need to be committed [13], hindering the usefulness in fog infrastructures.
In contrast to these approaches, Ouyang et al. [32] follow a similar concept for the acquisition of the state to the checkpoint/restore container migration approaches presented in chapter 3.1.2. However, while in these approaches, the whole container with its running applications is migrated, in [32], only the processes, which comprise the application, are migrated by using Berkeley Lab Checkpoint/Restart (BLCR). BLCR is a tool that can create checkpoints of running processes and can restart the processes from the checkpoints. Firstly, all processes are suspended and checkpointed, then the process images containing the checkpoints are transferred to the migration target node, where they are then restarted and reconnected to a stream processing pipeline. While this approach shares many characteristics of the checkpoint/restore container approaches, it performs the checkpointing and restoring actions on the application level, instead of the virtualization level. This makes the approach more versatile since singular processes can be migrated. The authors achieve a significant performance improvement by utilizing remote direct memory access, which reduces networking overhead and data Input/Output (IO). Requiring a an InfiniBand connection between nodes makes the approach proposed by the authors a poor fit for the fog infrastructure.

3.3 Research Gap

Table 1 gives a comprehensive overview of the approaches presented in the related work. Using this overview, the research gap can be conclusively outlined.
Altogether, the works regarding the migration of applications in the fog do not sufficiently address stream processing. In addition, they solely focus on the migration of correctly running applications, neither providing a possibility to migrate a faulty application nor proposing any measures to provide fault tolerance.
On the other hand, approaches from the "state in data processing" field do not address fog computing specific requirements.
Out of the twelve compared works only three ([34, 30, 13]) publications mainly focus on the migration process. This lack in research is also pointed out by Ding et al. [13], stating that other works often lack a precise description of the used migration tactics.
Moving forward, this thesis thus should fill these gaps, by proposing an extensive conceptual framework for the adaptation of stream processing pipelines in fog infrastructures. While doing so, the contents of the migration and the exact procedure should be described in detail.

Table 1: Comprehensive overview of related work

Publication	**Migration**	**Fog Considerations**	**Fault Tolerance**	**State Abstraction**	**Checkpointing Mechanism**	**Intended Use**	**Main Focus of Publication**
Baccarelli et al. [3]	✓	✓	✗	VM checkpoint	VM checkpointing	general	optimize energy usage
Osanaiye et al. [31]	✓	✓	✗	VM checkpoint	VM checkpointing	general	optimize migration
Puliafito et al. [34]	✓	✓	✗	container checkpoint	CRIU	general	migration
Brogi et al. [5]	✓	✓	✗	container checkpoint	CRIU	stream processing	fog node management
Oh, Kim [30]	✓	~	✗	container checkpoint	CRIU	general	migration
Ma et al. [26]	✓	~	✗	optimized container checkpoint	top level container storage	general	optimize migration
Dupont et al. [14]	~	~	✗	-	-	IoT functions	orchestration
Saurez et al. [37]	✓	✓	✗	operator state	interface in application	general	orchestration
Ouyang et al. [32]	✓	✗	✗	process checkpoint	BLCR	stream processing	optimize migration
Ding et al. [13]	✓	✗	✗	operator state	interface in application	stream processing	migration
Murray et al. [28]	✗	✗	✓	shared global state	interface in application	stream processing	orchestration
Power, Li [33]	✓	✗	✓	shared global state	Chandy-Lamport	data processing	orchestration
Fernandez et al. [7]	✗	✗	✓	operator state	interface in application	stream processing	fault tolerance
This Thesis	✓	✓	✓	*operator state*	*interface in application*	*stream processing*	*migration*

4 Methodology

In this chapter, a conceptual framework for the migration of a PE is proposed. The context of the migration is assessed first. Then the underlying requirements are outlined. After that, a conceptual framework satisfying these requirements is developed. Distinct approaches are proposed for the *PE live migration* and the *PE restoration*.

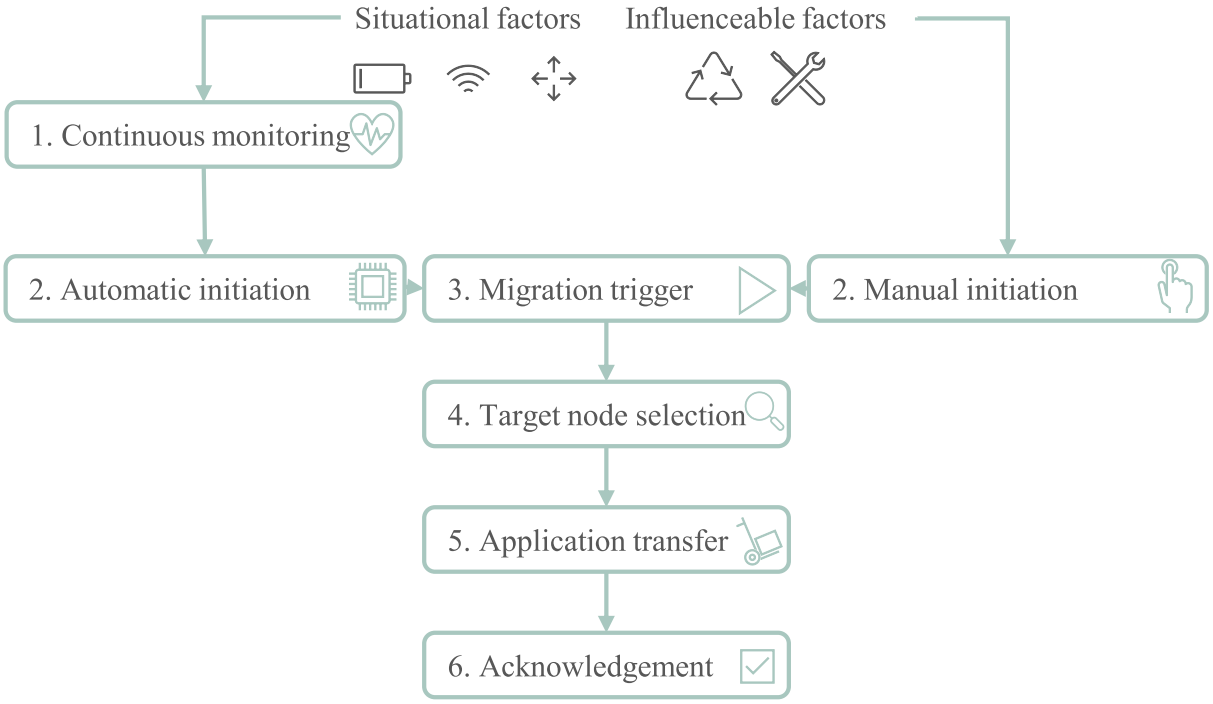


Figure 5: Context of the migration procedure

Figure 5 depicts the context of the migration procedure. The migration can either be initiated to react to situational factors, which can be constantly monitored or on demand to react to influenceable factors. Table 2 shows examples for these factors. Before moving the application away from the origin node, a target node must be selected. The optimization of this selection process is the subject of many publications regarding application migration in the fog [19, 42, 34, 37], but it is not further discussed in this thesis. Instead, the focus is on the next step, the transfer of the PE between fog nodes. This relocation process has the goal of moving a PE from its *origin node* to a specific *target node*. Subjects of the relocation are orderly *running PEs*, which must be stopped on the origin node and resumed on the target node and so-called *interrupted PEs*. PEs are interrupted when they have become unavailable, for example, due to network outages, network instability, or because they have encountered faults while processing events. These PEs need to be removed from the origin node and restored on the target node. The migration is finalized by acknowledging its success, after the PE is relocated.
In the thesis, the relocation process in step 4 is referred to as migration.

Table 2: Examples for situations in which migration is beneficial

	Factor	**Need for migration**
Situational factors	Resource load on node	Prevent resource overload
	Position of node	Reduce latency by computing on nodes in the close vicinity
	Node availability	Applications need to be redeployed if the origin node has become unavailable
	Remaining battery charge	Reduce load on low battery devices to reduce battery drain
Influenceable factors	Scheduled maintenance of nodes	Maintenance of node leads to unavailability of node for the application
	Planned usage of nodes	If node is scheduled for a different purpose in the future it will not be available for the application any longer

4.1 Requirements

As a guideline for developing and evaluating the conceptual migration framework, requirements on this framework are outlined in this section.
The requirements are derived from the research questions and further expand on them. Requirements can be categorized into functional and non-functional requirements. Functional requirements describe what the proposed conceptual framework should deliver. Non-functional requirements describe how the proposed conceptual framework should deliver its functionality.

Functional Requirements

Basic Migration The conceptual framework shall allow stateful and stateless PEs to be migrated from an origin node to a target node.

Recovery The conceptual framework shall support the migration of running PEs and interrupted PEs.

Problem Handling Possible problems shall be handled to facilitate continued processing.

Non-Functional Requirements

Consistency The resulting output-stream of the migrated PE shall be consistent with the output-stream that would have been generated if no

	migration had happened. The optimal scenario is one in which no events are lost, and no events are processed a second time, a so-called exactly once semantic [21].
Downtime	The time in which event processing is disrupted (referred to as downtime) shall be minimized.
Development	An additional complication in PE development due to the introduction of PE migration shall be minimized.
Fog Compatibility	The application shall be deployable in the fog, which means that the fog-specific requirements resource constraints [44] and varying networking conditions [34] have to be considered.

The proposed requirements define the nature of an optimal migration procedure. The goal of the conceptual framework developed in this thesis is to satisfy them as much as possible.
As mentioned these requirements are derived from the research questions. The *Basic Migration* and the *Recovery* requirements are derived from RQ1. The *Consistency* and the *Problem Handling* requirements are extracted from RQ2. As it describes the overall setting considered in this thesis, the *Fog Compatibility* requirement was derived from both requirements. Lastly, the *Downtime* and the *Development* requirements expand on the research questions.

Looking at the requirements, a particularly important implication for the conceptual framework can be derived from the *Recovery* requirement: The developed conceptual framework should allow the migration of interrupted PEs, meaning PEs that encountered faults or are unavailable. Migrating these PEs means restoring their state, which is only possible if state checkpointing measures are implemented. These measures need to provide the possibility to restore a PE's state even if the origin node is unreachable (e.g., in the case of a power outage).

4.2 Conceptual Migration Framework

Using the requirements, a conceptual framework for the migration is developed in this section. The conceptual framework is proposed for a fog computing application that consists of a centralized orchestrator and an undefined number of PEs deployed on computing nodes in the fog. The centralized orchestrator controls the deployment and the overall life-cycle of all pipelines in the application.

As a first step, the different existing solutions presented in chapter 3 have to be assessed regarding which category of approaches best fits the requirements at hand. Commonly

proposed approaches are *VM live migration*, *container migration using checkpoint/restore functionalities*, and *explicit state management at the application level*. This assessment is shown in Table 3. Each entry in the table contains one of three values:

- ✓: Approaches naturally fulfill the requirement, or can fulfill the requirement with a little amount of additional development effort.
- ~: Approaches need a considerable amount of additional development effort to fulfill the requirement, or only fulfill the requirement partly.
- ✗: The requirement is difficult to fulfill with these approaches. Non-trivial modifications are needed to facilitate the requirement's fulfillment.

Table 3: Comparison of migration approaches in related works in regards to their fit to the requirements

Requirement	**Virtual Machine Live Migration**	**Checkpoint/ Restore Container Migration**	**Explicit State Management**
Basic Migration	✓	✓	✓
Recovery	✗	✗	✓
Problem Handling	✓	✓	✓
Consistency	~	~	✓
Development	✓	✓	~
Downtime	✓	✓	✓
Fog Compatibility	~	✓	✓

When comparing the approaches in Table 3, the similarities between the *VM live migration*, and the *checkpoint/restore container migration* approaches become visible. In terms of their capabilities to fulfill the requirements, the only difference is that *VM live migration* approaches are less appropriate in terms of the *Fog Compatibility*. This is apparent since containerization is the better-suited technology for virtualization on low resource computing devices [24], which is also why deploying fog computing applications in VMs has been becoming less popular in recent years [34]. Because of this reasoning, the *VM live migration* approaches are dismissed in favor of the *container checkpoint/restore* based approaches.
Excluding the *VM live migration* leaves *explicit state management* and the *checkpoint/restore container* approaches to be compared further. Both classes of approaches fundamentally address the *Basic Migration* requirement, both can adopt measures for *Problem*

Handling, as wells as measures to minimize the *Downtime*. In addition, both approaches can provide the required *Fog Compatibility*.
The main differences in their capabilities therefore lie in their fulfillment of the *Development*, *Recovery* and *Consistency* requirements. While the *checkpoint/restore container* migration approaches migrate the whole container's state, using *explicit state management*, only the state of particular PEs is migrated. The container's state is acquired by checkpointing the whole container, which does not require any additional actions by a PE developer. In *explicit state management*, developers need to expose the PE's state, most commonly abstracted as operator state. Possibilities for exposing the state include requiring developers to implement getter and setter functions for the state [7, 37], and providing source code annotations to mark the PE's state [16]. This means that even though the additionally needed development effort for PEs can be minimized, the *explicit state management* approaches fulfill the *Development* requirement to a lesser extent than the *checkpoint/restore container* approaches.
When using *explicit state management* at the application level, the state can be abstracted as operator state. This provides an explicit representation of the buffer state. A correct buffer state is essential to enable *Recovery* since the correct processing state can only be restored when its corresponding position in the event stream is known. This is problematic when looking at the *checkpoint/restore container approaches* since the container checkpoint has no clear representation for the buffer state. While it may be possible to address this issue by adding either constant monitoring of the PE's processing position in the event stream, or by explicitly informing a PE about the migration, this would be non-trivial to achieve. Additionally, the added overhead would most probably come at the cost of impairments to the processing performance. This makes *explicit state management* the more appropriate approach in regards to the *Recovery* requirement.
Similarly, a missing representation for a buffer state hinders the *checkpoint/restore container* approaches in terms of the *Consistency* requirement. The *Consistency* requirement outlines an exactly-once semantic as the optimal outcome. To achieve this, positional information, like the buffer state information, is needed to guarantee that no event is processed twice or not processed at all. If a *checkpoint/restore container* approach would be followed, this would have to be addressed in-depth to fulfill the *Consistency* requirement better.
Summarizing, the *checkpoint/restore container* approaches achieve the ability to migrate an entire container without additional development effort at the cost of less control over the migrated contents and the context of the containers state. This makes providing measures for fault-tolerance difficult, which is required by the *Recovery* requirement. Additionally to these findings, the *checkpoint/restore container* approaches introduce a technology dependency between the containerization technology and the checkpointing tool, which can be problematic. This is shown by the fact that there are compatibility

problems between some containerization technologies, like Docker, and the checkpointing tool CRIU [5, 10].
Due to the overall better fit to the requirements and to stream processing in the fog, the developed conceptual framework further explores *explicit state management* at the application level.

According to Ding et al. [13], the two most important, but often neglected, questions when designing a mechanism for state migration are how to migrate and what to migrate. In the development of the conceptual migration framework, these two questions are therefore addressed. Since the migration process is dependent on the migration's contents, the question of what to migrate is answered first.

4.2.1 Operator State

Like most other application-level approaches, the state is abstracted as operator state, due to the benefits mentioned above. As described in 2.2, so-called stateless PEs possess routing and buffer state, which means that they can be treated like stateful PEs with no processing state.
On the application level, the components of the operator state have to be addressed individually:

Routing State
The centralized orchestrator handles the routing state. Since the centralized orchestrator keeps track of the information of all deployed pipelines, it also possesses the information about the routing state. During migration, the orchestrator provides the required routing state and adjusts routing accordingly.

Processing State
The PE possesses the processing state. It can be extracted from the PE and serialized for the migration. Additionally, the PE offers an interface that allows inserting a serialized processing state.

Buffer State
Handling the buffer state consists of handling the PE's position in the event stream and handling the output buffer. The PE keeps track of which events have been processed and which have not. This is referred to as the PE's position in the event stream, which is easily accessible. The output buffer has to be handled depending on its implementation. A possible way to handle the output buffer is by using a messaging service between PEs that possesses inbuilt capabilities to replay events. Another possibility is to keep output events stored on the upstream node and replay events from there, as proposed by Fernandez et al. [7].

4.2.2 Migration Procedure

As laid out before, two migration situations are distinguishable:

1. A correctly running PE is migrated from the origin node to the target node.
2. A PE that is interrupted (e.g., origin node lost network connection; power outage at origin node, etc.) is migrated to the target node.

In the first situation, it is possible to get the current state of the PE on the origin node and to migrate this state. This is not possible in the second situation since there is no available PE whose state could be transferred. Instead, a checkpoint of a past version of the PE's state is needed to recreate the current operator state on the target node and to start processing afterward. Therefore, this situation requires a mechanism to get operator state checkpoints, which are available even when the origin node is no longer reachable, as well as a mechanism to restore from a past checkpoint.

To account for the differences in these situations, two distinct approaches are proposed. The concept for situation 1 is referred to as *PE live migration* and the concept for situation 2 as *PE restoration*. To enable the *PE restoration* Dual Level Asynchronous Checkpointing (DLAC) is proposed to provide the needed checkpointing capabilities.

Pipeline Element Live Migration

The centralized orchestrator coordinates the *PE live migration*. It handles the deployment of all pipelines and thus possesses the information corresponding to the affected PEs. The procedure is depicted in Figure 6. The displayed activity of the PEs in the chart represents the event processing activity.

After the migration procedure is triggered, the centralized orchestrator pauses processing on the origin node and fetches its current operator state. The PE on the target node is invoked with the received operator state. The target node uses the received operator state to set its processing state and buffer state. After that, processing is started. The PE on the target node then informs the centralized orchestrator about whether the invocation was successful or unsuccessful. If it was successful, the paused PE on the origin node is discarded. If it was unsuccessful, the PE on the target node is discarded, and the PE on the origin node is resumed. This distinction protects from problems while inserting the operator state on the target node and other invocation related problems, therefore addressing the *Problem Handling* requirement.

This procedure requires PEs to offer an interface over which event processing can be paused and resumed, and over which the operator state is accessible.

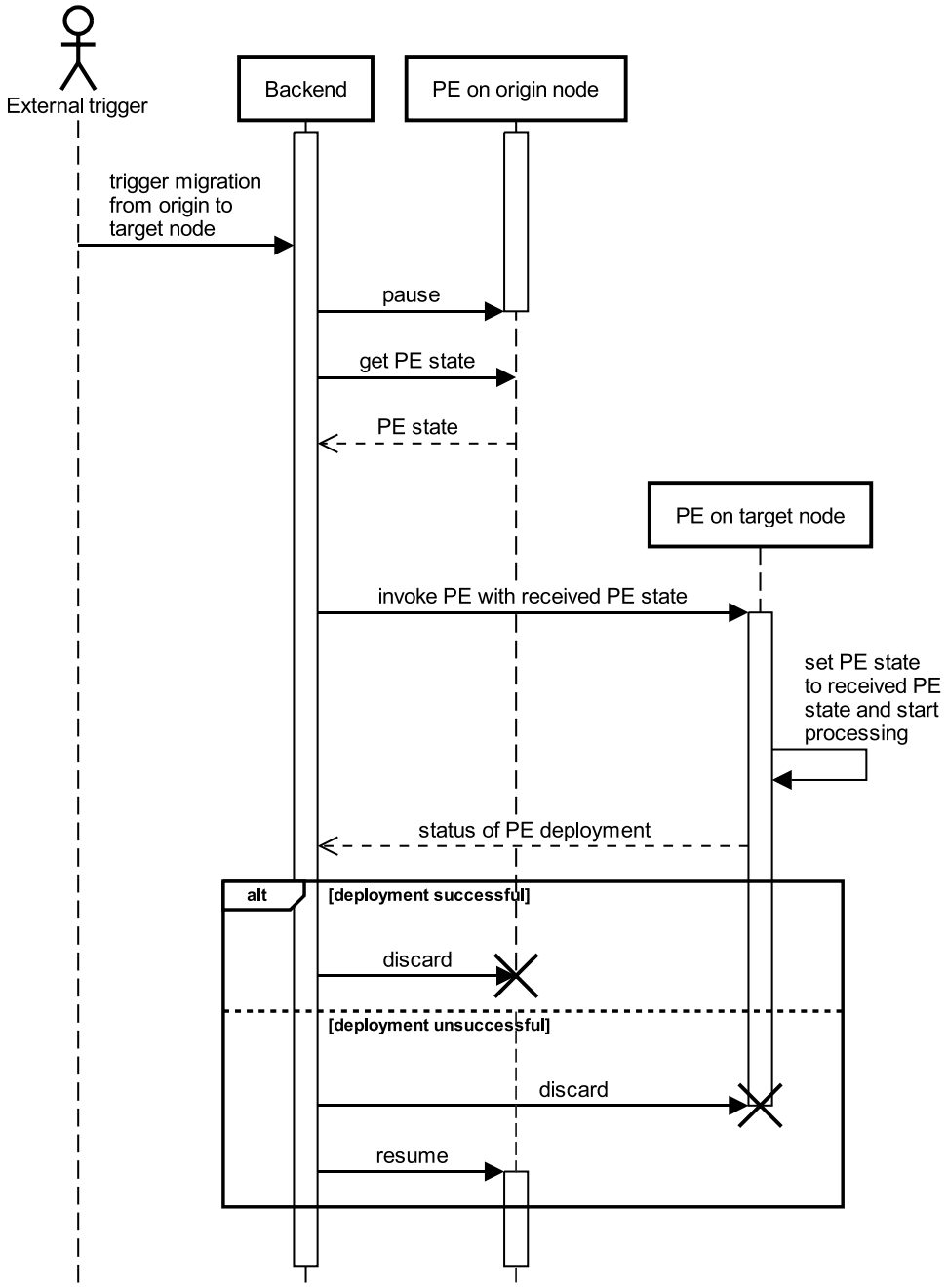


Figure 6: Procedure for the *PE live migration*

Dual Level Asynchronous Checkpointing

As mentioned before, it is necessary to integrate measures for state checkpointing to fulfill the *Recovery* requirement. These measures must allow the state recreation even when the node with the faulty PE is unreachable. To achieve this DLAC is proposed.

As a measure for fault-tolerance, a set of databases is deployed. Figure 7 depicts the setup for DLAC. The setup consists of a database at every node that hosts PEs (PE-level) and a centralized database that sits on the same centralized node as the centralized orchestrator (central-level). On the PE-level checkpoints of all running PEs are acquired

asynchronously and stored in the database. To achieve a consistent local checkpoint, the state acquisition requires a PE to pause processing events while its processing and buffer state information is gathered and returned. Additionally, checkpoints of all running PEs are retrieved at the central-level. For the central-level the checkpoints are obtained by contacting a PE-level interface that returns the most recently acquired checkpoints from the nodes database.

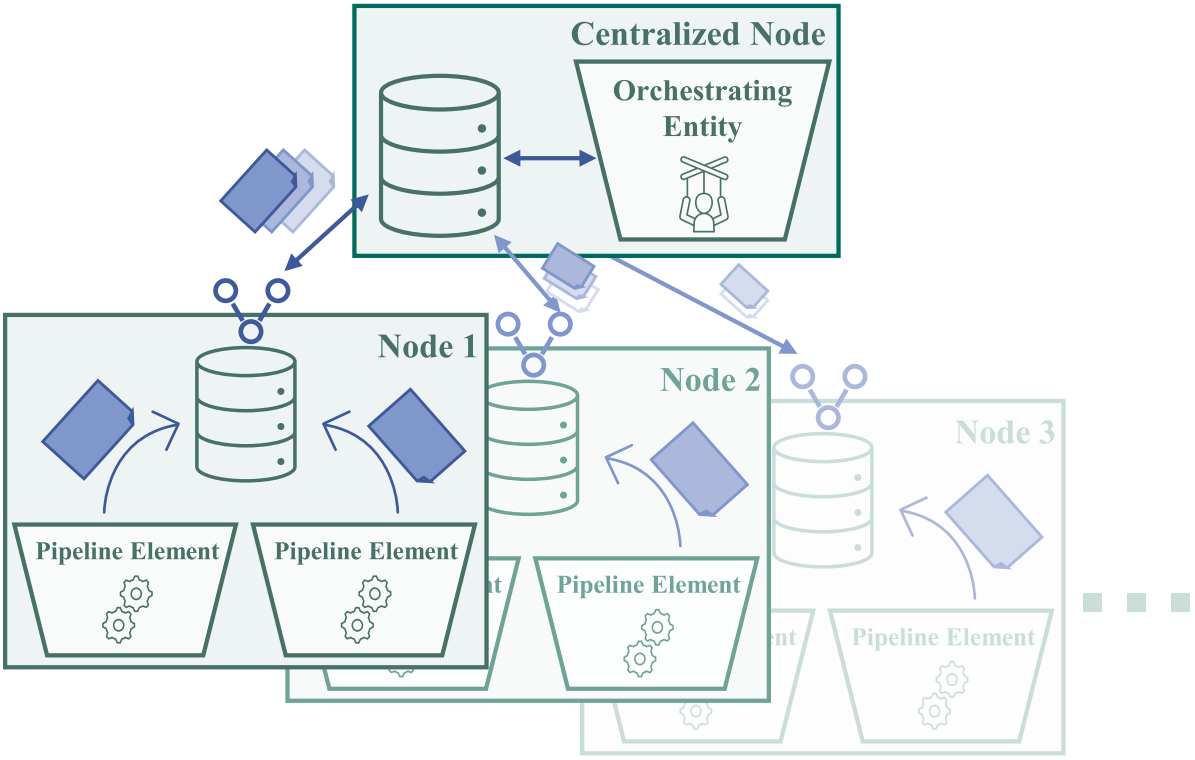


Figure 7: Setup for Dual Level Asynchronous Checkpointing

The deployed tactic for checkpointing is asynchronous independent checkpointing. This tactic is chosen since (a-)synchronous global checkpointing does not offer any benefits for the use case at hand, while imposing a higher complexity (e.g., Chandy-Lamport algorithm) or performance impairments (e.g., "stop-the-world" approaches). Instead, the PEs are checkpointed independently, which is sufficient since the state is not used to recreate the whole pipeline from a specific point in time, but rather to recreate single PEs.
While the use case of *PE restoration* solely relies on the centrally available state checkpoints, the additional co-location of operator state checkpoints with their respective PE allows for a local redeployment in the case of a fault at the PE. This local redeployment falls outside the scope of this thesis but should be addressed by future work.
Additionally, the event processing impairment is minimized by offering an interface, which provides the most recent checkpointed state, instead of offering the current operator state, whose acquisition would require event processing to be temporarily paused. This means that collecting checkpoints at the central-level does not influence event processing directly.

Pipeline Element Restoration

Figure 8 shows the situation encountered when the *PE restoration* is triggered. Three points in time are particularly important:

1. The time when the last checkpoint of the PE on the origin node was recorded before it was interrupted.
2. The point in time when the PE on the origin node was interrupted.
3. The timing at which the PE is restored on the target node.

The operator state's available checkpoint is most likely deprecated at the time the state restoration is triggered since events might have been processed after the last checkpoint was recorded and synchronized with the centralized database. To recreate the state the PE had at 2, the events that were processed between 1 and 2 have to be reprocessed. Reprocessing refers to processing events in the PE without outputting events to an event stream. This recreates the processing state at 2. After that, the events that have accumulated between 2 and 3 are processed before the regular processing of events can begin. The events between 2 and 3 are referred to as intermediate events.

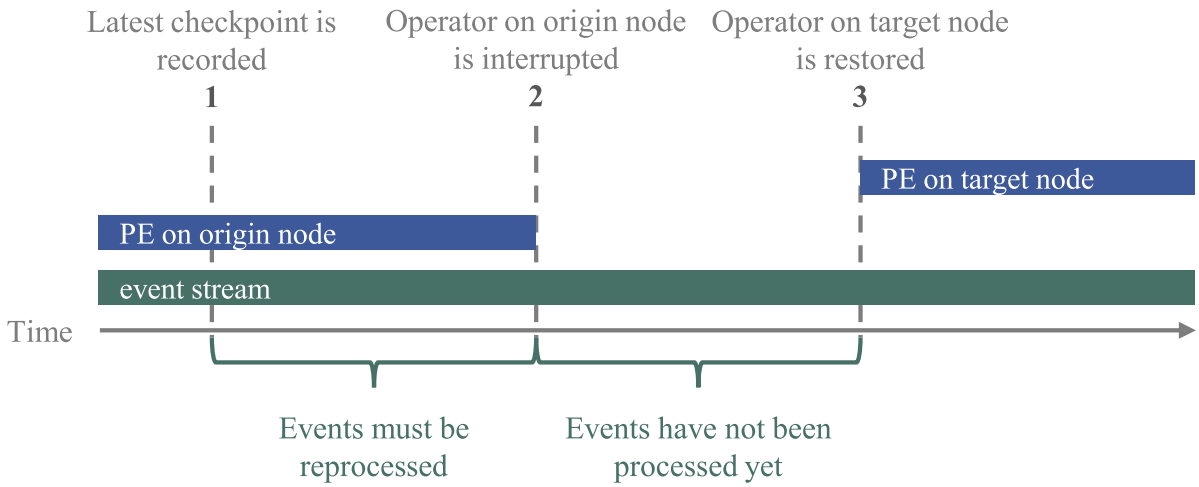


Figure 8: Context of the *PE restoration*

The procedure for the *PE restoration* is depicted in Figure 9. After the *PE restoration* is triggered for a specific interrupted PE and a target node, the centralized orchestrator fetches the latest available checkpoint of the interrupted PE's state from the centralized database. This checkpoint is then used to invoke a new PE on the target node. The newly invoked PE then reprocesses events, as described before, then processes the intermediate events until it catches up to the latest events and afterward starts processing incoming events regularly. As the last step, the newly invoked PE informs the centralized

orchestrator about its status. This information can be used to decide whether additional actions, like retrying the *PE restoration*, are needed.

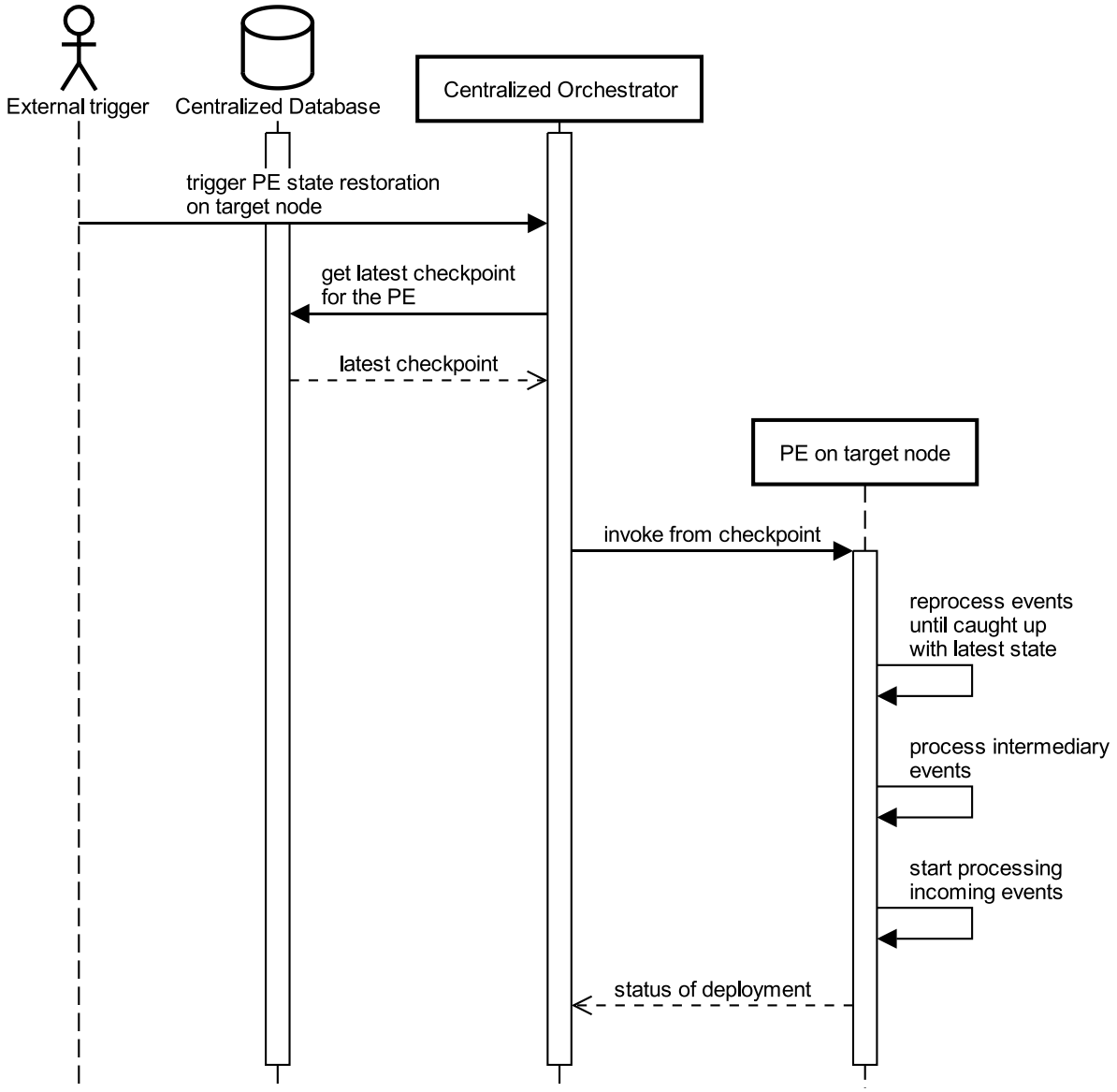


Figure 9: Procedure for the *PE restoration*

4.2.3 Assessment of the Conceptual Migration Framework

To close this chapter, the fit between the conceptual framework and the requirements is discussed.

The *Basic Migration* requirement and the *Recovery* requirement are the basis of the conceptual framework. Hence, these requirements are addressed throughout the conceptual framework.

The *PE live migration* focuses on the *Problem Handling* requirement by keeping the PE on the origin node paused until a successful migration is confirmed. Otherwise, the paused

PE could be resumed, and the migration could be retried at a later point in time or with another target node. The *PE restoration* cannot provide this safety measure since the PE on the origin node is not available in this case. A rudimentary measure is the return of the restoration status, which allows a situation-specific handling. For example, the *PE restoration* could be retried with another target node or from an older checkpoint.
The consistency of the resulting PE output (*Consistency* requirement) is addressed by including explicit information about the position in the event stream, from which processing should be picked up through the buffer state information. This information is mandatory for a consistent *PE live migration*, as well as for a consistent *PE restoration*.
The minimization of the *Downtime* is considered in the *PE live migration* as event processing is only paused for the duration of the state transferal and the invocation of the new PE.
The *Development* and *Fog Compatibility* requirements are not directly addressed by the developed conceptual framework and instead need to be taken into account when implementing it.

5 Implementation

To show the validity of the developed conceptual framework, the *PE live migration*, the *PE restoration*, and the supplementary concept element of DLAC are implemented in this chapter. This implementation serves as a proof of concept. The implementation is realized as an extension of the open-source data analytics platform Apache StreamPipes (incubating). The beginning of this section gives an introduction to StreamPipes and its intricacies. After that, the modifications to StreamPipes, necessary to enable migration, are presented. Closing out this chapter, the implementation details of the concept elements developed in chapter 4 are outlined.

5.1 StreamPipes

Apache StreamPipes (incubating) provides a self-service toolbox for the analysis of data streams. The target users are domain experts, who do not necessarily possess programming experience. For this reason, StreamPipes offers a Graphical User Interface (GUI) through which domain experts can create stream processing pipelines themselves. StreamPipes is mainly deployed in the (industrial) IoT and is extendable through an openly available software development kit that provides templates for different types of PEs.
The components that comprise StreamPipes are depicted in Figure 10. All components are deployed in an interconnected web of docker containers. Docker is a frequently used containerization software in fog computing [5]. The main component of the StreamPipes

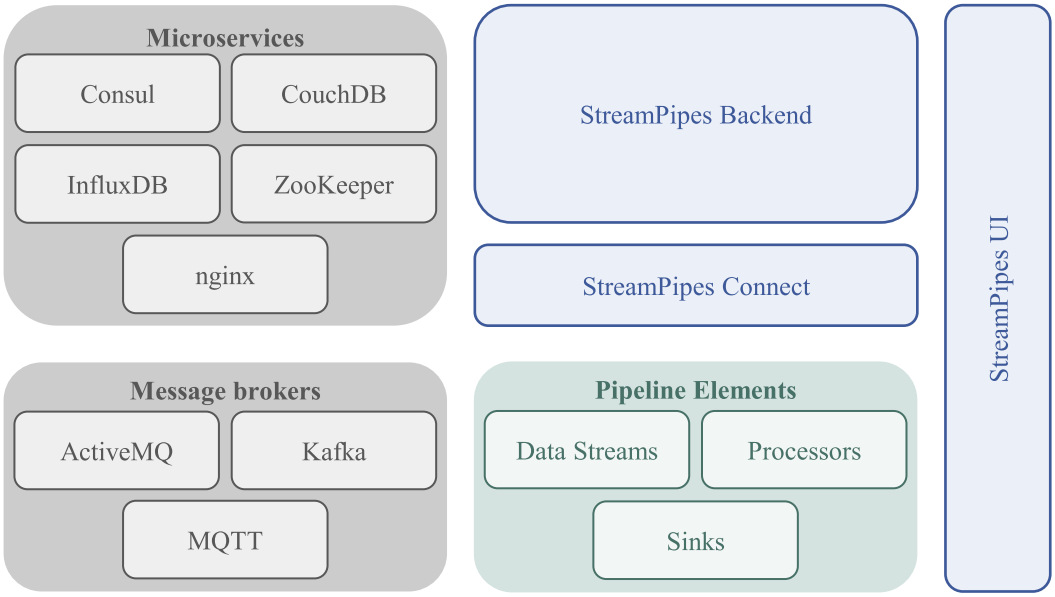


Figure 10: Components of the StreamPipes application

application is the backend, which orchestrates and oversees the lifecycle of deployed pipelines. It works in collaboration with separately set-up microservices, which offer extending functionalities, for example, store pipeline and user information. Users interact with the backend indirectly through the GUI, which is deployed as a web application. The GUI uses the backend's Representational State Transfer (REST) endpoints as an interface to its functionalities. Users create stream processing pipelines through the so-called pipeline editor, which is part of the GUI. StreamPipes realizes pipelines as an interconnected sequence of PEs.
The PEs are implemented as reusable, and independent event-driven functions, meaning that they can take events as input, perform a specific task on them, and/or output events. The PEs are further categorized into *data streams* feeding event streams into the pipeline, intermediary PEs called *processors* transforming one or more input event streams to an output event stream, and PEs named *sinks*, which solely take inputs but do not produce outputs. The StreamPipes Software Development Kit (SDK) provides wrappers to implement PEs in different programming languages (e.g., Java, Python) and for Big Data processing engines like Apache Flink. The proof of concept implementation only addresses the PEs implemented using the java wrapper since .
The interconnections between sequential PEs are handled by intermediary message brokers like Apache Kafka or Apache ActiveMQ. While it is possible to implement the conceptual framework for all types of message brokers, Apache Kafka is used in this thesis because its native message retention capability and offset management enable a convenient way to manage buffer state.

5.2 Extension of StreamPipes

To implement the conceptual framework into StreamPipes, the existing application has to be modified and extended. First, the PE framework is modified to allow operator state management. Second, the supplementary checkpointing concept DLAC is implemented.

5.2.1 Integrating Operator State

As discussed in chapter 4, the PE needs to provide and handle the buffer state and the processing state information.

Processing State

To add a new java PE to StreamPipes, developers using the java PE wrapper must implement a set of classes. The class containing the event processing logic (processor class) must either implement the *EventProcessor* or the *EventSink* interface, depending on what kind of PE is designed. These interfaces do not expose the processing state, which means that

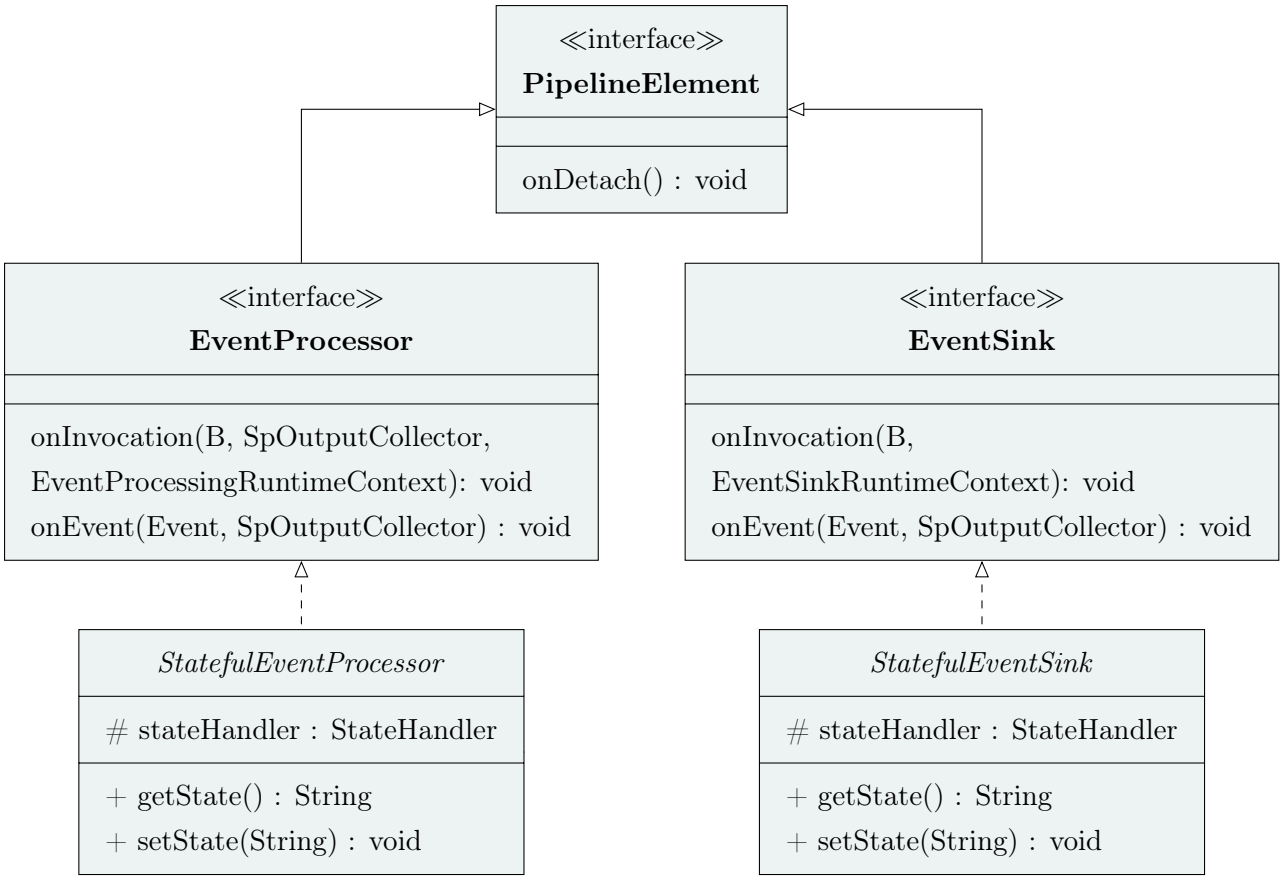


Figure 11: UML class diagram for the classes *StatefulEventProcessor* and *StatefulEventSink*

the processing state cannot be migrated without an extension. To facilitate manageable processing state in PEs, the abstract classes *StatefulEventProcessor* and *StatefulEventSink* were added, as shown in Figure 11. These newly introduced classes add the methods "getState" and "setState", which act as interfaces to the PE's processing state.
Using this extension, a stateful PE is implemented with a processor class that extends either the *StatefulEventProcessor* or the *StatfulEventSink* class. From thereon, developers can either choose to use the provided default processing state handling or opt to override the "getState" and "setState" methods. Developers might choose to implement a customized processing state handling, if it yields performance improvements or if the default state handling struggles to serialize or restore the processing state consistently.
This extension introduces the differentiation between stateful and stateless PEs and allows access to the processing state of stateful PEs. Additionally, existing PEs that implement the *EventProcessor* or *EventSink* interface can still be used without any adaptation. However, if they possess a processing state, they should be updated to extend one of the introduced abstract classes instead and to expose their processing state accordingly.

```
1  public class Counter extends StatefulEventProcessor<
   CounterParameters> {
2  @StateObject public int counter = 0;
3
4  @Override
5  public void onInvocation(CounterParameters parameters,
   SpOutputCollector spOutputCollector, EventProcessorRuntimeContext
   runtimeContext) {
6  this.stateHandler = new StateHandler(this);
7  }
8  @Override
9  public void onEvent(Event event, SpOutputCollector out) {
10     event.addField("aggregation", ++counter);
11     out.collect(event);
12 }
13 @Override
14 public void onDetach() {}
15 }
16
```

Figure 12: Implementation of a PE that counts the events it has processed

The default processing state handling is coordinated by a *StateHandler* entity. This *StateHandler* entity has to be initialized in the obligatory "onInvocation" method implemented in the PE. The *StateHandler* serializes and restores the PE's processing state. It uses a pluggable serializer (default: Gson-serializer) to serialize and deserialize the processing state. Developers, who use this default processing state handling for their PE, need to mark all member variables that comprise the processing state with an "@StateObject" annotation, as illustrated in Figure 12. The member variables that compose the processing state are acquired by the *StateHandler* through the Java Reflection API.
By offering the *StateHandler*, and the possibility to provide custom functions that expose the processing state, the benefits of both approaches can be combined. On the one hand, using the provided *StateHandler* makes exposing the processing state easily implementable, which addresses the *Development* requirement. On the other hand, implementing PE-specific processing state getter and setter functions can help to address PE-specific particularities. For example, PE-specific possibilities for performance improvements can be seized, which is beneficial regarding the *Downtime* requirement. Dealing with consistency issues is also facilitated, addressing the *Consistency* requirement.

Buffer State

The buffer state is obtained from the connection to the intermediary Apache Kafka message broker. Apache Kafka is a distributed event streaming platform, which offers a

publish-subscribe based messaging system [18]. This system works with events as messages, which are organized and stored in so-called topics [18]. Each event in a topic gets assigned an offset, which refers to the event's position in the overall order of events. Additionally, Kafka keeps tracks of which events have already been successfully processed and which have not.

The PE connects to the message broker through *EventConsumer*s, which fetch data from a message broker, and *EventProducer*s, which publish events to a message broker. The *EventConsumer* interface is expanded with a "getConsumerState" method and a "setConsumerState" method, which expose the buffer state. These methods are implemented in the *SpKafkaConsumer* class as shown in Figure 13. Additionally, the methods "pause" and "resume" are implemented. When "pause" is executed, event processing is suspended until the "resume" function is triggered. This temporary suspension of event processing

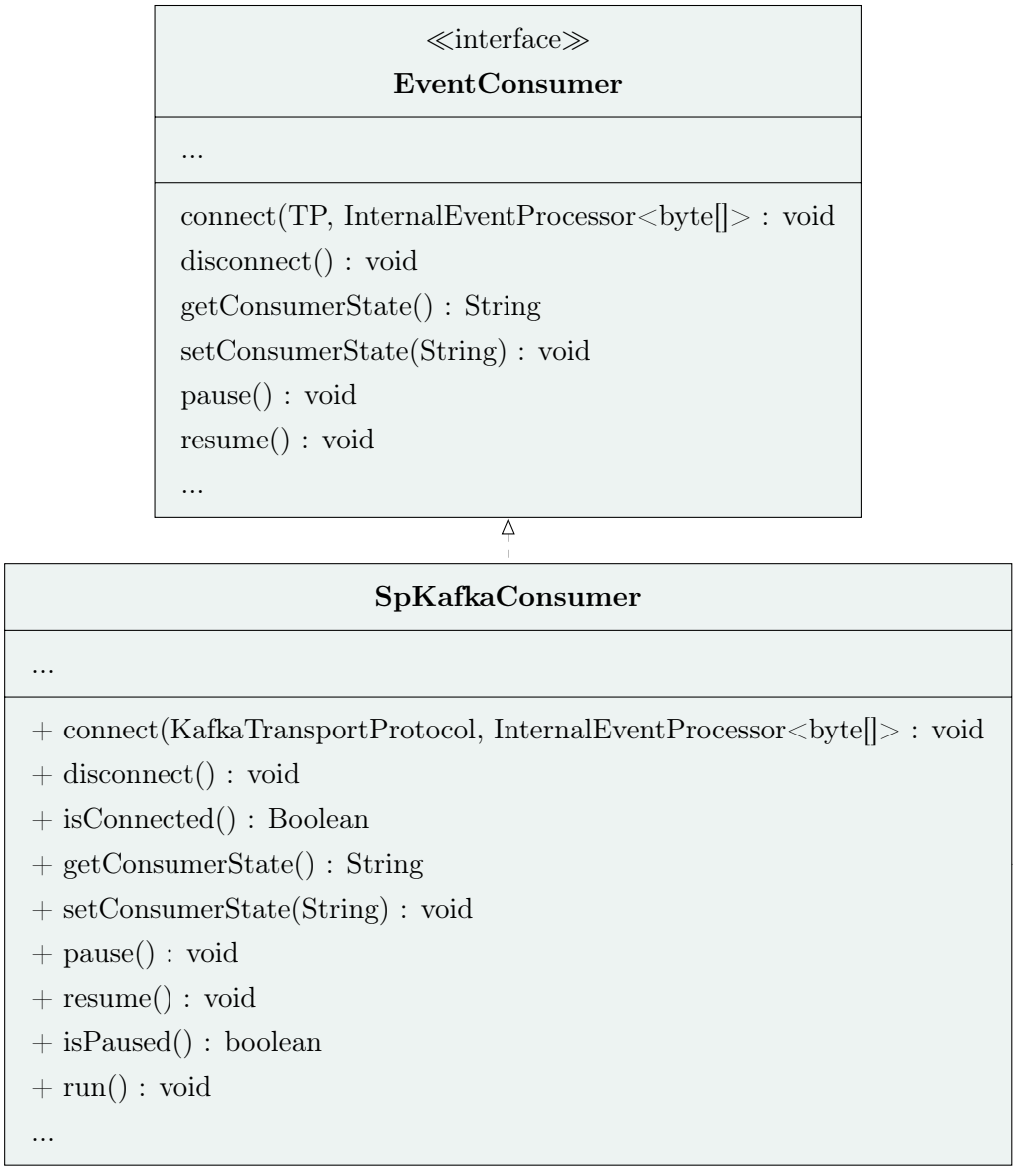


Figure 13: Abbreviated UML class diagram for the *SpKafkaConsumer*

is not only needed in the *PE live migration* but also every time the operator state is serialized. Pausing the event consumption during PE state serialization enables a consistent serialization of the entire operator state.

For the representation of the buffer state, Apache Kafka's offset management is used. Thus, the serialized buffer state contains the consumer's current offsets and its group Id, which represent the exact position in the data stream when the serialized state is requested. When the buffer state is set, this information is deserialized and is used in the consumer's connection procedure, during which it connects to the Kafka message broker. This connection procedure was modified to enable processing pick-up and state-restoration. Algorithm 1 presents the newly introduced state restoration procedure as simplified pseudo-code. It replaces the previous connection process, which did not allow reconnecting to a specific offset. In line 4, the algorithm checks whether the provided offset, from which processing should be picked up, is unequal to the latest offset stored by Kafka. If this is untrue, processing can start as usual, because the provided offset already corresponds with the latest offset. If it is true, events were processed after the provided snapshot was taken, signifying that events must be reprocessed (as depicted in the last chapter in Figure 8). As already mentioned, reprocessing refers to replaying past events to recreate the correct processing state, from which regular processing can continue. Reprocessing is performed in line 11. In contrast to regular processing, reprocessing does not produce any output. When all events until the latest offset have been reprocessed, the loop is interrupted (line 9 and 6), and regular processing starts from the latestOffset onwards.

Algorithm 1 Consumer restoration algorithm

```
1:  procedure RestoreConsumer(offset)
2:      KafkaConsumer.subscribe(topic)
3:      latestOffset ← KafkaConsumer.currentOffset
4:      if offset ≠ latestOffset then
5:          KafkaConsumer.setOffsetTo(offset)
6:          while KafkaConsumer.currentOffset < latestOffset do
7:              events ← KafkaConsumer.pollEvents()
8:              for each event in events do
9:                  if event.offset ≥ latestOffset then
10:                     break
11:                 engine.reprocess(event)
12:         KafkaConsumer.setOffsetTo(latestOffset)
```

5.2.2 Dual Level Asynchronous Checkpointing

The foundation for DLAC is the state-database, which stores checkpoints of PE states. This database is implemented using the java wrapper for RocksDB named RocksJava. RocksDB is a persistent key-value store that offers high performance, especially on flash storage [46]. The setup of the state-database is presented in Figure 14. PEs are registered in the state-database through the *DatabasesSingleton*. Each registered PE has a corresponding instance of a *PipelineElementDatabase*, which acts as an interface to a column family in the *StateDatabase*. Column families are logical partitions in RocksDB, which can be thought of like columns for key-value pairs. Each key-value pair is affiliated with exactly one column family. The *StateDatabase* assigns a unique column family to each registered *PipelineElementDatabase*. Using this implementation, each PE has its own corresponding column family inside one database. The *StateDatabase* uses RocksJava to persist checkpoints to storage. It uses a singleton design pattern to facilitate the setup of a single database with multiple column families. When a new *PipelineElementDatabase* is instantiated, it registers itself in the *StateDatabase* and gets assigned its unique *ColumnFamilyHandle*, which is needed to save data to and read data from a column family. In short, the *StateDatabase* sets up PE-specific column families, which are interfaced to through *PipelineElementDatabases*. All *PipelineElementDatabases* are tracked by the *DatabasesSingleton*.

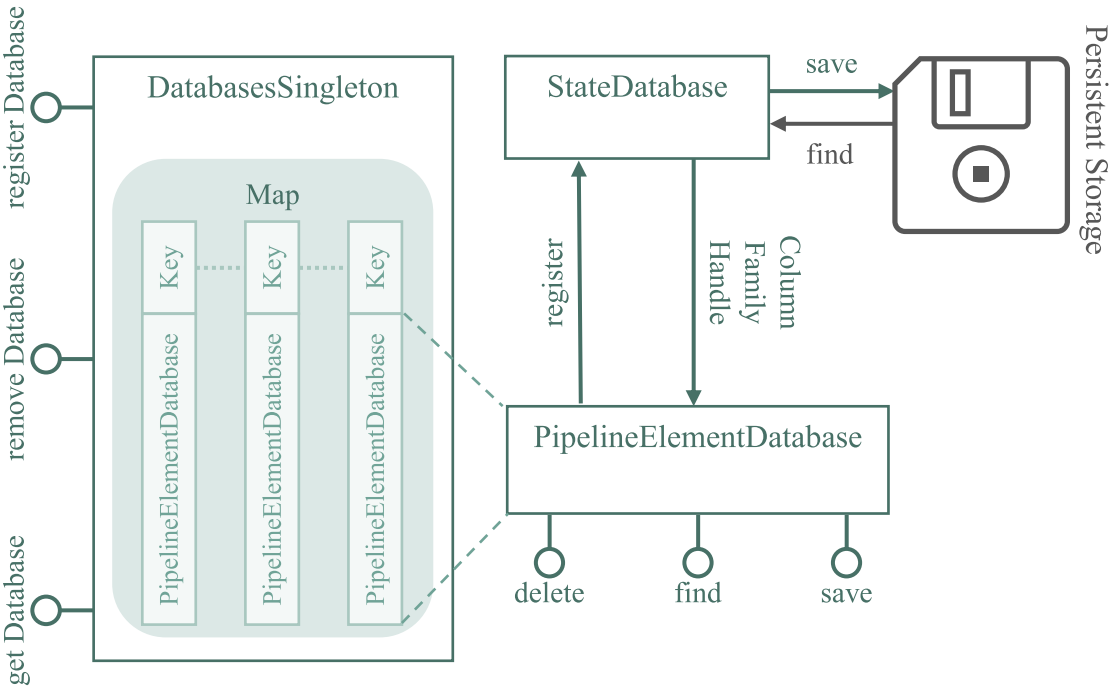


Figure 14: Setup of the StateDatabase used for DLAC

This setup has the advantage of only requiring a single RocksDB database, regardless of the number of PEs whose checkpoints are stored. The checkpoints from different PEs are stored in PE-specific column families. Each *PipelineElementDatabase* instance provides an encapsulation of one column family. By assigning exactly one *PipelineElementDatabase* to one PE, the assignment of column families to PEs becomes definite.

The database setup is used as storage for PE state checkpoints. When a pipeline is started, two *PipelineElementDatabases* are created for each PE in the pipeline. The backend creates the first one at the central-level. The second one is created in the invoked PE's container at the PE-level. In addition to creating these instances, each *PipelineElementDatabase* and its respective PE are registered for checkpointing in the container's specific *CheckpointingWorker* (depicted in Figure 15). The *CheckpointingWorker* oversees and executes the asynchronous checkpointing. All registered PEs are periodically checkpointed until they are unregistered from the *CheckpointingWorker*. The *CheckpointingWorker* has

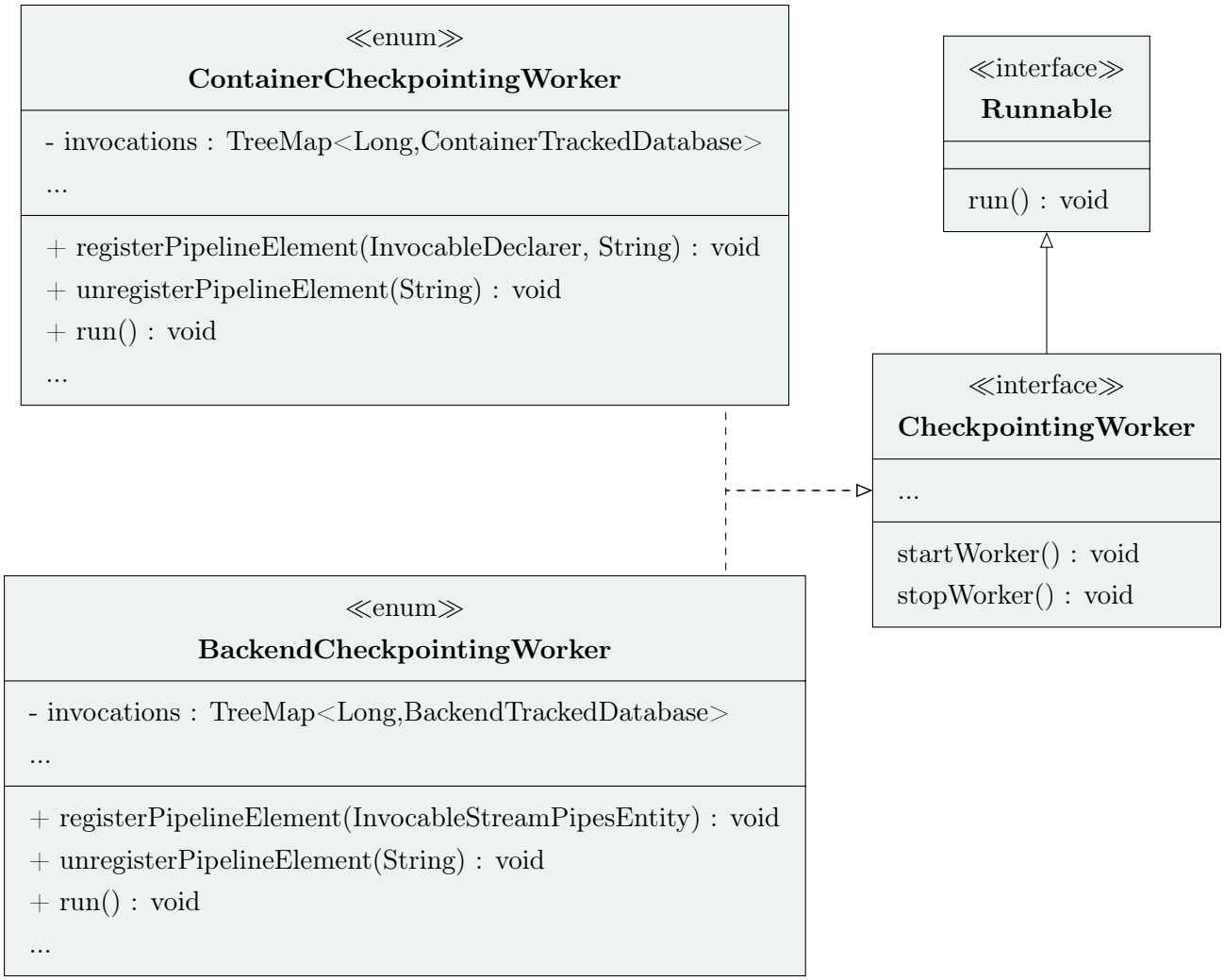


Figure 15: Abbreviated UML class diagram for the *Container/BackendCheckpointingWorker*

distinct implementations for the backend (central-level) and for the containers containing the PEs (PE-level). The PE-level *CheckpointingWorker* uses the PE's "getState" method to get a checkpoint, which is then saved in the PE's *PipelineElementDatabase*. The central-level *CheckpointingWorker* calls the PE's "getCheckpoint" method through its REST endpoint. This method returns the most recent checkpoint from the *PipelineElementDatabase* at PE-level.
DLAC is realized as the composition of a set of *ContainerCheckpointingWorkers* at PE-level and one *BackendCheckpointingWorker* at central-level. Both of the *CheckpointingWorker* implementations are optimized to idle for as much time as possible between checkpointing registered PEs, to minimize the impact on computing resources.

5.3 Migration Procedure

The aforementioned modifications to the PEs need to be made accessible on the PE side to implement the conceptual migration framework. Additionally, the procedures for the *PE live migration* and the *PE restoration* need to be implemented in the backend.

5.3.1 Pipeline Element-Side Implementation

To make the modifications and extensions introduced in 5.2 accessible, the PE's REST interface is extended. The newly implemented and modified methods, which are accessible through individual REST endpoints, are listed in Table 4.

Table 4: Overview of methods added on the PE-side

Method	Description
pauseElement(String, String)	Pauses the consumption of events from the message broker until the PE is resumed or detached. Event processing is paused through the "pause" method added to the *SpKafkaConsumer* class.
resumeElement(String, String)	Resumes the PE if it has been paused before. Event processing is resumed through the "resume" method added to the *SpKafkaConsumer* class.
getState(String, String)	Returns the current operator state. When executed, this method pauses event consumption, gets a representation of the operator state, using the aforementioned functionalities to serialize the buffer and the processing state, and resumes event processing immediately after that.
getCheckpoint(String, String)	Returns the most recent operator state saved in the PEs state-database. This method relies on the PE-side state database, where the PE's checkpoints are regularly saved to by the *ContainerCheckpointingWorker*.
invokeRuntime(String, String)	Modified to recreate the PE state from a PE state checkpoint before event processing begins. The processing and buffer states are restored using the functions and procedures described in section 5.2.

5.3.2 Backend-Side Implementation

On the backend side, the *PE live migration* and the *PE restoration* are implemented.

Pipeline Element Live Migration

Figure 16 provides a shortened and simplified representation of the *PE live migration* implementation, realizing the concept outlined in 4.2.2. The depicted "migrate" method is implemented into the *PipelineExecutor* class and is accessible through a REST endpoint in the backend.

```
public PipelineOperationStatus migrate(PipelineElement pe, String
    targetNodeUrl){
     Pipeline pipeline = PipelineStorage.get(pe.getPipelineID());
     String originNodeUrl = pe.getUrl();
     BackendCheckpointingWorker.unregisterPipelineElement(pe.ID)
     //Pause Pipeline Element that should be migrated
     PipelineOperationStatus status = HttpRequest.pause(pe);
     if(!status.isSuccess()){
       status.setTitle("Could not pause the Pipeline Elements.");
       BackendCheckpointingWorker.registerPipelineElement(pe)
       return status;
     }
     //Get state of the Pipeline Elements
     PipelineElementState state = HttpRequest.getState(pe);
     //Start Pipeline Element with received states
     pe.setUrl(targetNodeUrl);
     PipelineOperationStatus statusInvoc = HttpRequest.invoke(pe, state);
     if(statusInvoc.isSuccess()){
       //Discard Pipeline Element on origin node
       operator.setUrl(originNodeUrl);
       PipelineOperationStatus detach = HttpRequest.detach(pe);
       if(!detach.isSuccess()){
         statusInvoc.setTitle("Failed to detach old Elements.");
         statusInvoc.setSuccess(false);
       }
       pipeline.getPipelineElement(pe).setUrl(targetNodeUrl);
       pe.setUrl(targetNodeUrl);
       BackendCheckpointingWorker.registerPipelineElement(pe);
       return statusInvoc;
     }else{
       //Resume processing on origin node
       operator.setUrl(originNodeUrl);
       PipelineOperationStatus statusRes = HttpRequest.resume(pe);
       if(statusRes.isSuccess()){
         statusRes.setSuccess(false);
         statusRes.setTitle("Migration unsuccessful, resumed on origin");
       }else{
         statusRes.setTitle("Migration unsuccessful, could not resume.");
       }
       BackendCheckpointingWorker.registerPipelineElement(pe);
       return statusRes;
     }
}

```

Figure 16: Abbreviated and simplified implementation of the *PE live migration*.

In addition to what is outlined in the concept conceptual framework, implementation related factors are addressed. First, the correct registration to the *CheckpointingWorker* is handled. Second, the stored pipeline related metadata is adjusted. This metadata contains, amongst other information, a topological description of the pipeline. Since this information changes during a successful migration, the saved pipeline representation is modified as well. Furthermore, the implemented method returns a *PipelineOperationStatus* instance, which provides information about the migration outcome. This information can be used elsewhere to handle the outcome and to react to it.
Looking at the implementation in detail, the code can be divided into five blocks. From lines 2-4, the procedure is prepared with its necessary preceding actions. Next, in lines 6-11, the PE on the origin node is paused. If this is unsuccessful, the procedure is aborted. In the third block, the current PE state is fetched from the origin node (line 13). After that, lines 15 and 16 invoke the PE with the received PE state on the target node. Lastly, lines 17-41 handle the outcome of the invocation. If it was successful, the PE on the origin node is discarded (lines 19-28). If it was unsuccessful, the PE on the origin node is resumed, and a failure notice is returned (lines 31-40).
This implementation aims at achieving an exactly-once semantic throughout the migration. By leveraging the Kafka offset management for the buffer state representation, the target node PE can connect to exactly the position in the event stream, at which processing was suspended on the origin node. Pausing the PE on the origin node in a controlled manner guarantees that the consumer offset corresponds with the latest processed event.

Pipeline Element Restoration

Figure 17 shows a shortened and simplified illustration of the *PE restoration* implementation. The *PE restoration* procedure is implemented in the *PipelineExecutor* class and accessible through its own REST endpoint.

Similarly to the implementation of the *PE live migration*, the procedure handles the PE checkpointing registration, and the modification of the saved pipeline description.
The implemented procedure can be partitioned into four code blocks. In lines 2-4, necessary preparations for the procedure are taken. Next, the most recent state checkpoint is fetched from the centralized state database in line 6. In lines 7 and 8, the saved pipeline representation is modified, and the PE's Uniform Resource Locator (URL) is adjusted to match the target node's URL. Finally, the PE is invoked from the checkpoint on the target node (lines 10-12). After the invocation, the invocation's status is returned.

```
public PipelineOperationStatus restore(PipelineElement pe, String
    targetNodeUrl){
     Pipeline pipeline = PipelineStorage.get(pe.getPipelineID());
     String originNodeUrl = pe.getUrl();
     BackendCheckpointingWorker.unregisterPipelineElement(pe.ID)
     //Get state from database
     PipelineElementState state = DatabasesSingleton.INSTANCE.getDatabase
    (pe.ID).getLast();
     pe.setUrl(targetNodeUrl);
     pipeline.getPipelineElement(pe).setUrl(targetNodeUrl);
     //Start Pipeline Element with state
     PipelineOperationStatus  statusInvoc = HttpRequest.invoke(pe, state)
    ;
     BackendCheckpointingWorker.registerPipelineElement(pe);
     return status;
}

```

Figure 17: Abbreviated and simplified implementation of the *PE restoration*.

The implementation of the *PE restoration* cannot guarantee an exactly-once semantic throughout the migration. Instead, it provides an at-least-once semantic. An at-least-once semantic guarantees that no event stays unprocessed [21]. However, it does not guarantee that an event is not processed twice [21]. In the implementation at hand, an event could be processed a second time because the consumer offset tracked by Kafka is only adjusted when an event is committed. Committing an event refers to sending an acknowledgment that an event has been processed to the Kafka broker. If a PE is interrupted after successfully processing an event but before committing this event, the *PE restoration* does not achieve an exactly-once semantic throughout the migration. In such a case, all uncommitted events are fully processed again.

6 Evaluation

In this chapter, the implementation of the migration procedure and the checkpointing routine are evaluated. The evaluation is split into different experiments, each of which consists of several test series. To assess the implementation's suitability in terms of the requirements stated in chapter 4, the experiments focus on different aspects of a common test-case. The first experiment (E1) analyzes the downtime and overall migration time in the *PE live migration* under different networking conditions and state-sizes. The second experiment (E2) addresses the *PE restoration*, focusing on the reprocessing performance. Lastly, checkpointing is examined by analyzing the impact of different state sizes on the checkpointing duration (E3). Thereafter, the evaluation results and the proof of concept implementation are critically discussed in the light of the requirements. Finally, the consequential limitations of this thesis are briefly outlined.

6.1 Setup

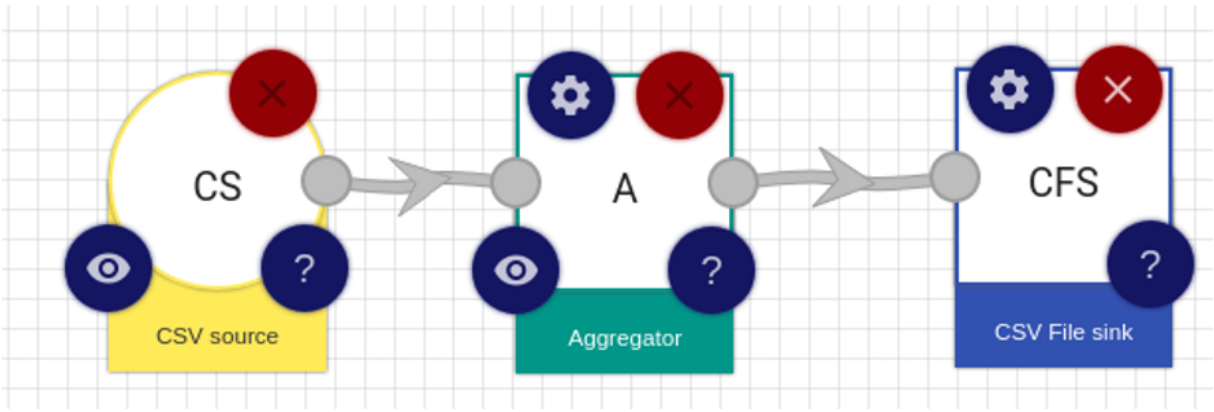


Figure 18: Screenshot of the test pipeline used throughout the evaluation

The test case used for the evaluation is the aforementioned Case 1. The needed PEs were implemented in StreamPipes, and the pipeline was recreated, as shown in Figure 18. Using Case 1 as an exemplary case allows for a reliable way of determining whether the resulting output is consistent. The correct outcome is known since this pipeline is a composition of deterministic PEs with a fixed input from a predefined CSV file. Any deviations from the expected output, which is also saved to a CSV file, flag inconsistent results.

This test-case is evaluated on a setup of two devices, which are both connected to a local router via gigabit ethernet. The first device is a desktop PC with a 6-core, 3.60GHz AMD Ryzen 3600 processor with a total of 16GiB of RAM running Ubuntu 20.04. It simulates a centralized computing entity. To simulate a small computing device in the fog, a Raspberry Pi 4 single-board computer is deployed as the second device. It is powered by

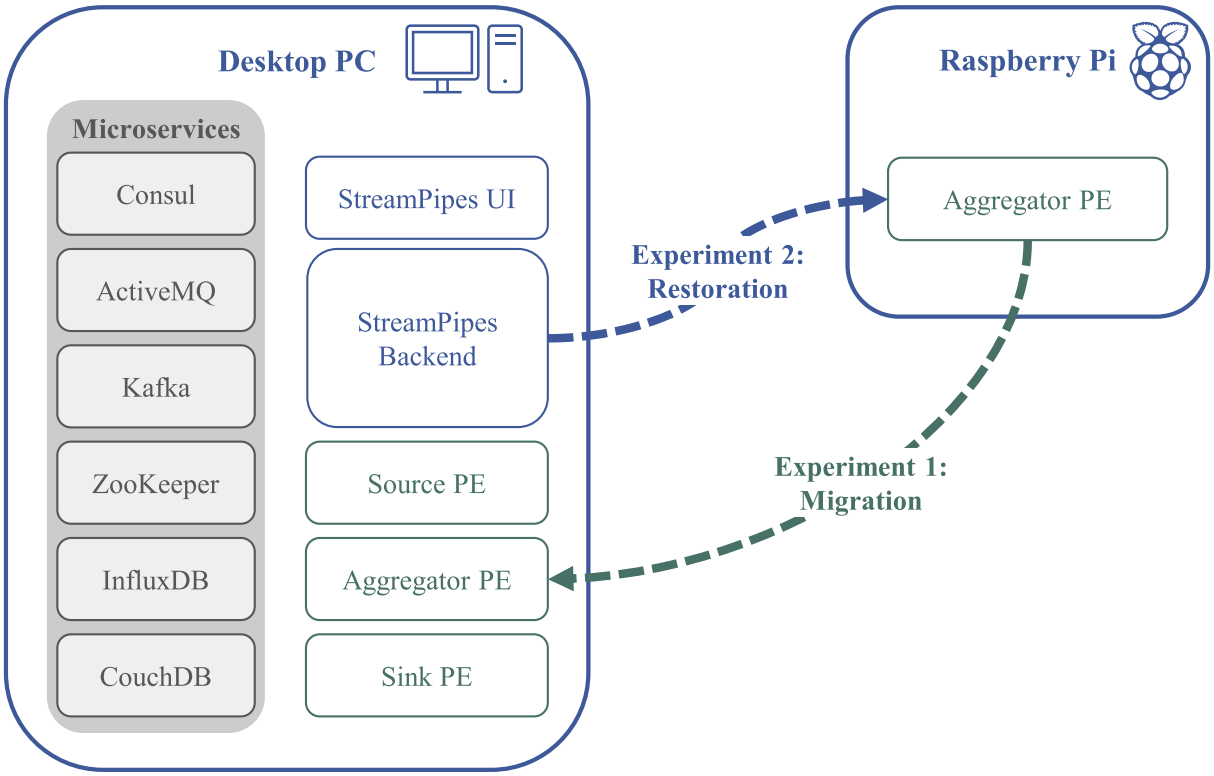


Figure 19: Setup for the evaluated experiments

a 4 core 1.50GHz ARM Cortex-A72 processor, has 4GiB of RAM, and runs a 64bit version of Raspberry Pi OS (a Debian based operating system optimized for the Raspberry Pi). To evaluate the test-case, the application is deployed, as depicted in Figure 19. The desktop PC hosts the micro-services mandatory for the StreamPipes deployment, namely Consul, Apache ActiveMQ, Apache Kafka, Apache ZooKeeper, InfluxDB, and Apache CouchDB, as well as the StreamPipes Backend and the user interface. These microservices are all deployed in independent but connected docker containers. Additionally to this base setup, the desktop PC hosts the source and sink PE, as well as an instance of the aggregating PE. The Raspberry Pi also hosts a containerized instance of the aggregating PE.

6.2 Results

Experiments E1, E2, and E3 were conducted using the above described test case on the test setup. The results of these experiments are presented in this sub-chapter.

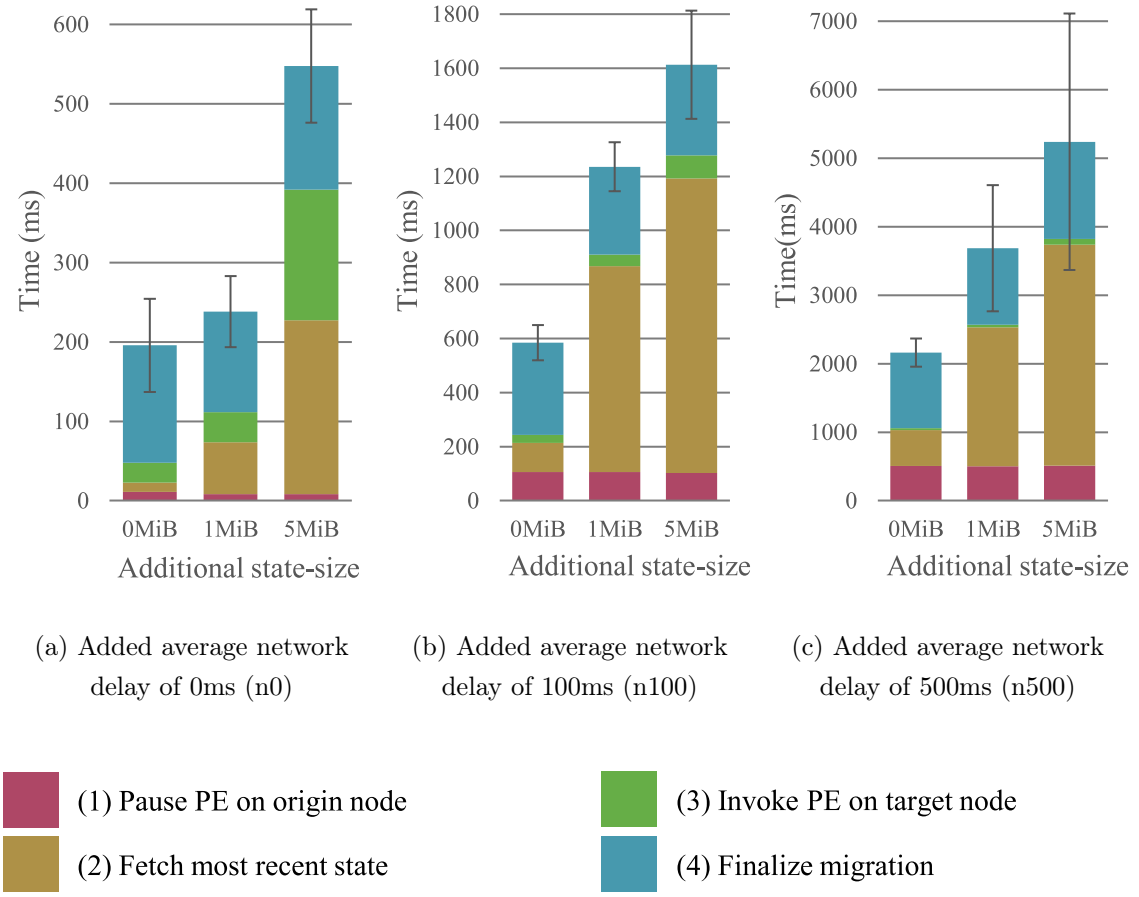
6.2.1 Pipeline Element Live Migration

In the first experiment, the test case pipeline was set up using a source and sink PE on the desktop PC and an aggregator PE on the Raspberry Pi. Following a random wait time after starting the pipeline, the aggregator on the Raspberry Pi was migrated to the

desktop PC using the *PE live migration*.
To simulate a real-world use case, this experiment was conducted while varying the network conditions and the state-size. The network conditions were varied to mimic the varying latencies encountered in fog computing [34]. This variation was realized by using the linux traffic control tools to add a network delay to traffic between the Raspberry Pi and the desktop PC. Three scenarios were distinguished. Tests with no added network delay (n0) (averaging a latency of 0.2ms between the devices), tests with an added network delay that is normally distributed averaging 100ms with a variance of 10ms (n100), and the last tests with a normally distributed network delay averaging 500ms with a variance of 50ms (n500), were conducted. State-sizes are the second varied factor. The variation of the state-size simulates different PEs, whose states can have vastly different sizes. State-sizes range from sub-Kibibyte sizes, for example, in the aggregator used in the experiments, when it is configured to aggregate over a narrow window of past events, to state-sizes in the tens of Mebibyte (1024^2 Byte) (MiB). For instance, a PE that uses a machine learning model that changes over time due to reinforcement learning could have a larger state-size. The same aggregating PE is used throughout all experiments and was therefore equipped with the capability to increase its state-size artificially. The state-size is increased by adding a String value with configurable length to the state. The experiment was conducted with three state-size setups. The first test-series was conducted with no added state (s0), the second one with 1MiB of added state (s1), and the last test-series with 5MiB of added state (s5).

In total, nine test-series were conducted, with differing network conditions and state-sizes between test-series. In all test-runs, the aggregating PE was configured to aggregate the last ten values. The events were fed into the pipeline with an average frequency of one event every 300ms. Each test was performed ten times to achieve more reliable results.

Figure 20 depicts the total migration times for the different test-series. The total time is further broken down into 4 phases, corresponding to the migration procedure. In the first phase, the PE is paused on the origin node. The current state is fetched from the PE on the origin node in the second phase. In the third phase, the PE is invoked on the target node with the previously fetched state. The migration is finalized by discarding the PE on the origin node and saving the changed pipeline description. Since the PE on the target node begins processing after its invocation, the sum of the first three phases provides an upper limit for the PE downtime. The error bars in Figure 20 represent the standard deviation for the total time of migration.
In the test-series without added network delay (Figure 20a), the PE downtime increased from 48ms with s0, to 112ms with s1, and further to 308ms with s5. In the test-series with an average of 100ms added network delay (Figure 20b), the PE downtime rose from an average of 243ms with s0, to 911ms with s1, and 1278ms with s5. In the final test-



(a) Added average network delay of 0ms (n0)

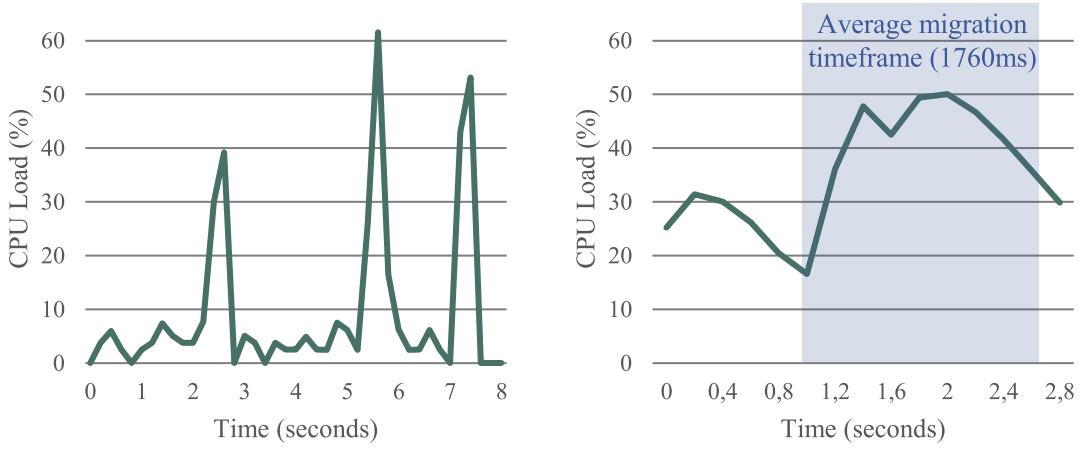
(b) Added average network delay of 100ms (n100)

(c) Added average network delay of 500ms (n500)

Figure 20: Results of E1 under different networking conditions

series, with an average of 500ms added network delay (Figure 20c), the results increased similarly. They went from 1057ms with s0, to 2571ms at s1, and 3823ms at s5.
Overall, the results show two correlations. Firstly, the total time of migration and the PE downtime both increase with an increasing state-size. Secondly, the total time of migration and the PE downtime also increase with an increasing network delay. The phases whose duration increased the most with increasing state-size were phases 2 and 3. Across all network delays, the average duration of phase 2 rose by 540% between tests with s0 and s1 and by 200% between tests with s1 and s5. The average duration of phase 3 across all network delays increased by 150% between tests with s0 and s1 and by 220% between tests with s1 and s5. In contrast to these two phases, which involve handling the operator state, the duration of phases 1 and 4 did not change significantly in regards to the state-size. However, the duration of these two phases, together with phase 2 did rise with increasing network delay. The average duration of Phase 1 across all state-sizes rose by 1160% between tests with n0 and n100 and by 490% between tests with n100 and n500. A similar increase can be observed for phase 2, with an increase in duration by

880% between tests with n0 and n100 and by 350% between tests with n100 and n500. This relation can also be seen for phase 4, whose duration increases by 230% between tests with n0 and n100 and by 360% between tests with n100 and n500.



(a) CPU load in an exemplary test case (b) Average CPU load during *PE live migration*

Figure 21: CPU load in the PE's container on the Raspberry Pi

In addition to these measurements, the CPU on the Raspberry Pi was monitored. Figure 21a shows an exemplary test run with s5 and n0. The CPU load is on a level of under 10% for most of the time with three spikes of 40% to 60% at 2.4 seconds, 5.4 seconds, and 7.2 seconds. The first two spikes correspond with the timing of the checkpointing. The last spike coincides with the migration.
To further inspect the CPU load during the migration, a follow-up test-series was conducted, for which the artificial state-size was increased to 50MiB. Figure 21b shows the average CPU load for the migration timeframe. During this timeframe, an increase of the CPU load to 50% can be seen.

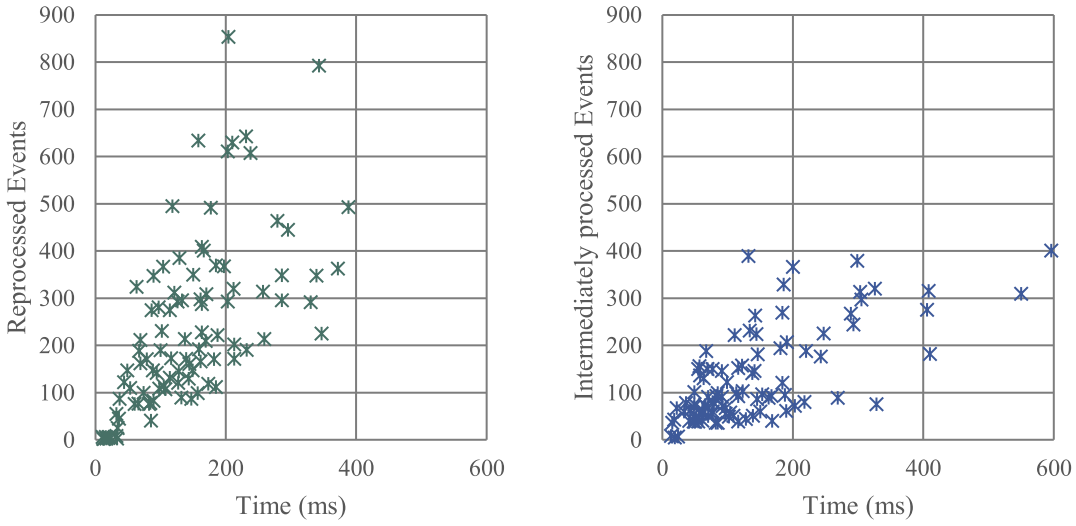
The last evaluated factor was the consistency of the results. The consistency was assessed by analyzing the output of the sink PE. As previously explained, the expected output is deterministic. Therefore, the consistency check was conducted by comparing the sink output with the expected output. In a total of 110 evaluated runs, the result was always consistent.

6.2.2 Pipeline Element Restoration

For the second experiment, the test case pipeline was set up using a source, sink, and aggregator PE on the desktop PE. After a random period of time, the aggregator on the desktop PC was discarded to simulate a PE becoming interrupted. Thereafter, a random waiting period followed, after which the aggregator was restored on the Raspberry Pi

using the *PE restoration*. Since checkpointing is done at regular intervals, the random timing of discarding the PE leads to a variation in the number of events that need to be reprocessed. As the time period between discarding and restoring the PE is also random, the number of intermediary events is also irregular. Intermediary events are the events occurring in the time between the interruption of the PE and the start of the restoration (as described in section 4.2 Figure 8).

For this experiment, a total of 100 test runs were conducted. In all test-runs, the aggregating PE was configured to aggregate the last ten values. The state-size was artificially increased by 1MiB. The events were fed into the pipeline with an average frequency of one event every 300ms. Additionally, the PE-level checkpointing was configured to a frequency of checkpointing once every five seconds.



(a) Duration of reprocessing in dependence of the number of reprocessed events

(b) Duration of processing intermediate events in dependence of the number of intermediately processed events

Figure 22: Duration of reprocessing and intermediately processing events after *PE restoration*

Figure 22 displays the results of E2. In Figure 22a, the time spent reprocessing is depicted in dependence on the number of reprocessed events. The results show an increase in processing time with an increase in the number of reprocessed events. A similar relation can be observed between the time spent processing intermediary events and the number of intermediary processed events (Figure 22b). The average time per reprocessed event is 0.608ms, which is significantly lower than the average time per intermediary processed event, which takes 1.109ms on average. This difference can be attributed to the fact that the result of processing intermediary events is output by a Kafka producer, whereas the output of reprocessing is discarded.

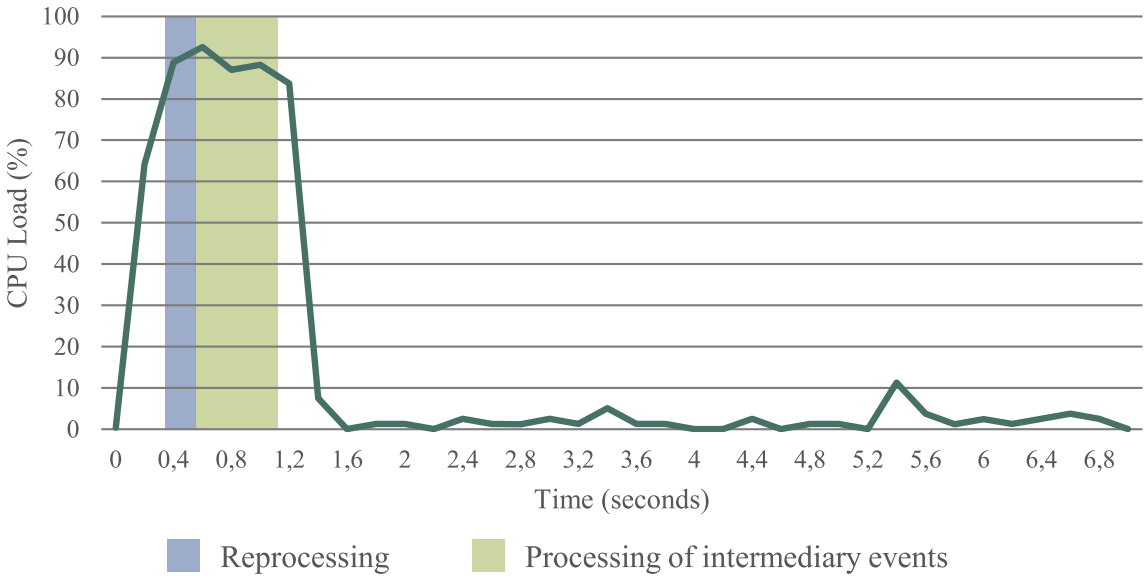


Figure 23: CPU load in the PE's container on the Raspberry Pi for an exemplary test case

The *PE restoration* has a significant impact on the CPU load. This is shown in Figure 23, which depicts the CPU load of the aggregating PE's container on the Raspberry Pi in a selected test-run. In the time between 0.338 seconds and 0.570 seconds, 191 events were reprocessed. After that, 309 intermediary events were processed until 1.120 seconds. Over this period, the CPU load is increased to around 90%. Once regular event processing takes over, the CPU load drops to under 10% where it levels off.

To evaluate the consistency of the *PE restoration*, five additional test-runs were conducted. For these test runs the set up was adjusted. Instead of initially deploying the test pipeline solely on the desktop PC, the aggregator was deployed on the Raspberry Pi. In the test-runs, the Raspberry Pi was forcefully interrupted by cutting of its power connection a random time period after the pipeline was started. Thereafter, a *PE restoration* with the desktop PC as target node was performed.
The resulting output of the five conducted test runs show no inconsistencies.

6.2.3 Checkpointing

The checkpointing performance was assessed based on the data gathered in E1. Figure 24 depicts the average time checkpointing the aggregator's operator state takes on the Raspberry Pi, depending on the additional state size. While the duration of checkpointing was only 3.1ms in the tests with no added state, the duration increased to 23.8ms at 1MiB of artificially added state and rose to 84.1ms with 5MiB of added state. The share of the serialization time in the total checkpointing duration also increased from 0.3% with no

added state to 32.3% with 1MiB added state and further to 40.6% with 5MiB added state.

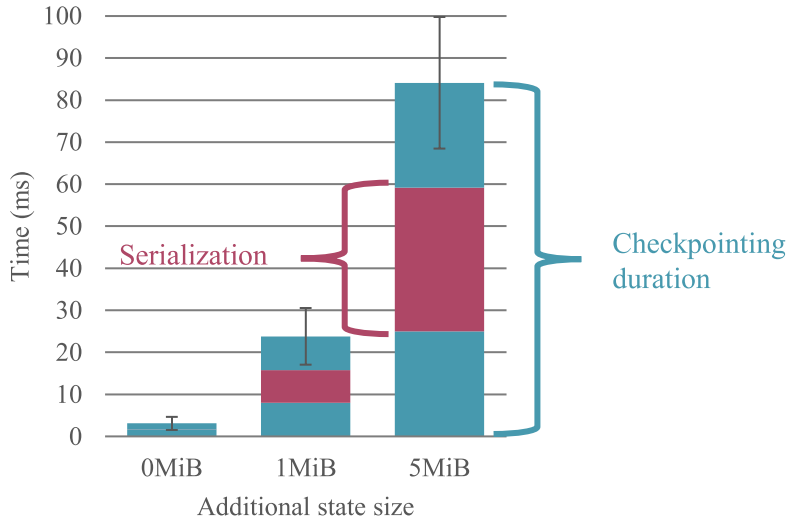


Figure 24: Checkpointing duration in dependence of operator state size

6.3 Discussion

In this discussion, the implementation and the evaluation results are examined against the backdrop of the requirements. From thereon, the limitations of this thesis are derived.

Basic Functionality

The basic functionality required by the *Basic Migration* and the *Recovery* requirement is provided by the conceptual framework developed in 4.2. The proof of concept implementation could satisfy these requirements throughout the experiments.
Therefore, the *Basic Migration* and the *Recovery* requirement are met.

Problem Handling

Problem Handling is another requirement concerning the migration functionality. While the *PE live migration* implementation addresses problem handling, the *PE restoration* does it only to a lesser extent. In the *PE live migration*, the PE on the origin node is paused until the correct invocation on the target node is confirmed. Processing is restarted on the origin node if the PE cannot be invoked on the target node, providing a safety measure for problems during the migration. The *PE restoration* implementation only informs about the restoration attempt's success or failure but does not provide any active measure to address problems.
Overall, the *Problem Handling* requirement is addressed partly by the conceptual framework. To fully meet this requirement, software components that orchestrate the migration (e.g., decide when and with which target to migrate) need to react to the feedback received

from the migration procedures.

Consistency

Achieving an exactly-once semantic with a consistent output is the best-case scenario, according to the *Consistency* requirement. The *PE live migration* is implemented with the intention of an exactly-once semantic. The results of E1 showed no inconsistencies, providing evidence that the exactly-once semantic was achieved and that the operator state is migrated as desired.
The *PE restoration* implementation is designed to provide an at-least-once semantic, with the intention to avoid processing events a second time when possible. The preliminary results from E2 showed no inconsistencies, meaning that the correct operator state was reconstructed and that no event was not processed or processed twice (except for reprocessing, which does not output events).
Therefore, the consistency requirement was largely met. A possible further improvement of the *PE restoration* could be achieved with Apache Kafka as an intermediary message broker, using its in-build capabilities for exactly-once processing [29]. On the flip side, this would add additional overhead to event processing.

Pipeline Element Downtime

The *Downtime* requirement states that the PE downtime should be as low as possible, which is addressed by the developed conceptual framework and the implementation. The evaluation results reveal factors which influence the PE downtime in both the *PE live migration* and the *PE restoration*.
For the *PE live migration*, the results of E1 imply a positive correlation between state-size and migration duration (p=xx) and a positive correlation between network delay and migration duration. This increase in the migration duration also entails an increase in the PE downtime.
In E2, the *PE restoration* was examined. A positive correlation between the number of reprocessed events and the reprocessing duration was observed ($p<0.0001$). A positive correlation was also found in the relation between the number of intermediate events processed and the duration of the intermediate processing of events ($p<0.0001$).
The information gathered in E1 reveals a potential for downtime improvement. A high PE downtime can be attributed to network impairments and great state-sizes. In a situation where the origin node of the PE that should be migrated suffers from a high network delay, it could be beneficial to restore the PE on the target node from an old checkpoint using the *PE restoration* procedure (instead of *PE live migration*). The *PE restoration* is favorable if getting the checkpoint from the database and reprocessing events can be done faster than getting the PE's most recent state. However, the decision between the procedures, additionally depends on the PE in question, since it can be suspected that different PEs vary in their performance when reprocessing events. Thus, a holistic approach is needed

to address this problem.
In conclusion, there is further potential to lower the *Downtime*, even though the PE *Downtime* is considered in the proof of concept implementation. Using a holistic approach and applying the tactics proposed in related works could help to minimize the downtime.

Development

According to the *Development* requirement, the migration capability should add as little effort as possible to the development of PEs. This is addressed in the proof of concept implementation by providing the universal *StateHandler*, which offers a convenient way for developers to expose the processing state. Additionally, the buffer and routing state are handled automatically.
Overall, the *Development* requirement is predominantly met, only requiring a minor additional effort in the PE development.

Fog Compatibility

The *Fog Compatibility* requirement demands resource constraints and varying networking conditions in the fog to be addressed.
The influence of varying network conditions on the *PE live migration* and all associated steps was examined in E1. The *PE live migration* was not susceptible to faults or inconsistencies with network delays between 0ms and 500ms. However, the network delay had a significant impact on the duration of the migration. By allowing workload mobility, the PE migration gives the possibility to lower the latency by relocating PEs to available nodes with lower latency.
The impact on CPU load was investigated for *PE live migration* in E1 and for *PE restoration* in E2. The results in E1 showed an increased CPU load of around 50% during migration. In E2, the CPU load was even higher during event reprocessing and intermediate processing, culminating at more than 90%. These observations were made with a PE deployed on a Raspberry Pi, which simulates a fog node with limited computing capabilities. Even though the CPU load increased during migration and checkpointing, these increases resembled short spikes that were limited to the duration of the respective action. During the tests, the Raspberry Pi's limited capabilities did not provoke faults in the migration procedure and did also not impact the consistency of the result. Workload mobility enables PE relocation based on resource constraints, which could allow a more balanced and effective resource usage across the devices deployed in the fog.
These preliminary results suggest that the developed proof of concept is deployable across devices with different computing capabilities and under varying networking conditions, meeting the *Fog Compatibility* requirement.

All in all, the set requirements are predominantly met by the proof of concept implementation. Out of all requirements, the *Problem Handling* and the *Downtime* requirement

show the most potential for further improvement.

Limitations

The limitations of this thesis can be categorized into the limitations of the developed conceptual framework, the limitations of the implementation, and the limitations of the evaluation.

A clear limitation of the developed conceptual framework is that it revolves around two distinct procedures, which address different cases. This is a limitation since it requires a preceding categorization of the current case to decide which procedure to choose.
Additionally, the conceptual framework is developed for a use case with a centralized orchestrator. The centralization in DLAC limits the possibility of decentralized fog computing because it creates a dependency on a centralized orchestrator and a centralized database to provide globally available checkpoints.

The implementation is limited by technology dependency, missing handling of interrupted PEs, and compatibility constraints.
A dependency exists to Apache Kafka because the implementation relies on its integrated offset management to handle the buffer state. This technology dependency could be addressed by persisting the buffer state, or even the whole operator state to the upstream PE, as proposed by Fernandez et al. [7].
Additionally, the implementation lacks measures to handle interrupted PEs. However, these measures are needed to guarantee that a supposedly interrupted PE does not come back to live, meaning that it does not start processing events again. A PE coming back to life is undesirable because it would most certainly introduce inconsistencies in the resulting output event stream. Therefore, it needs to be discarded in such a case. One scenario where this could happen is when a PE is restored on a new node because of temporary networking problems on its origin node. If the PE on the origin node reconnects to the network, it might start processing events again.
Furthermore, the proof of concept implementation cannot be deployed on devices with a 32-bit arm processor. This is due to RocksDB, which only supports arm processors with a 64-bit architecture that supports the "AArch64" instruction set. This can be problematic since small computing devices used as fog nodes might have an insufficient arm processor.

Due to the high number of possibly influencing variables on the implementation's performance, only a selection of them could be examined in the evaluation.
Neither E1, nor E2 address other components of varying networking conditions than latency. For example packet loss, could also be problematic in the context of fog computing.
Another apparent weakness of the evaluation is that only experiments in a single test scenario and with a single PE were conducted. This neglects the differences between PEs, ranging from the complexity of the processing state to the effort needed to process events.

These factors can be expected to influence the invocation time from an old state, as well as the reprocessing duration.
Lastly, the evaluation was conducted in a highly controlled environment. To determine the real-world performance, more realistic test scenarios need to be explored, for example, Case 2.

7 Conclusion

7.1 Summary

This work addressed the consistent migration of stream processing PEs between nodes in the fog. This was motivated by outlining the need for PE migration in fog computing applications, to address a multitude of characteristic constraints and requirements in fog computing. The migration allows for workload mobility, which can be utilized to minimize latency, to react to resource constraints or to address privacy issues. This becomes increasingly important with the proliferation of connected IoT devices, which can significantly profit from fog computing.

Laying the groundwork, the background section focused on the intricacies of fog computing and event-based stream processing. After introducing the concept of stateful and stateless PEs, the operator state was introduced as a proper representation of state for the use case of stream-processing. The operator state is composed of the routing, processing, and buffer state. This abstraction is used to describe the state of stream processing PEs.

Other works addressing the migration of stateful applications in the fog were subsequently presented. While early research focused on live migration of VMs, more recently, the focus shifted to the migration of containerized applications, which proves to be less resource-intensive and better suited for the migration of applications in the fog. Some authors take a different approach by utilizing explicit state-management at the application-level, instead of migrating the whole operating system with the applications. These approaches resemble the ones introduced in the literature concerning state-management in data processing, which was thoroughly examined next. In these works, state is used not only for migrating, but also to provide scalability and fault tolerance.

Based on the related works, a conceptual framework addressing the research questions was developed. It adopts application-level state-management and addresses the requirements emerging from the research questions. To address the migration of running and of interrupted PEs, both cases are treated separately. The migration of a running PE uses the current operator state, which is obtained from the PE on the origin node, to start the PE on the target node. This means that processing can be picked up on the target node at the same point it was interrupted on, on the origin node. In contrast, the migration of an interrupted PE (a PE that is unreachable) is performed using a previously acquired checkpoint of the PE's state. This checkpoint is used to restart the PE on the target node and to recreate the correct operator state. Checkpoints are obtained through DLAC, which addresses PE availability, fault tolerance, and global checkpoint availability by combining periodic asynchronous local checkpointing with periodic asynchronous global checkpointing.

As proof of concept, the developed conceptual framework was implemented into Apache StreamPipes in chapter 5. At first, the fundamental modifications and extensions to StreamPipes, enabling the migration, were implemented. Next, the implementation of the *PE live migration* and the *PE restoration* procedure was presented. The implementation addresses the requirements set in the previous chapter and shows a possible realization of the conceptual framework, addressing different challenges that are not covered by the conceptual framework.

This implemented proof of concept was evaluated in three experiments. The first experiment evaluated the *PE live migration* and showed that increased network delay and bigger state sizes impact the PE downtime and the overall migration duration. However, a negative effect on the consistency of the results could not be observed. In the next experiment, the *PE restoration* was examined. The results revealed a correlation between the number of events that have to be reprocessed and the duration of reprocessing. A similar correlation was observed between the number of intermediary events to be processed and the duration of processing them. The third experiment examined the checkpointing performance for different state sizes. The results show that the share of the processing state serialization in the checkpointing duration increased with the state-size.
After that, the acquired results were discussed, and the limitations of this thesis were outlined. In the discussion, it was established that the implementation does predominantly satisfy the requirements. Shortcomings were laid out in the limitations section.
In this chapter, this thesis is contrasted against related works. After that, the limitations are followed up with suggestions for future research that could expand on this thesis.

This thesis expands on related work by proposing a conceptual framework for the migration of stream-processing PEs in the fog.
The application-level approach proposed in this thesis represents a contrast to the virtualization level migration approaches taken in the majority of works concerning the migration of applications in the fog. The choice of virtualization level approaches can be attributed to their generality. As elaborated on in chapter 4.2, the virtualization level approaches facilitate this generality. However, using an application-level approach in the use case of stream processing allows addressing intricacies of stream processing PEs. This facilitates the fulfillment of the requirements.
While the works concerning state in data processing deploy application-level approaches for state handling, they do not address fog computing specific requirements. Additionally, only a small fraction of these works address the migration of the operator state, while most others focus on providing fault tolerance and scalability. This thesis combines measures for fault tolerance with a concept for the migration, while addressing fog specific requirements at the same time.

7.2 Future Work

During the writing of this thesis, several topics for future work were identified. Most of these topics have already been broached in the Discussion section (6.3).
Future research could explore the transfer of the developed conceptual framework to a more decentralized setting, without a centralized node that handles the global availability of state backups. This could be addressed by distributing the state checkpoints to other nodes hosting PEs, as proposed by Fernandez et al. [7].
A logical next step to improve the conceptual framework and the implementation is to unify the *PE live migration* and the *PE restoration*. This would also allow for an improvement in performance, as elaborated on in chapter 6.3.
Before the implementation of the *PE restoration* is used, the handling of interrupted PEs should be addressed. It is essential to guarantee that a supposedly interrupted PE does not come back to live, meaning that it does not start processing events again.
The implementation could be tested and examined in a test scenario closer to a real-world application and under variation of factors not yet investigated in the evaluation to achieve a more comprehensive evaluation.
Moreover, future work in general should assess the migration of applications in fog computing further, developing a more general approach that allows the migration of stream processing PEs and further applications as well.

A Appendix

A.1 Search interest in IoT

Year	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019
Average search interest	9,17	13,92	11,58	9,92	11,45	25,42	52,42	78,08	90,00	91,25	90,42

Table 5: Worldwide search interest for the topic of "Internet of Things" according to data from Google Trends. Yearly average for the years 2009-2019.

References

[1] Apache Flink. Apache flink 1.11 documentation: State backends. https://web.archive.org/web/20201001130306/https://ci.apache.org/projects/flink/flink-docs-stable/ops/state/state_backends.html, 06/08/2020. [Website; accessed 01-October-2020].

[2] B. Safaei, A. M. H. Monazzah, M. B. Bafroei, and A. Ejlali. Reliability side-effects in internet of things application layer protocols. In *2017 2nd International Conference on System Reliability and Safety (ICSRS)* (2017), pp. 207–212.

[3] Baccarelli, E., Scarpiniti, M., and Momenzadeh, A. Fog-supported delay-constrained energy-saving live migration of vms over multipath tcp/ip 5g connections. *IEEE Access 6* (2018), 42327–42354.

[4] Bonomi, F., Milito, R., Zhu, J., and Addepalli, S. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing* (New York, NY, 2012), M. Gerla, Ed., ACM Digital Library, ACM, p. 13.

[5] Brogi, A., Mencagli, G., Neri, D., Soldani, J., and Torquati, M. Container-based support for autonomic data stream processing through the fog. In *Euro-Par 2017: Parallel Processing Workshops*, D. B. Heras, L. Bougé, G. Mencagli, E. Jeannot, R. Sakellariou, R. M. Badia, J. G. Barbosa, L. Ricci, S. L. Scott, S. Lankes, and J. Weidendorfer, Eds., vol. 10659 of *Lecture Notes in Computer Science*. Springer International Publishing, Cham, 2018, pp. 17–28.

[6] Castro Fernandez, R. Stateful data-parallel processing.

[7] Castro Fernandez, R., Migliavacca, M., Kalyvianaki, E., and Pietzuch, P. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD '13 : International Conference on Management of Data : June 22-27, 2013, New York, New York, USA* ([Place of publication not identified], 2013), K. Ross, Ed., ACM, p. 725.

[8] Chandy, K. M., and Lamport, L. Distributed snapshots. *ACM Transactions on Computer Systems (TOCS) 3*, 1 (1985), 63–75.

[9] Chowdhury, M. E. H., Alzoubi, K., Khandakar, A., Khallifa, R., Abouhasera, R., Koubaa, S., Ahmed, R., and Hasan, M. A. Wearable real-time heart attack detection and warning system to reduce road accidents. *Sensors (Basel, Switzerland) 19*, 12 (2019).

[10] CRIU. Criu - status of integration with docker. `https://web.archive.org/web/20201001130722/https://criu.org/Docker`, 27/03/2020. [Website; accessed 01-October-2020].

[11] CRIU. Criu - lazy migration/post-copy memory migration. `https://web.archive.org/web/20201001130018/https://criu.org/Lazy_migration`, 27/11/2018. [Website; accessed 01-October-2020].

[12] CRIU. Criu - homepage. `https://web.archive.org/web/20201001130753/https://criu.org/Main_Page`, 29/04/2020. [Website; accessed 01-October-2020].

[13] Ding, J., Fu, T. Z. J., Ma, R. T. B., Winslett, M., Yang, Y., Zhang, Z., and Chao, H. Optimal operator state migration for elastic data stream processing.

[14] Dupont, C., Giaffreda, R., and Capra, L. Edge computing in iot context: Horizontal and vertical linux container migration. In *GIoTS 2017* (Piscataway, NJ, 2017), IEEE, pp. 1–4.

[15] Facebook. rocksdb wiki - rocksdb basics. `https://web.archive.org/web/20201001131058/https://github.com/facebook/rocksdb/wiki/RocksDB-Basics`, 06/08/2020. [Website; accessed 01-October-2020].

[16] Fernandez, Migliavacca, M., Kalyvianaki, E., and Pietzuch, P. Making state explicit for imperative big data processing: June 19 - 20, 2014, philadelphia, pa. In *Proceedings*. USENIX Association, 2014.

[17] Forsman, M., Glad, A., Lundberg, L., and Ilie, D. Algorithms for automated live migration of virtual machines. *Journal of Systems and Software 101* (2015), 110–126.

[18] Foundation, A. S. Apache kafka - introduction. `https://web.archive.org/web/20201008174852/https://kafka.apache.org/intro`, 2020. [Website; accessed 08-October-2020].

[19] Goncalves, D., Velasquez, K., Curado, M., Bittencourt, L., and Madeira, E. Proactive virtual machine migration in fog environments. In *ISCC 2018* (Piscataway, NJ, 2018), IEEE, pp. 00742–00745.

[20] Hesse, G., and Lorenz, M. Conceptual survey on data stream processing systems. In *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)* (2015), pp. 797–802.

[21] Huang, Y., and Garcia-Molina, H. Exactly-once semantics in a replicated messaging system. In *17th International conference on data engineering* (2001), IEEE Comput. Soc, pp. 3–12.

[22] Hutchison, D., Kanade, T., and Kittler, J. *From Active Data Management to Event-Based Systems and More: Papers in Honor of Alejandro Buchmann on the Occasion of His 60th Birthday*, vol. 6462 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[23] hypothesis, and Microsoft. Iot signals. Tech. rep., hypothesis group, 2020.

[24] Jalali, F., Hinton, K., Ayre, R., Alpcan, T., and Tucker, R. S. Fog computing may help to save energy in cloud computing. *IEEE Journal on Selected Areas in Communications 34*, 5 (2016), 1728–1739.

[25] Kudinova, M., and Emelyanov, P. Building mathematical model for restoring processes tree during container live migration. In *2017 Fourth International Conference on Engineering and Telecommunication* (Piscataway, NJ, 2017), E. Pavlyukova, Ed., IEEE, pp. 160–164.

[26] Ma, L., Yi, S., Carter, N., and Li, Q. Efficient live migration of edge services leveraging container layered storage. *IEEE Transactions on Mobile Computing 18*, 9 (2019), 2020–2033.

[27] Michelson, B. Event-driven architecture overview, 2006. Patricia Seybold Group.

[28] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. Naiad: a timely dataflow system. In *SOSP '13* (New York, 2013), M. Kaminsky and M. Dahlin, Eds., ACM, pp. 439–455.

[29] Narkhede, N. Exactly-once semantics are possible: Here's how kafka does it. `https://web.archive.org/web/20201011180642/https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/`, 2017. [Website; accessed 11-October-2020].

[30] Oh, S., and Kim, J. Stateful container migration employing checkpoint-based restoration for orchestrated container clusters. In *"ICT convergence powered by smart intelligence"* (Piscataway, NJ, 2018), IEEE, pp. 25–30.

[31] Osanaiye, O., Chen, S., Yan, Z., Lu, R., Choo, K.-K. R., and Dlodlo, M. From cloud to fog computing: A review and a conceptual live vm migration framework. *IEEE Access 5* (2017), 8284–8300.

[32] Ouyang, X., Rajachandrasekar, R., Besseron, X., and Panda, D. K. High performance pipelined process migration with rdma. In *Proceedings, 11th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing* (Piscataway, NJ, 2011), IEEE, pp. 314–323.

[33] Power, R., and Li, J. Piccolo: Building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2010, pp. 293–306.

[34] Puliafito, C., Mingozzi, E., Vallati, C., Longo, F., and Merlino, G. Companion fog computing: Supporting things mobility through container migration at the edge. In *2018 IEEE International Conference on Smart Computing* (Piscataway, NJ, 2018), IEEE, pp. 97–105.

[35] Puliafito, C., Vallati, C., Mingozzi, E., Merlino, G., Longo, F., and Puliafito, A. Container migration in the fog: A performance evaluation. *Sensors (Basel, Switzerland) 19*, 7 (2019).

[36] Saurez, E., Gupta, H., Ramachandran, U., and Mayer, R. Fog computing for improving user application interaction and context awareness. In *2017 IEEE/ACM Second International Conference on Internet-of-Things Design and Implementation* ([Piscataway, NJ], 2017), I. I. C. o. I.-o.-T. D. a. Implementation, Ed., IEEE, pp. 281–282.

[37] Saurez, E., Hong, K., Lillethun, D., Ramachandran, U., and Ottenwälder, B. Incremental deployment and migration of geo-distributed situation awareness applications in the fog. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems - DEBS '16* (New York, New York, USA, 2016), A. Gal, M. Weidlich, V. Kalogeraki, and N. Venkasubramanian, Eds., ACM Press, pp. 258–269.

[38] Stephens, R. A survey of stream processing. *Acta Informatica 34*, 7 (1997), 491–541.

[39] Stonebraker, M., Çetintemel, U., and Zdonik, S. The 8 requirements of real-time stream processing. *SIGMOD Rec. 34*, 4 (Dec. 2005), 42–47.

[40] To, Q.-C., Soto, J., and Markl, V. A survey of state management in big data processing systems.

[41] Venkatesh, R. S., Smejkal, T., Milojicic, D. S., and Gavrilovska, A. Fast in-memory criu for docker containers. In *Proceedings of the International Symposium on Memory Systems - MEMSYS '19* (New York, New York, USA, 2019), Unknown, Ed., ACM Press, pp. 53–65.

[42] Yao, H., Bai, C., Zeng, D., Liang, Q., and Fan, Y. Migrate or not? exploring virtual machine migration in roadside cloudlet-based vehicular cloud. *Concurrency and Computation: Practice and Experience 27*, 18 (2015), 5780–5792.

[43] Yigitoglu, E., Mohamed, M., Liu, L., and Ludwig, H. Foggy: A framework for continuous automated iot application deployment in fog computing. In *2017 IEEE 6th International Conference on AI & Mobile Services - AIMS 2017* (Piscataway, NJ, 2017), S. Tata and Z. Mao, Eds., IEEE, pp. 38–45.

[44] Yousefpour, A., Fung, C., Nguyen, T., Kadiyala, K., Jalali, F., Niakanlahiji, A., Kong, J., and Jue, J. P. All one needs to know about fog computing and related edge computing paradigms: A complete survey. *Journal of Systems Architecture 98* (2019), 289–330.

[45] Zheng, G., Shi, L., and Kale, L. V. Ftc-charm++: an in-memory checkpoint-based fault tolerant runtime for charm++ and mpi. In *Cluster Computing, 2004 IEEE International Conference on, San Diego, CA, 04.12-04.12.2004* ([S.l.]n[s.n.], 20–), IEEE, pp. 93–103.

[46] zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)* (Santa Clara, CA, 2020), USENIX Association, pp. 209–223.

Assertion

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, October 19, 2020 Daniel Gomm