Received: 12 February 2021 | Revised: 1 October 2021 | Accepted: 5 November 2021

DOI: 10.1002/cpe.6759

WILEY

RESEARCH ARTICLE

Self-adaptation on parallel stream processing: A systematic review

Adriano Vogel[1,2] | Dalvan Griebler[1,3] | Marco Danelutto[2] | Luiz Gustavo Fernandes[1]

[1]School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil

[2]Department of Computer Science, University of Pisa (UNIPI), Pisa, Italy

[3]Laboratory of Advanced Research on Cloud Computing (LARCC), Três de Maio Faculty (SETREM), Três de Maio, Brazil

Correspondence
Adriano Vogel, School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil.
Email: adriano.vogel@edu.pucrs.br

Funding information
Conselho Nacional de Desenvolvimento Científico e Tecnológico, Grant/Award Number: 437693/2018-0; Coordenação de Aperfeiçoamento de Pessoal de Nível Superior, Grant/Award Number: 001; Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul, Grant/Award Numbers: 19/2551-0001895-9, 21/2551-0000725-7; Open access funding provided by Universitá di Pisa within the CRUI and Wiley Agreement

Summary

A recurrent challenge in real-world applications is autonomous management of the executions at run-time. In this vein, stream processing is a class of applications that compute data flowing in the form of streams (e.g., video feeds, images, and data analytics), where parallel computing can help accelerate the executions. On the one hand, stream processing applications are becoming more complex, dynamic, and long-running. On the other hand, it is unfeasible for humans to monitor and manually change the executions continuously. Hence, self-adaptation can reduce costs and human efforts by providing a higher-level abstraction with an autonomic/seamless management of executions. In this work, we aim at providing a literature review regarding self-adaptation applied to the parallel stream processing domain. We present a comprehensive revision using a systematic literature review method. Moreover, we propose a taxonomy to categorize and classify the existing self-adaptive approaches. Finally, applying the taxonomy made it possible to characterize the state-of-the-art, identify trends, and discuss open research challenges and future opportunities.

KEYWORDS

autonomic computing, distributed systems, parallel computing, self-adaptive systems, stream processing

1 | INTRODUCTION

The continuous increase in the number of connected devices (e.g., sensors, cameras, and radars) communicating through the network results in a large amount of streaming data being generated. Therefore, stream processing has become one of the mainstream research areas in computer science. The programmer/developer has several challenges to guarantee the quality of service in different application types (audio, media, video, and data analytic processing).[1-3] These challenges are relative to proper targeting of the underlying computer architecture and performance requirements. Hence, new programming frameworks, APIs, and languages are investigated.[4]

The underlying computer architectures have several processing units that can execute applications in parallel and accelerate their executions. This task is known as parallelism exploitation and tends to increase the level of programming complexity.[5] To efficiently use the parallel resources, the programmers have to deal with different parallel programming models, memory hierarchy, communication, synchronization, load balancing, scheduling, as well as heterogeneity management.[6,7] Unfortunately, exploiting parallelism in stream processing is still complicated for application programmers with the available programming alternatives. Therefore, parallelism abstractions are necessary to increase application programmers' productivity by focusing only on their business logic code.

Stream processing applications are subject to changing conditions and fluctuations (e.g., workload, input rates, and environment) while they are running (at run-time).[3] In this context, a configuration that sustained some quality of services can become instantly suboptimal or outdated.[8] Additionally, the unbounded data arrival requires stream processing applications to execute for long/infinite time periods. Therefore, applying adaptation actions online at run-time can improve responsiveness to changes.[9]

Consequently, new techniques are being developed to cope with scenarios that suffer changes at run-time, where a relevant example is self-adaptation.[10-12] Self-adaptation can be broadly defined as the capability of the systems/environments to be autonomous, deciding and changing their behavior in response to fluctuations. In this vein, one can change many entities for achieving self-adaptiveness, for example, in stream processing, self-adapting entities such as the batches size, and the parallelism degree for pursuing a given performance or efficiency. Importantly, self-adaptation can be viewed as a broader context encompassing many optimizations and entities. For instance, auto-scaling can be a facet within self-adaptation as the management of computing resources can be autonomous using mechanisms for providing elasticity.

Self-adaptation can make stream processing executions more intelligent, reducing human efforts, and assisting in error-prone activities by avoiding incorrect configurations. Despite this great potential of using self-adaptation in parallel stream processing, there are still challenges in implementation and validation. The design and practical implementation of self-adaptation have been reported to be difficult.[4,13] Considering that self-adaptation can affect execution's safety and cause overheads,[14] the evaluation and validation of the proposed self-adaptive solutions for stream processing are of paramount importance. Thus, from the parallel stream processing standpoint, we argue that there is a demand to understand better what is being proposed and how the existing solutions are validated.

Self-adaptiveness can also be used for providing additional parallelism abstractions to application programmers, which is a potential opportunity to simplify the process of running stream processing applications.[15,16] However, it is an open question to what extent self-adaptiveness is being applied and how efficient it is for providing parallelism abstractions. This work provides a systematic literature review (SLR) of self-adaptive approaches used in the stream processing scenario. Notably, we focus on the techniques employed, the support for parallel executions, and the validation of the proposed solutions. Also, we survey the literature regarding the use of self-adaptiveness for providing parallelism abstractions. In summary, we provide the following main scientific contributions:

- Categorizations of self-adaptation characteristics, parallelism properties, and validation categories in existing tools/frameworks for self-adaptive executions in parallel stream processing.
- A unified taxonomy for categorization and validation of self-adaptation in parallel stream processing.
- A catalog defining self-adaptation goals and entities managed.
- A discussion of open research challenges and perspectives for future enhancements on stream processing with self-adaptiveness.

This article is organized as follows. Section 2 overviews the background scenario and discusses related studies. Section 3 describes the SLR research method. Then, Section 4 presents the proposed categorization and taxonomy. Section 5 shows the results and evaluates the literature attempting to answer our research questions, and Section 6 provides perspectives of open research challenges. Finally, Section 7 discusses potential threats to validity, and Section 8 concludes the article.

2 | BACKGROUND

This section evinces concepts that are relevant to this study's context. We introduce stream processing and parallelism aspects. Also, we describe the foundations of self-adaptation and we discuss related surveys.

2.1 | Stream processing

Stream processing applications can be defined as programs that continuously compute data items. A stream is a given input that arrives from sources in the form of an infinite sequence of items.[1] Examples of stream sources can be equipment (radars, telescopes, cameras, among others) and file bases (text, image). Moreover, processing stages (a.k.a. operators) are entities that consume the incoming streams by applying computations. Usually, a stream processing application is composed of stages that communicate between them and provide results in a timely manner. The stages tend to be organized as a graph where each stage performs specific computations and the stream item flows through the graph using communication channels. Currently, a large number of computing applications characterizes the processing of streams of data in real-time or near real-time.

The characteristics of stream processing applications vary depending on the data source and computations performance. One of the most relevant aspects is the continuous and unbounded arrival of data items.[3] Lately, we have witnessed a significant increase in the number of devices producing data to be processed in real-time.

The generic concept of stream processing systems was categorized by Röger and Mayer[2] in two classes: general stream processing systems and complex event processing (CEP) systems. General stream processing represents the broad concept, encompassing systems and applications that may receive different data types and produce a result in real-time. CEP systems, on the other hand, concerns systems dedicated to detecting information from streams coming in the form of events. CEP is highly related to the processing of data coming from Internet of Things (IoT) devices, where each stream of data that is sent from a given sensor is handled as an independent event. In this work, we focus on the general stream processing, but our results can be also applicable for CEP systems.

2.2 | High-level parallelism in stream processing

Considering that performance is a very relevant concern of stream processing applications, parallelism is one of the most important aspects. Currently, parallelism is everywhere as computer systems support many techniques (e.g., vector instructions, multicore processors, co-processors). However, automatic parallelization approaches for sequential codes is still mostly unachievable,[7] exploiting the machine's resources requires parallel programming.

Dealing with parallel mechanisms like threads, synchronization, and dependency control tends to be a challenging task for application programmers who are not experts in performance. Consequently, supporting parallel execution results in application programmers facing a trade-off between coding productivity and application performance. This occurs because parallelism increases the performance, but the efficient use of parallel mechanisms/resources is usually time-consuming.

The use of high-level parallel programming methodologies is a potential alternative to provide coding abstractions for application programmers.[5] Abstractions can reduce the application programmers' burden. The main goal of high-level parallel programming can be defined as reducing programming efforts while ensuring a reasonable performance and portability. High-level abstractions tend to be provided by approaches that hide from programmers the complexities related to parallelism.[6]

There are two related concepts attempting to raise the abstractions for parallel programming in a more structured mode: parallel patterns[7] and algorithmic skeletons.[17-19] The basic idea is to provide for programmers recurrent constructors (skeletons) to be used for modeling parallel applications, where the common goal is to increase the coding productivity. Such constructors are called *patterns*, where several patterns may exist with different communication models, synchronization techniques, and task execution.[7]

Structured parallel programming may be viewed as a methodology that uses libraries or languages to facilitate coding. A usual approach toward structured parallel programming consists in using parallel patterns, which are well-accepted concepts that emerged from best coding practices in software engineering for optimizing and reusing specific code parts. The term parallel pattern refers to how the task distribution and data access are used recurrently in the design of a parallel program.[7]

Patterns are expected to be generic and universal, which can theoretically be implemented in any programming interface.[7] Consequently, flexibility is targeted by providing a vocabulary with several different patterns. In this study, important examples of patterns are map, pipeline, and farm.[7,19] A map can be simplistically defined as a function replication of elements that are processed in collections separated by indexes, where the replication is mostly suitable for independent elements. A pipeline is a pattern handling tasks in a producer-consumer fashion (like an assembly line) that is composed of connected stages. In a pipeline pattern, the data items flow through an acyclic graph, where each stage performs different computations/tasks.

The farm is a pattern composed of a pipeline. While a pipeline is composed of sequential stages, a farm has one or more parallel (replicated) stages. A farm has also at least one stage called emitter (A), which gets the input tasks and sends them to the next stage, according to a scheduling policy. In a farm, the stage following an emitter is usually fissioned (a.k.a. replicated) with a number N of parallel agents (called workers or parallelism degree), where N is the parallelism degree. A collector (C) gathers the tasks from the workers and places them into the output stream. In the context of stream processing, the stream items flow through the graph, continuously gathering input items as well as producing output results. The farm pattern used in a parallel execution characterizes a composition, which can be seen as the graph topology of the application. In Figure 1, we show a representation of parallel patterns, where data items can be seen as elements to be processed and tasks are a generalization of computations that are executed (e.g., by a thread). Moreover, parallel patterns can be semi and arbitrary nested to compose new parallel patterns.[7]

These parallel patterns have been implemented with programming abstractions as languages and APIs for parallel stream processing.[5] Some of them are known as distributed stream processing engines (SPEs), for instance, Apache Storm,[20] Apache Spark,[21] and Apache Flink.[22] These SPEs are designed for large scale clusters using Java Virtual Machine (JVM) to provide hardware and communication abstractions. There are also languages/frameworks for multicore parallelism exploitation, such as Intel TBB,[23] FastFlow,[24] GrPPI.[25] Additionally, there are domain-specific languages for exploiting stream parallelism, like StreamIt[26] and SPar.[27] Particularly, SPar is a stream annotation-based language while StreamIt is a new stream-based language. Both SPar and StreamIt aim to simplify the parallelization of stream processing applications with higher-level parallelism abstractions. However, although complementary surveys[2,4] addressed parallelism aspects on existing tools/frameworks, the current support for parallelism self-adaptation at run-time demands more analysis of the state-of-the-art.

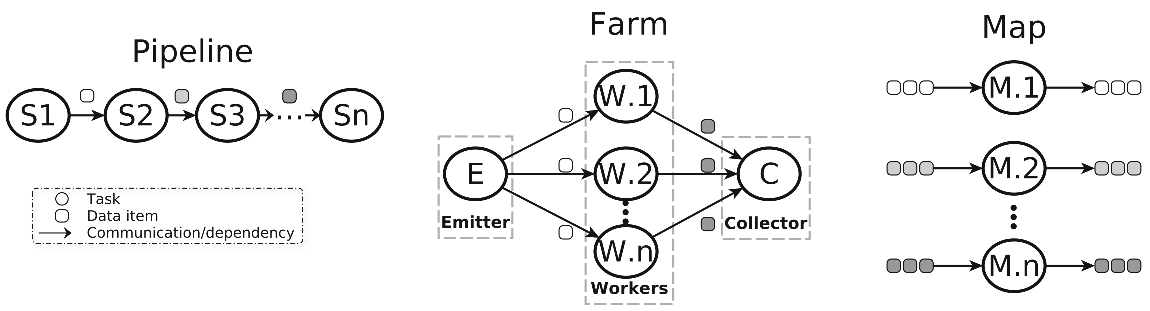

FIGURE 1 Representation of parallel pattern examples

2.3 | Theoretical foundations of self-adaptation

The software engineering field has been evincing that modern software systems/applications should operate in dynamic conditions without downtime.[10,28,29] Concepts of self-adaptation are implemented in systems to use the information to self-adjust. In this work, foundations of self-adaptivity are relevant for autonomous management of parallel stream processing applications. In this section, we provide an introduction to self-adaptive concepts and properties.

The definition of self-adaptive concepts varies in the related literature.[10,28,29] In Reference 28, self-adaptation was described as a characteristic of a system that "is able to adjust its behavior in response to their perception of the environment and the system itself." Although other similar definitions are available, this definition allows us to reason about the properties of self-adaptive systems. Considering that the relevant aspect of the self-adaptation is *how* to adapt, Weyns[29] proposed a conceptual model of self-adaptive systems that encompasses the following parts and properties:

- *Environment* refers to the "world" that the system or application runs inside. The environment tends to be a part not under full control by the system, which is subject to uncertainties and variations. For instance, the environment of a parallel application is the operating system and the running machine architecture.
- *Managed system* is the entity controlled and changed by adaptive components (sensors, actuators). The adaptation actions are expected to be applied while ensuring safety to the managed system. For instance, in the case of a self-adaptive parallel application, the managed system is the business logic code of the application that is expected to produce correct results.
- *Adaptation goals* are (service level objectives) SLOs or constraints handled by the self-adaptivity. There are four main types of goals:[29,30] self-configuration (a system that arranges itself), self-optimization (autonomously optimizing the execution), self-healing (seamlessly detecting and repairing issues), and self-protection (a system that transparently defends itself from problems). For instance, a parallel application can have an adaptation goal to increase its performance or the efficiency of the resources usage by applying self-optimization goals.
- *Managing system* characterizes policy, strategy, or mechanism that controls the managed system. The adaptation goals are used for deciding which actions to take, and the low-level constraints of the managed system for defining the aspects that can be adapted in the system.

The workflow of sensing and applying adaptations characterizes a feedback loop. Control engineering proposed the use of feedback loops as entities for providing adaptation capabilities. Moreover, feedback loops are conceptually essential entities for enabling self-adaptation to computing systems. Moreover, control engineering and control theory attempt to automate systems[11] by using concepts like feedback loops.

2.4 | Complementary surveys

Before conducting the SLR on self-adaptation applied to parallel stream processing, we researched the related literature to find existing efforts. Here we describe, discuss, and compare the complementary approaches found in the literature. Regarding environments for running stream processing applications, de Assunção et al.[4] and Klinaku et al.[31] presented reviews of efforts for stream processing on elastic cloud and edge environments. de Assunção et al.[4] surveyed environments, architectures, and programming models. Klinaku et al.[31] introduce a systematic mapping of approaches for elastic data streams running specifically in cloud environments. Data stream processing applications are also tacked by Roger and Mayer[2] with a survey of parallelism and elasticity. Additionally, Qin et al.[32] provide an SLR regarding adaptation for latency optimizations in data stream processing. Herodotou et al.[33] propose a review related to parameter tuning in batch and stream processing systems where they classified the approaches in six categories. Importantly, one category is adaptive parameter tuning that is one possible facet of self-adaptation for stream processing applications.

TABLE 1 Comparative summary of related surveys on self-adaptive stream processing

		Parallelism		Self-adaptiveness			Validation	
Reference	SLR	Tool	Abstractions	Entities	Information	Decision	Variations	Overhead
de Assunção et al.[4]		√		√	√			
Roger and Mayer[2]		√		√	√			
Klinaku et al.[31]				√		√		
Qin et al.[32]	√			√			√	
Herodotou et al.[33]		√		√	√	√		
This work	√	√	√	√	√	√	√	√

Compared to related surveys, to the best of our knowledge, this is the first SLR on the field of self-adaptation for parallel stream processing. In this context, we believe that our review is wider and at the same time addresses novel aspects. This work differs from Roger and Mayer,[2] de Assunção et al.,[4] and Klinaku et al.[31] in two main aspects. First, our scope is broader because elastic aspects tackled by Roger and Mayer,[2] de Assunção et al.,[4] and Klinaku et al.[31] can be viewed as only one possible optimization of self-adaptation, where elasticity typically concerns cloud scaling in terms of computational resources.[34] Consequently, our SLR considers additional scenarios that are discussed in our particular research scope. Second, we conducted an SLR with an automated search on different databases and digital libraries. Although an SLR is still having threats to validity (Section 7), it significantly reduces the risk of only mentioning approaches known by the authors of the survey.

This work has a different scope and revised literature in comparison with Qin et al.[32] and Herodotou et al.[33] Contrasting with Herodotou et al.,[33] this survey focuses on comprehensively reviewing approaches that enable self-adaptation for real-time stream processing. Qin et al.[32] considered only latency of application as a goal for optimizations. This work, on the other hand, reviews all metrics and goals that can be pursued by using/applying self-adaptation. Moreover, we focus on adaptation actions performed at run-time (online) because this is of paramount importance for stream processing, while the approaches revised by Qin et al.[32] and Herodotou et al.[33] do not necessarily adapt at run-time.

Table 1 shows an overview of the related surveys that considered adaptation on parallel stream processing. Section 4 provides specific definitions of the categorization criteria and relevant features used in Table 1. It is important to note that this categorization and features are considered relevant in the scope of these works, which does not necessarily cover the universe of stream processing features. Table 1shows that this work is novel in considering the parallelism abstraction topic, which is relevant for the use of self-adaptivity in facilitating the autonomous management of stream processing applications. Another novel topic considered here concerns how the current approaches are being validated, important for the use and reliability of self-adaptivity. In short, this work provides a novel categorization review of self-adaptive approaches (Section 4.1), a categorization of parallelism properties and abstractions (introduced in Section 4.2), and a categorization of properties for the validation of self-adaptivity (Section 4.3).

3 | RESEARCH METHOD

We conducted an SLR to map and comprehend the current state-of-the-art regarding self-adaptation applied to parallel stream processing. Our research scope considers the known characteristics of stream processing applications of being dynamic, irregular, and unbounded. Self-adapt executions considering the aforementioned characteristics are a potential solution for improving the applications to different goals and scenarios. First, the research questions are presented. Then, we contextualize our search strategy and protocol.

3.1 | Research questions

Considering the research scope and research problem, we distilled the following research questions (RQ):

- RQ1: Which are the publication's goals when applying self-adaptation to stream processing?
- RQ2: Which entities that enact adaptation are being dynamically adapted in existing solutions?
- RQ3: Which information is considered for performing adaptation?
- RQ4: How is the adaptation decision-making performed?
- RQ5: What parallelism aspects are being exploited in self-adaptive solutions for stream processing?

- RQ6: Are the approaches focusing on providing parallelism abstraction to application programmers when applying self-adaptation?
- RQ7: Which experimental procedures are used for validating the proposed solutions?
- RQ8: Which variations (e.g., workload, application) are considered for evaluating the proposed solutions in the experiments?
- RQ9: Are the approaches considering the overhead that adaptation may cause?

RQ1 aims to show up the main goals for applying software adaptation to stream processing. RQ1 also aims to provide a better understanding of the motivations and potential advantages of self-adaptation in the context of stream processing.

RQ2, RQ3, and RQ4 concern internal aspects of the self-adaptive solutions available. As parameters or system settings may be changed, it is very relevant to list what each solution is changing at run-time. Moreover, for performing adaptation, it is required that an entity decides for it. Thus, an entity has to consider some information for deciding what change to make, RQ3 aims at answering which information or statistical data is used in the decision-making step. Additionally, RQ4 attempts to unveil which mechanisms are used on specific scenarios for actually performing adaptation actions.

RQ5 relates to how parallelism is being exploited on self-adaptive solutions, specifically the framework/language used and which architecture/environment is the solution targeting. Moreover, a relevant aspect for parallel stream processing is how many replicated stages the self-adaptive solution is able to manage, where in this broad SLR different scenarios of stream-like applications are being considered (e.g., data stream, CEP, streaming systems). Considering the importance of providing high-level parallelism abstraction described in Section 2.2, RQ6 focuses particularly on evaluating if the self-adaptation properties can be used/adapted for providing parallelism abstractions for application programmers.

Another relevant aspect of self-adaptation is how to validate a proposed solution. RQ7 seeks representative aspects used for evaluating self-adaptive solutions. Additionally, a representative and the broad experimental setup are expected to properly evaluate the quality of self-adaptive solutions. Thus, RQ8 intends to identify whether the solutions are being comprehensively evaluated regarding variation that can be application characteristics, workloads, environments, and so forth. Yet concerning the validation of the proposed solutions, the overhead caused by the self-adaptation is very relevant. RQ9 has this concern, looking at whether the validation of the solution is considering the potential overhead of adaptation.

3.2 | Search strategy

The search strategy is composed of incremental steps. The first step was to elaborate on the search string that was used for automating the search. As recommended by Kitchenham's guidelines,[35] the research questions of this literature review were separated into facets related to three main aspects: stream processing, self-adaptiveness, and parallelism. We defined five control studies[36-40] that were previously known as relevant primary studies from our previous works.[8,15,16,41,42] Pilot searches were conducted on Scopus with the different terms and synonyms for testing if the string found the control studies. Thus, we converged to a search string evinced in Table 2 with the terms of the three main aspects separated by boolean ANDs, where these terms searched studies' titles, abstracts, and keywords.

In the area of stream processing, we included relevant terms known to represent stream processing characteristics and paradigms. In the self-adaptation properties, we included also terms related to "autonomic" that can be seen as a way to achieve self-adaptation. Regarding the parallelism scenario, we included also terms like the ones related to concurrency because it is used interchangeably with parallelism in some works. In short, we included similar known terms in order to find a high number of relevant studies. But at the same time, we avoid broader terms that bring irrelevant papers. In Section 7, we discuss a compromise between these two objectives.

The next step was running the search string in the most relevant research databases and digital libraries for finding primary studies. The research databases used are Scopus[*] and Web of Science.[†] Moreover, the two digital libraries, namely ACM Digital Library[‡] and IEEE Explore.[§] The search string was defined according to the systematic review procedure adapted to the syntax of each repository. The search string was executed in November of 2020.

In the third step, the titles and abstracts of the papers were read and filtered accordingly to the inclusion and exclusion criteria presented in Section 3.3. Then, we performed a skimming step, similar to what was performed in related surveys,[32] covering a full-text view (figures, tables, flowcharts, graph results) of the papers. In the fifth step, we read the full papers for performing the final decision. This was the most intensive step, where we critically double-checked the paper considering the inclusion and exclusion criteria.

TABLE 2 Search terms

Area: Stream processing	Scenario: Parallelism	Property: Self-adaptation
"stream processing" OR "data stream" OR "complex event processing" OR "streaming application" OR "streaming system"	"parallel" OR "concurren*" OR "scal*"	"adapt*" OR "autonomic" OR "autonomous" OR "elastic" OR "on-the-fly reconfiguration" OR "online reconfiguration" OR "self-*" OR automatic scaling

3.3 | Study selection criteria

The following criteria were used for filtering the papers:

- *Inclusion criteria 1*: The study applies self-adaptation properties to stream processing. Rationale: we include technical studies that do not explicitly mention self-adaptive systems, but that encompass self-* properties.
- *Exclusion criteria 1*: Not a scientific paper that is not written in English or that is a short version (e.g., editorial, abstract, poster). Rationale: This type of study lacks in space for proposing and validating relevant solutions.
- *Exclusion criteria 2*: A publication that has no self-adaptation aspects, concerning only stream processing. Rationale: Our focus is on adaptive properties applied to stream processing.
- *Exclusion criteria 3*: A publication that has no aspects related to parallelism, that is, only concerning self-adaptation. Rationale: Considering that parallelism is a pervasive topic that is relevant for several optimizations, we focus on studies addressing self-adaptation in parallel systems.
- *Exclusion criteria 4*: Publications that are not considering stream processing. Rationale: We focus on solutions for the stream processing context where applications have unique characteristics, that is, long-running and dynamic.

4 | SELF-ADAPTIVENESS APPLIED TO PARALLEL STREAM PROCESSING: CATEGORIZATION, TAXONOMY, AND CONCEPTUAL FRAMEWORK

Here, we propose categorizations and taxonomy to organize and extract relevant results from the literature. In Section 2.4, one can note that there are some classifications of studies addressing self-adaptive properties in the context of parallel stream processing. This article intends to unify these existing classifications into a taxonomy. Moreover, we propose new categories to improve the categorization of proposed solutions and answer our research questions. Figure 2 depicts the structure of the proposed taxonomy that has three main classification groups described in the next sections: self-adaptiveness (Section 4.1), parallelism properties (Section 4.2), and validation of the proposed self-adaptive solutions (Section 4.3).

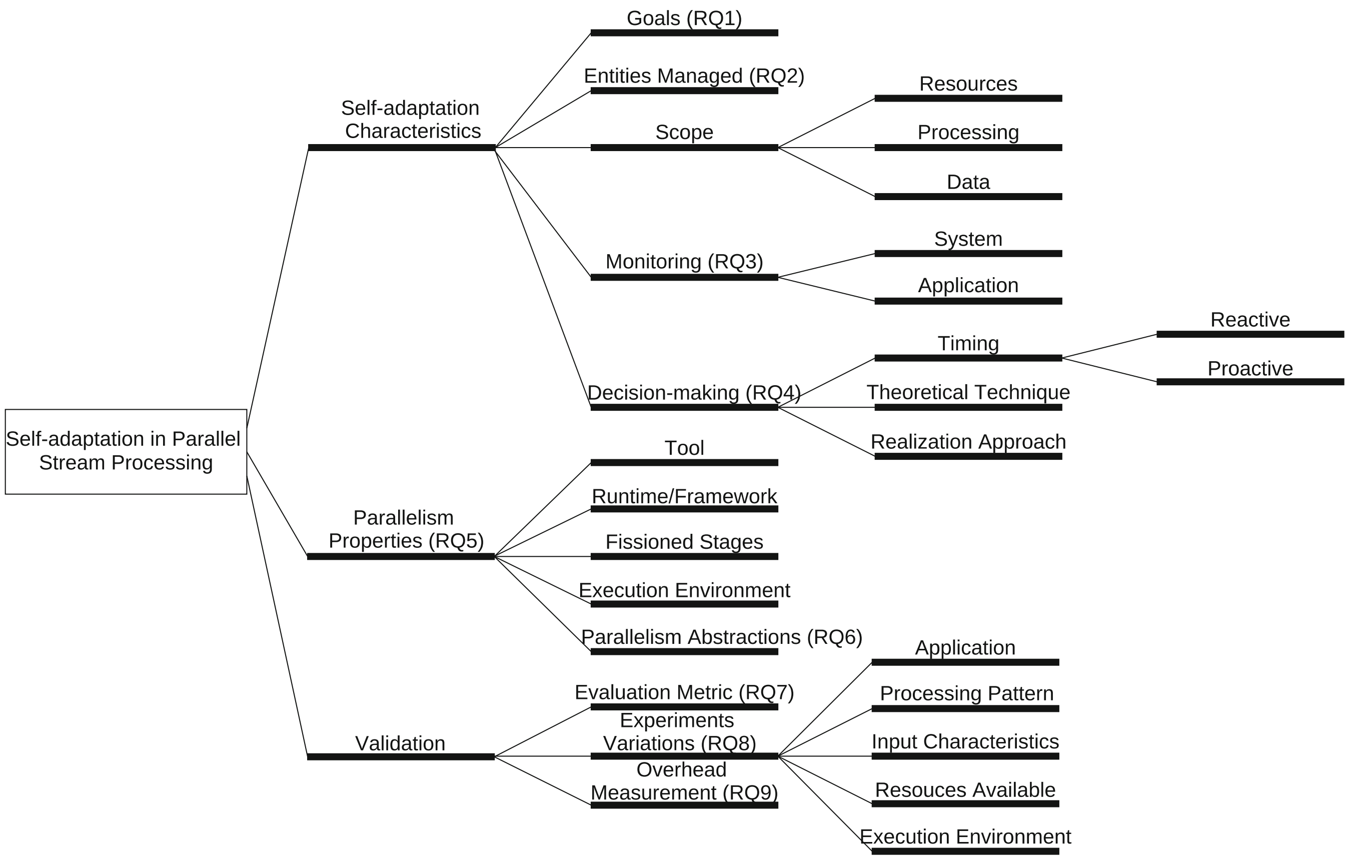

FIGURE 2 Proposed taxonomy for self-adaptiveness in stream processing

4.1 | Self-adaptation categories

The categorization presented here concerns self-adaptiveness foundations[10] and we divided the self-adaptation characteristics into three major properties described below.

4.1.1 | Category of adaptation

One category is the adaptation goals (RQ1) and the entities managed in adaptation actions (RQ2) for pursuing a given goal. Regarding where the adaptation is performed, Qin et al.[32] provide a taxonomy that helps in our scenario. Here, the scope of adaptation considers three adaptation classes: resources, data, and system processing. *Resources adaptation* concerns the modifications applied only to the physical execution environment that is agnostic to the application/system. For instance, adapting the amount of CPU computing resources available while a given application is running. *Data adaptation* is related to modifying the data stream/items that are the application is processing, such as adapting the size of data batches. The *processing adaptation*category covers adaptations performed in the application processing entities. Examples of this category are changes applied in the application processes/threads (parallelism degree) and task scheduling.

4.1.2 | Monitoring (abbreviated: *Mon.*)

Considering that the managing system needs updated information and statistics to self-adapt the executions, monitoring is a potential way of feeding the system to measure/evaluate the execution. In this work, we divide monitoring into system and application level, which is a categorization related to RQ3. System-level relates to monitoring the operating system and hardware (the environment). Application monitoring relates to the collection of information from the application and its runtime systems, such as application performance metrics or indicators.

4.1.3 | Decision-making

The managing systems decide to perform adaptation actions, which enables a managed system to achieve self-adaptiveness. The alternative of applying an action to change a given entity considering a goal can make a system and the executions more intelligent. RQ4 considers how is the decision-making performed. As explained in Reference 2, the decision may have different timing, reactive or proactive. A reactive decision responds to a specific scenario, while the proactive one attempts to anticipate a given occurrence. Additional categories that we created for evaluating the decision-making regards the *theoretical technique* and *realization approach* used. Considering that designing and implementing support for adaptation to real-world stream processing applications tends to be complex, there are several theoretical foundations available.[10,11] Thus, the category *theoretical technique* attempts to survey which theories are being used for designing the self-adaptive strategy. Moreover, the *realization approach* category is the core of the decision-making that is represented by a decision algorithm and control mechanisms.

4.2 | Categorization of parallelism properties

Considering that parallelism is relevant for stream processing applications and the RQ5, we also propose a categorization of the properties concerning the parallelism exploitation in self-adaptive approaches.

4.2.1 | Tool

This category concerns which existing tools are supporting self-adaptive parallelism in stream processing. It is also important to note that the existing runtime or frameworks can be extended by approaches proposing new tools that enable self-adaptiveness.

4.2.2 | Runtime library or framework

A relevant aspect to survey the literature relates to which existing parallel runtime libraries or frameworks have features supporting self-adaptation. Our SLR attempts to list all parallel tools/frameworks that support self-adaptation in the context of stream processing.

4.2.3 | Fissioned stages

Figure 1 shows a farm with one fissioned (a.k.a replicated) stage, which represents a graph composition or application topology.[41] Such a graph composition is suitable for embarrassingly parallel computation that easily executes in parallel. However, there is a trend in software applications to be more complex with several functions that can be decomposed using parallel patterns, resulting in robust graph compositions. We consider robust compositions the ones comprising more than one fissioned stage or a combination of patterns (e.g., a pipeline with farms). The property of fissioned stages aims at surveying how many replicated stages are supported by the existing self-adaptive approaches/tools. Manage a single fissioned stage tends to be less complex. On the other hand, robust compositions with multiple fissioned stages tend to require additional control mechanisms for managing the execution at run-time.

4.2.4 | Execution environment

The execution environment category inside the group parallelism properties considers the characteristics of the execution architecture of a given stream processing application, which is a very relevant aspect for parallelism. A single machine environment can be composed of a multicore or heterogeneous architecture (co-processors, GPUs, FPGAs). The environment may also be a cluster with a distributed shared-nothing memory subsystem. The target environment used tends to be related to performance requirements. Some applications running in a multicore machine can sustain a suitable QoS while other applications may require several machines for achieving QoS.

4.2.5 | Parallelism abstractions

This category evaluates if the self-adaptation employed is able to optimize stream processing aspects for providing abstractions to users/programmers. Examples of abstraction can be a parallelism abstraction (Section 2.2), attempting to simplify the execution, or self-optimize specific concerns. Providing abstraction can be an advanced goal of self-adaptiveness because it tends to be difficult to facilitate the application programmers' tasks by transparently managing the systems. Additionally, considering stream processing and parallelism abstractions, RQ6 attempts to conceptually highlight whether parallelism abstractions are being considered for self-adaptive stream processing.

4.3 | Self-adaptiveness validation

A relevant aspect related to RQ7 is how the literature approaches available are validated. Although self-adaptivity can provide several advantages, we expect the approaches to be properly evaluated for maintaining performance and execution safety. In Reference 32, evaluation metrics were extracted from the papers found in their context. In this work, we intend to extract relevant information and evaluate the current validation state of the approaches. Hence, we propose a categorization to assess the characteristics and variations used in the validation. Below we describe the categories and properties covered by the proposed categorization.

4.3.1 | Evaluation metric

Considering that the self-adaptive solutions are expected to be extensively evaluated, this category intends to survey the evaluations metrics. Examples of metrics to be considered are performance, energy, and resource consumption. For instance, the performance metric can be utilized to measure the effectiveness of the self-adaptive solutions, where examples of relevant metrics are execution time, throughput (how many stream items are computed per second), and latency (time taken to compute the stream items).

4.3.2 | Experiments variation

Experiments variation is a relevant concern related to RQ8. Considering that a representative and broad testbed is expected to be used for evaluating the proposed solution under different scenarios, this category is divided into subcategories for better assessing the approaches:

- *Application* considers whether different applications were used for characterizing and evaluating the behavior of the proposed approaches. The self-adaptive managing systems are supported in applications that are executed for evaluation purposes.

- *Processing pattern* considers whether the tested applications present changes in behavior or processing characteristics. In our context, we consider as relevant examples of different processing patterns if the applications have a different performance trend with respect to the parallelism (degree, level, grain, scheduling, placement) used or processing characteristics (e.g., CPU bound, memory-bound, I/O bound).
- *Input characteristics* evaluate if different input types or rates were considered for the applications used.
- *Resources available* considers if the number of computational resources available for the running applications changed during their executions. This is a concern of applications running in modern dynamic environments.
- *Execution environment* is a variation related to evaluating the approaches in more than one scenario/environment. An example can be executing the application and characterizing the self-adaptive decision-making in different computational architectures (multicore, GPUs, FPGAs, clusters) or paradigms (cloud, fog).

4.3.3 | Overhead measurement

This category is related to the overhead (RQ9) that can be caused by self-adaptation, where it attempts to survey what is the impact on real-world applications. For instance, self-adaptation demands additional mechanisms and processing parts, which consume more computational resources. Moreover, being adaptive can impact positively or negatively in the performance of the applications. The overhead measurement intends to evaluate whether the related literature considers the drawbacks that self-adaptiveness can cause in some cases.

4.4 | Conceptual framework

Considering the generic theoretical representation of self-adaptive systems provided by Weyns[29] and the categorization showed in previous sections, we present a conceptual framework for our research context. This framework is used to review and categorize the literature. Figure 3 provides a representation of our conceptual framework, where the users/programmers specify adaptation goals that are used for a self-adaptive managing system.

The conceptual framework has a self-adaptive strategy/policy with decision-making (analyze, plan) properties and mechanisms for controlling specific entities. The entities are controlled for applying adaptation actions (execute). For instance, a low-level mechanism of a given runtime library applies a change to a knob or entity. Consequently, a strategy can be viewed as a generic solution while a mechanism is targeting a specific runtime library or environment. The self-adaptive managing system collects information (Monitor) from the environment and software layers for sensing

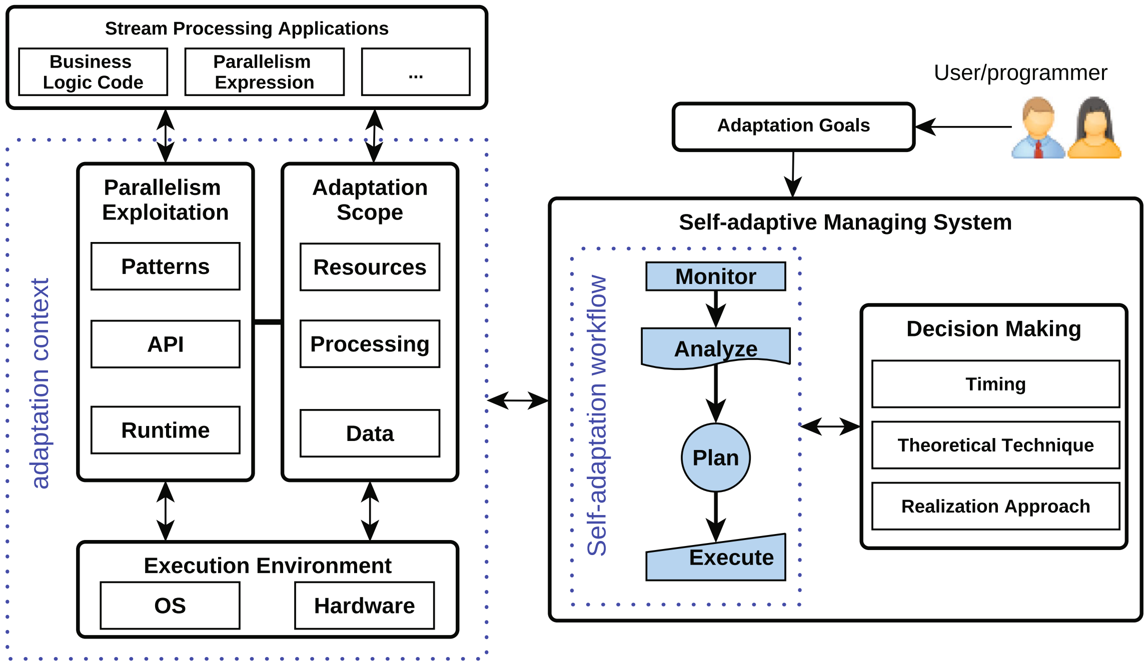

FIGURE 3 A conceptual framework for self-adaptive executions in parallel stream processing

changes and applying adaptation actions when necessary. As evinced in Section 4.1.1, the scope of adaptation has three classes (resources, data, system processing) used for applying adaptation.

Stream processing applications run on specific computational environments, where an environment characterizes at least an operating system and a given hardware architecture. In the software layer, right above the operating system, we have the runtime library or framework used for parallelism exploitation. Also, programmers can use application programming interfaces (APIs) and patterns (Section 2.2) for introducing parallelism in their applications.

The top part of our conceptual framework refers to the stream processing applications, which is a sensitive layer where the intrusion is mostly avoided. The application business logic code represents the regular code of a stream processing application. Moreover, the parallelism expression is the first step for the parallel execution. When expressing parallelism, the application programmers can define code regions that are suitable and profitable for running in parallel. In this sense, we argue that abstraction must be provided for application programmers since there is a significant gap between parallelism expression and an actual self-adaptive parallel execution. Effective parallelism abstractions can be achieved when self-adaptive systems regulate low-level mechanisms, which prevent users/programmers from performing non-intuitive/error-prone activities. In this context, parallelism abstractions can potentially increase programmers' productivity as well as providing system optimizations (additional performance or efficiency).

The conceptual framework is proposed for better explaining the research context and for evaluating the current approaches. The SLR research method evinced in Section 3 helps in identifying studies. This framework encompasses categorization taxonomies that are used for evaluating the literature studies. The characteristics provide an evaluation of how the current solutions are being designed and which features are currently supported. Consequently, the next section summarizes the solutions from the literature that are then categorized and discussed.

5 | RESULT ANALYSIS AND DISCUSSION

In this section, we discuss the results found in the literature with respect to the categorizations.

5.1 | Studies overview and their execution environments

This section presents an overview of the literature approaches that are separated according to their execution environment and ordered by publication year. The execution environment is classified according to the description provided in each study. The execution environments are classified in multicores, cluster, cloud, and heterogeneous environments with multicore and accelerators (GPUs or co-processors). It is important to note that cloud environments support multicore environments (a single instance) or a virtual cluster with multiple instances. Consequently, a parallel application running in a cloud environment can be still running in a multicore or cluster. However, as we focus on the environment instead of the programming model, studies that are executed in clouds are only presented in the cloud category. The same organization is applied to studies showed in the accelerators section because GPUs and co-processors are added to multicore machines that may or may not be part of a cluster.

5.1.1 | Multicores

The first approach proposing self-adaptiveness for stream processing applications was found in Schneider et al.[43] They proposed elasticity as an adaptation that extended SPADE,[44] a language and compiler for developing stream processing applications. The solution was implemented using a dispatcher thread, which manages the stream system (load distribution, queues). The dispatcher thread is characterized as a component that manages the number of active threads. A key component of the system is the alarm thread which runs periodically for monitoring the system.

The work of Choi et al.[45] proposed a solution for detecting performance bottlenecks with a performance model and adaptation algorithm. The proposed solution was validated in different scenarios comparing its performance to the related solution from Reference 43. The validation considered a different number of bottlenecks and a scenario with changes regarding the availability of computing resources. The proposed solution outperformed the related approach in terms of performance. It is also important to note that the solution of Reference 45 assumes a stable workload, where the application load has only one processing phase. Consequently, this approach only configures streams programs once.

Tang and Gedik[46] proposed an autopipelining solution for stream processing that transparently attempts to improve the application efficiency and throughput. It is important to note that the approach concerns a scenario where each thread executes a pipeline. Autopipelining was implemented with a runtime profiling that performs optimizations concerning the number of threads aiming to overcome application bottlenecks. The solution was evaluated on real-world applications with different tuple sizes.

Selva et al.[39] provided runtime adaptation for streaming languages. The StreamIt language was extended in order to allow the programmer to specify the desired throughput. The adaptation is provided by the runtime system that controls the environment. Moreover, an application and system monitor was implemented to check the throughput and system bottlenecks. The adaptation concerns the migration of actor (stage) according to the load on specific CPU cores.

Proposing the term "self-aware" to be applied in stream processing, Su et al.[38] introduced StreamAware. It is a programming model with adaptive parts targeting dynamic environments. The aim was to allow the applications to automatically adjust at run-time. The adaptivity implemented was based on the MAPE-K closed-loop. The adaptive parts proposed adjusts the runtime in three aspects: integration and removing of idle nodes and adjusting data parallelism. However, the mechanisms and policies used for adapting the programs at run-time are not described. The adaptive method is evaluated and validated using the PARSEC benchmark suite.

De Matteis and Mencagli[37] presented elastic properties for data stream processing to improve performance and energy efficiency (number of cores and frequency). They argue that using the maximum amount of resources is expensive and inefficient. Therefore, they proposed elasticity as a solution for efficient usage according to QoS requirements. The latency was managed using a model predictive control (MPC) method while the energy consumption was reduced with dynamic voltage and frequency scaling (DVFS) techniques. The solution was validated in a high-frequency trading application as well as compared to related solutions. De Matteis and Mencagli[47] extended Reference 37 with strategies for energy-aware on data stream processing.

Additional parallelization techniques for stream processing on multicores were presented by Gad et al.[48] There, a new domain specific language (DSL) was proposed. The self-adaptive part encompasses a mechanism providing data distribution optimizations among the CPU cores, where functions to be computed can be moved from processing elements. The solution was tested on "pleasingly" parallel tasks, which are easily parallelized and showed promising results.

Karavadara et al.[49] proposed a framework for stream processing on embedded systems with many-cores processors that were implemented in the library called S-Net. The adaptive part is performed by the control system that uses DVFS for reducing, at run-time, the power consumption. The DVFS proposed solution was evaluated showing significant energy savings.

Sahin and Gedik[50] proposed C-Stream, which is a SPE for customizable and elastic executions. C-Stream empowers users with the option to set SLA, such as high throughput or low latency. Importantly, the solution has an adaptation module that adjusts the number of threads and the level of parallelism. The main goal of adaptation was to detect bottlenecks and increase performance. The solution was validated with real-world applications and compared to Storm, where C-Stream achieved a good performance.

Schneider and Wu[51] presented an elastic scheduler for the IBM Streams system. The work tackled the problem of determining the best number of threads in stream processing. Particularly, they proposed an elastic algorithm that finds the number of threads that yields the best performance without user inputs. The solution was tested on different machines showing a good performance. Later, Ni et al.[52] provided an elastic threading model for IBM Streams.

De Sensi et al.[13,53] proposed Nornir, a framework applicable for stream processing. Nornir aims to predict performance and power consumption using linear regression as a learning technique. Their goal was to reduce power consumption with "acceptable" performance losses. Nornir enables the application to change different knobs at run-time. The Nornir system was aimed to satisfy power consumption or performance bounds, which have to be defined by the user. It then triggers actions when it detects changes in the input rate or application. Nornir interacts with the operating system (OS) and with FastFlow's runtime. Importantly, Nornir also includes a flexible framework that can be used for designing custom decision-making strategies as well as non-intrusive instrumentation of executions.

The impact of parallelism on the latency of stream processing items was addressed on Vogel et al.[54] There was showed that although more parallel replicas usually increase the throughput of stream processing applications, more replicas also increase the latency. The proposed solution was a compromise between latency and throughput, where the application programmer is expected to provide a latency constraint in the SPar DSL. With the latency constraint, the proposed solution controls the latency by autonomously managing the number of replicas, in such a way that the throughput is increased while the latency constraint is met. The solution was validated with a real-world application showing its effectiveness.

Griebler et al.[55] designed SLO attributes for the user to express the target QoS. The solution combines SPar for parallelization and Nornir for power-aware runtime. From the language side, SPar is able to generate parallel code from annotations added by the user on a sequential code. Nornir provides changes in terms of the number of cores and their frequency for enforcing a given power or performance SLO. The solution was validated with real-world stream processing applications showing its effectiveness. An extended version of this work was provided in Reference 16 for supporting additional SLOs as well as a comparison to related solutions.

The work of Kahveci and Gedik[56] proposed optimization for solving bottlenecks on stream processing applications. They proposed a development API and a runtime library in a system called Joker, which runs in multicore machines. The solution was validated with different applications and compared to related solutions.

The work of Vogel et al.[14] addressed another relevant facet of self-adaptiveness: converge faster to a suitable configuration and minimize instability for reducing the decision-making overhead. The proposed solution improved the decision-making step and the evaluation with different applications and workloads show performance gains as well as lower overhead.

In a distinct vein, Vogel et al.[42] provided completely seamless parallelism management for video stream processing applications. Higher parallelism abstractions were achievable thanks to a smarter decision-making strategy that detects workload change for adapting the parallelism degree. The seamless strategy was evaluated with different applications showing its effectiveness.

5.1.2 Clusters

The work of Gulisano et al.[57] proposed StreamCloud, a solution to scale data stream processing applications running in cloud environments. StreamCloud (SC) implements parallelism to process queries and distribute the tasks among nodes where a major concern is to handle load balancing through optimized task distribution. Parallelism optimizations are triggered considering the number of active nodes and their CPU loads. The evaluation shows that elasticity and dynamic load balancing optimized resources consumption.

Balkesen et al.[58] proposed a framework for actively managing parallelism configurations on SPEs. The framework attempts to optimize the parallelism configuration related to the cluster size where the decision considers the input events. Moreover, latency minimization was the main goal, which was optimized with load balancing. Additionally, the framework attempts to predict the future behavior of input streams. The proposed solution was integrated with the Borealis system. Although in the evaluation conducted the latency was significantly higher than static cluster size, the authors claim that the approach is effective by reducing resources consumption.

Yet another approach to stream processing is provided by Heinze et al.[59] It addressed the complexity of determining the right point to increase or decrease the degree of parallelism. The authors investigated issues of elasticity in the data stream to meet requirements for auto-scaling (scaling in or out). They explored the impact of latency in distributed processing. These authors categorize the approaches for auto-scaling applications in five groups: Threshold-based, time series, reinforcement learning, queuing theory, and control theory. They argue that time series is not feasible for adaptivity on stream processing applications, because it considers historical data and the stream load is unpredictable. The queuing model was also excluded due to its limited adaptivity. The remaining classes were then tested with stream processing applications. Threshold approaches are characterized by the need for the user to set upper and lower bounds with respect to the resource utilization and/or performance. On the other hand, reinforcement learning is based on the system state for taking optimization actions, using a feedback control that monitors the execution and chooses another configuration the next time. Control theory is based on an independent controller that responds fast to input changes based on a feedback loop.

Gedik et al.[36] tackled elastic auto-parallelization[40] to locate and parallelize parallel regions. They also address the adaptation of parallelism during the execution. Moreover, these authors argue that the parallelism profitability problem depends on workload changes (variation) and resource availability. They propose an elastic auto-parallelization solution, which adjusts the number of channels in their runtime to achieve high throughput without wasting resources. It is implemented by defining a threshold and a congestion index in order to control the execution regardless of if more parallel channels are required. This approach also monitors the throughput and adapts to increase performance. The experimental evaluation shows that the proposed approach performs adaptations and maintains a fair performance.

Wu and Liu[60] proposed DoDo that is a load-adaptive software layer. DoDo runs on top of cluster nodes and dynamically manages the stages distribution considering the load of the physical servers. DoDO attempts to increase the application throughput by improving the load balancing and resource utilization on the cluster nodes. The experimental evaluation conducted demonstrated performance gains with the proposed solution.

Chatzistergiou and Viglas[61] addressed job reconfiguration approaches for improving tasks distribution in stream processing. Their solution monitors the performance of running applications and reconfigures the jobs placement in case of bottlenecks. The implementation and validation of the proposed solution evinced throughput increases with synthetic and real-world workloads.

Martin et al.[62] proposed StreamMine3G, an event SPE that is scalable and elastic. A relevant part is the elasticity support that enables efficient processing under fluctuating workloads. The elasticity concerns the number of nodes used, and the decision algorithm simply monitors the utilization of the node and takes action in case of underloading or overloading.

Zacheilas et al.[63] proposed an approach for scaling performance in CEP systems. Their solution attempts to predict fluctuations in terms of input rates or latency. The technique used for prediction was Gaussian process. However, the prediction concerns the application case study and requires previous historical data for making predictions. The challenge is that it tends to be very application specific, is hard to generalize the prediction to other applications and workloads. Moreover, it takes a significant amount of time to train the model which performs dynamic adaptations. The solution was evaluated and showed performance improvements and a fair prediction's accuracy.

Lohrmann et al.[64] provided a reactive strategy for guaranteeing latency constraints on stream processing. The proposed solution was validated on Nephele system, where a queuing model estimated latency responses and performed scale actions when necessary. The solution was validated with synthetic and real-world applications. The solution was also compared to the state-of-the-art and achieved performance gains.

Mayer et al.[65,66] proposed a technique to timely adapt the degree of parallelism in CEP. The adaptation is performed aiming to limit the buffer size, where huge buffering is assumed to negatively affect the detection of the events. Consequently, a method was proposed to predict event rates and proactively adapt to the degree of parallelism. The validation of the proposed solution measured the queue lengths under different workloads, where the results were compared to a CPU threshold approach. The proposed solution showed to work well in the CEP scenario. However, it was not measured the impact of buffering in common performance metrics (throughput, latency, service time) of stream processing applications.

Heinze et al.[67] provided an elastic scaling technique focused on the trade-off between monetary cost and latency of stream processing. An online parameter optimizer was proposed for finding a scaling configuration that yields less cost. Moreover, the parameter optimizer attempts to facilitate usability by requiring the user to set only the expected service rate instead of several error-prone parameters. The evaluation of the solution evinced that it reduced costs and maintained a reasonable quality of services.

Adaptive fault tolerance for stream processing was addressed by Martin et al.[68] The proposed solution dynamically changes the fault tolerance scheme (e.g., active replication, active or passive standby, passive replication) aiming to facilitate for users that are not required to manually set the best replication scheme. Consequently, the users are expected only to provide high-level constraints, such as recovery time, gap, or precision. This is provided by the adaptive controller that sets and changes to the most suitable scheme considering the user constraints, workload, and the lowest resource consumption. The validation showed a low resource consumption overhead and without losses in recovery time.

Zhang et al.[69] aimed to minimize latency on stream processing by adapting the batch and block size. They proposed DyBBS, which uses a heuristic to learn and set the batch configuration according to the current workload. The solution was validated compared to related solutions showing that it reduces the latency.

Meet latency requirements on stream processing was also a goal of Liu et al.[70] There, the latency is controlled from the perspective of resource management, scheduling, and load balancing. Presuming load fluctuations, the authors proposed scheduling solutions for tasks redistribution aiming at reducing the latency and achieving more stability by avoiding overloading specific servers. The evaluation evinced the performance gains of the proposed solution.

Gil-Costa et al.[71] provided an elastic strategy that balances the load according to the processing time and the load of each specific node. Using a monitor and manager, the overloaded nodes are released while new nodes process the bottleneck stages. The evaluation showed a good performance considering the throughput of applications.

Li et al.[72] presented an approach for elastic scaling in distributed stream processing. The proposed techniques for scheduling stream processing with batch processing. The stream processing part was implemented on Storm. A very relevant aspect of this approach is that they highlight downtimes that occur in Storm when scaling actions are taken. The downtime occurs because Storms can only scale by reconfiguring and restarting the application that results in shutting down all active operators. Consequently, the data stored in operator's memory are lost which results in downtime that lasts from 20 to 30 s. Previous efforts working with adaptivity in Storm did not consider this issue. Consequently, the authors proposed a solution that saves the state of the operators for minimizing downtime when scaling the application. In order to decide when to perform elasticity actions, the proposed solution includes a monitor for congestion detection attempting to avoid application bottlenecks. The solution was validated showing gains in terms of latency and throughput.

Kombi et al.[73] provided an approach that monitors the congestion attempting to improve the performance and resources efficiency. The continuous dynamic adaptation was proposed as a solution. The solution encompasses an estimation of the future input size with time series analysis algorithms. This estimation has the potential to forecast the load for the near future. Additionally, they proposed a technique that evaluates and adapts the degree of parallelism. The solution was validated emphasizing improvements in latency and resources consumption. This work was extended in Reference 74 for improving the performance and stability.

De Matteis and Mencagli[75] extended References 37 and 47 for supporting execution on distributed cluster environments. They proposed a control-theoretic strategy for elastic scaling in data stream processing. The solution targets latency sensitive applications by providing an MPC. The solution was experimentally evaluated in terms of latency and reconfigurations.

Cardellini et al.[76] extended Reference 77 by proposing and evaluating optimizations regarding replication and placement of operators for stream processing. The model supports multiple QoS metrics (response time, inter-node traffic, cost, availability) and attempts to find a balance between those multiple metrics. Moreover, Cardellini et al.[78] proposed an elastic distributed framework for stream processing applications. The important self-adaptive part concerns tasks migration and adaptation in the degree of parallelism. A threshold-based policy and two reinforcement learning (RL) policies were implemented for adapting the configurations. One RL policy is Q-learning that uses a cost function for learning from samples and estimating optimal actions. The second policy is a model-based RL algorithm that exploits different system knowledge for estimating an approximate configuration.

It is important to note that the policy proposed by Cardellini et al.[76] requires configurations and parameters from the user, such as cost weights, scaling thresholds, and response time. Such parameter definitions may be difficult and error-prone for application programmers. The proposed policies were implemented in Storm, which is worth mentioning the known downtime on reconfigurations.[72,79] The validation covered a real-world application, where it was noted limitations in the resources threshold policy and that the Q-learning policy requires too much time to learn and find a suitable configuration. The model-based RL policy was the best performing solution in the evaluation conducted.

Cheng et al.[80] provided a new scheduler for Spark Streaming. This new scheduler is adaptive in a way that dynamically schedules and adapts the parallel jobs. Adaptivity makes it possible to reconfigure the execution with the implemented solution in such a way that the level of parallelism (number of concurrent jobs) and the sharing/availability of resources. The approach was evaluated on a security event application showing improvements in terms of performance and energy efficiency.[80] was extended in Cheng et al.[81] for supporting dynamic batching.

Lombardi et al.[82] proposed Elysium, an approach for elasticity on stream processing. The novelty of Elysium concerns evaluating, estimating, and managing elasticity in two independent dimensions: application parallelism and resources. The solution also presents a resource estimator that computed the expected resource utilization, which proactively runs elasticity actions. The solution was validated with real-world applications showing gains in terms of elasticity adaptations.

Kalavri et al.[83] addressed a very important aspect of online adaptiveness: find fast and accurate new configurations. They proposed the DS2 controller that has a performance model which estimates the true processing capacity of stages and converges faster, accurately, and in a stable mode to a new parallelism configuration. DS2 was implemented as a decoupled decision-making that was implemented in two frameworks, Flink and Timely. The experimental results show a fast convergence and a low monitoring overhead.

Wang et al.[84] proposed Elasticutor for faster elasticity on data stream processing. The solution encompasses elastic executors (operators processes) that optimizes the load balancing and a dynamic scheduler that elastically manages computational resources. Elasticutor was implemented on Storm and tested with benchmarks and real-world applications, where it demonstrated significant performance improvements.

Bartnik et al.[85] addressed aspects related to elasticity and fault tolerance of stateful stream processing applications in Apache Flink. Their solution was a protocol that provides the alternative to adapt the execution at run-time in three aspects: migration of operators, adding new operators, and changing the functions computed by operators. The proposed solution was evaluated with benchmarks running on a cluster, emphasizing the straightforward result that, in distributed stateful stream processing, the adaptation overhead is impacted by the job's state size that has to be migrated.

Kombi et al.[74] presented DABS-Storm for elastic stream processing on Storm by dynamically adapting the parallelism degree. Importantly for generalization purposes, they argued that the proposed solution could be implemented is other related solutions.

Talebi et al.[86] proposed elasticity for CEP running in cluster environments. The proposed solution called ACEP adapts the degree of parallelism is using a cost model for improving the load balance. ACEP was evaluated showing its effectiveness.

5.1.3 | Clouds

Das et al.[87] explored the impact of batch size on the latency of stream processing applications. Based on an understanding of the relation between throughput and latency, they proposed a control algorithm that autonomously adapts the batch size. The experimental validation of the proposed solution evinced that it seamlessly optimizes the latency.

Tudoran et al.[88] tackled inter network transfer overhead by providing an adaptive transfer strategy that self-optimizes the batch sizes. The strategy sets the batch size to the value that considers the instant environment condition, such as transfer rates, aiming to reduce the latency. The proposed solution was validated on a real-world cloud environment showing its effectiveness.

Heinze et al.[89] presented an approach addressing latency aspects on stream processing. There, it was stated that scaling decisions performed for optimizing resources utilization tend to result in latency violations. Consequently, they proposed a solution attempting to minimize latency violation by estimating latency spikes before running scaling decisions. The adaptation regarding the scaling strategy was implemented and compared to related solutions, where their solution evinced fewer latency violations.

A game-theoretic controller was proposed by Mencagli[90] as a control strategy for distributed stream processing. In this model, each fissioned stage is managed with a local controller that sets the internal degree of parallelism, while there are several global controllers. In order to settle globally, two strategies were proposed, a non-cooperative and an incentive-based. In the non-cooperative, each controller only considers its local configuration. On other hand, the incentive-based proposes a controller that senses global conditions for improving the system globally, which tends to improve in terms of performance and efficiency. The theoretical solution was validated with simulations of a mobile cloud computing platform.

An application profiler for stream processing was proposed by Liu et al.[91] Profiling was used to identify potential bottlenecks and applying self-adaptivity in the parallelism configuration to improve the performance. The relation between resources and application performance was handled by the profiler with resource provision at an optimization level, attempting to further improve the performance with the combination of provisioning, scheduling, and placement. The solution was evaluated with real-world applications, where it achieved higher performance in comparison to related solutions.

Adaptivity was considered by Floratou et al.[92] for real-time stream processing analytics, where the notion of self-regulation in Twitter's Heron framework was introduced with the proposed system called Dhalion. In this solution, the user sets a target throughput and Dhalion transparently configures the number of processes and cloud instances for achieving the user goal. Dhalion was validated with real-world applications, showing that the system can dynamically self-regulate to meet SLOs.

Venkataraman et al.[93] observed that stream processing systems need to constantly adapt to failures and workload fluctuations. They proposed Drizzle, a solution for reducing the overhead in case of a recovery adaptation that maintains a QoS. Compared to related solutions, Drizzle achieved lower latencies and significantly faster recovery adaptability.

Tolosana-Calasanz et al.[94] proposed an autonomic controller based on queuing theory for managing resources. The controller manages and controls the number of VMs allocated for the running stream processing applications. The controller also monitors the items queuing time. The approach was evaluated showing its effectiveness.

Mai et al.[95] proposed chi, a control-plane that dynamically adapts stream processing applications. The goal can be seen as facilitating the usage and efficiency of users. With chi, users can simply express SLOs (latency, throughput) while the control plane enforces the goal by using feedback for adjusting the system parameters/configurations. The adaptivity concerns mainly the number of nodes used. The solution was validated in comparison to related solutions showing significant performance gains.

The very relevant problem of downtime on reconfigurations of distributed stream processing was addressed in Rajadurai et al.[9] There, Gloss was proposed as a solution for avoiding downtime on synchronous data flow (SDF) applications. Such an application scenario is arguably a narrow part of stream processing where parallel executions are stable, static, synchronous, and deterministic. Consequently, SDF is a solution only for specific regular applications. However, Gloss can be viewed as a quite elaborated solution. With Gloss, live reconfiguration and optimizations are possible at run-time. In order to avoid downtime, Gloss employs input duplication and concurrent execution of new and old graph topologies during reconfiguration. The solution was validated on real-world cloud environments showing to be effective for avoiding application downtime.

Fardbastani and Sharifi[96] proposed adaptive load balancing for CEP. The solution periodically collects the load of each node and decides if the nodes are balanced or not. Considering the decision based on load, the solution redistributes the tasks. The approach was evaluated emphasizing an increase in terms of performance.

Marangozova-Martin et al.[97] provided a strategy for elastic management of resources on Storm in cloud environments. This approach attempts to reduce latency while consuming minimum resources. The solution is multi-level in the sense that covers application (performance) and environment (number of VMs). The solution was validated in a real-world stream processing application.

Lombardi et al.[98] proposed PASCAL for automatic scaling of distributed applications. PASCAL is combined with a performance model attempting to predict incoming workloads and a system for estimating the number of computing resources to be provisioned. The performance model was evaluated with synthetic and real-world traces that generate the input load (workload). However, it is difficult to estimate how representative and generalizable are the workloads used for other stream processing applications. Importantly for stream processing, different workloads were tested using Storm. The solution was validated in a cloud environment.

Abdelhamid et al.[99] showed Prompt, a solution for the dynamic data partition. In this scenario, the data are partitioned in micro-batches and the dynamism concerns the size of the batches. Noteworthy, adaptive parallelism properties are also covered with an elastic part that dynamically adapts the parallelism degree in case of workload changes. The solution was validated with different applications and workloads showing its effectiveness.

Russo et al.[100] addressed the very relevant problem of heterogeneous computing infrastructures for running stream processing applications. Most of the works considering elasticity for distributed applications assume that all machines and clusters will be homogeneous in such a way that each machine provides the same performance/profitability. However, this assumption is not true anymore in most cases because the environments and machines are becoming more dynamic and irregular. The authors use Markov decision process (MDP) for controlling elasticity and reinforcement learning (RL) algorithms for optimizing the definition of parameters. The solution was validated in cloud environments in terms of performance and cost.

5.1.4 | Accelerators

Schor et al.[101] proposed AdaPNet, an approach aiming at maximizing the performance of streaming applications by adapting the degree of parallelism. In AdaPNet, parallelism is adapted for responding to changes in terms of resources availability. The performance and overhead of the proposed solution were evaluated, where the overhead considers the time taken to change the application at run-time as well as the related memory usage.

Vilches et al.[102] addressed aspects related to efficiently running stream processing applications on heterogeneous architectures composed of CPU and GPUs. They proposed a framework that at run-time finds the best mapping of processing stages on CPU cores, GPUs, or the combination of them. The framework collects runtime statistics for performing the decision-making, where the goal was throughput, energy, or a trade-off between both goals. The solution was validated on different architectures showing performance improvements.

Mencagli et al.[103] proposed Elastic-PPQ, a layered autonomic system for dynamic data stream processing. The layered architecture comprises two adaptation levels. One is a load balancing mechanism for fast variations using the control-theoretic approach. Moreover, a relevant part regarding the parallelism level for slower variations uses fuzzy logic. The solution was evaluated showing effectiveness in terms of adaptability and performance.

De Matteis et al.[104] proposed Gasser, a system for running windowed stream processing applications on hybrid architectures composed by CPU and GPUs, where computations are offloaded to GPUs. Importantly, the proposed solution has an adaptive feedback part for auto-tuning, which tests and tries several configurations for finding the one that achieves the best performance. The configuration considers the different batch sizes and degrees of parallelism. Gasser was validated with data stream processing applications and compared to related solutions. An arguable shortcoming of Gasser is that it only performs adaptation once, which is only suitable for stable and regular workloads.

Stein et al.[105] provided techniques for dynamically adapting the batches size of the specific class of stream processing applications that perform data compression. Moreover, the work targets particularly applications suitable for running on GPUs, where a different decision-making algorithm was proposed. The solution was evaluated considering latency as the target and the suitability of algorithms varies according to the specific workloads.

5.2 | Self-adaptation classification

Table 3 characterizes self-adaptation and parallelism properties of the approaches described in the previous section. It is important to note that here the approaches are separated according to their execution environment and ordered by publication year. Regarding the RQ1 and the categorization

TABLE 3 Self-adaptive properties and tools

Approach	Goal	Adaptation actions/ entities	Theoret. tech.	Realization approach	Tool	Framework	E. E.
Schneider[43]	Throughput	Parallelism degree	/	Adapt. algo.	Algorithm	SPADE-Sys. S	Multicores
Choi[45]	Throughput	Parallelism degree	Delay propag.	Bottleneck detection	Algorithm	/	
Tang[46]	Throughput	Parallelism degree	/	Optimization algo.	Algorithm	System S	
Selva[39]	Throughput	Task mig.	/	Adapt. algo.	Algorithm	StreamIt	
Su[38]	Resource util.	Parallelism degree	Feedback loop	Adapt. algo.	StreamAware	StreamMDE	
De Matteis[37,47]	Energy, latency	Parallelism degree, cores freq.	MPC queuing	Runtime mechanisms	Strategy	FastFlow	
Gad[48]	Throughput	data distr.	Feedback loop	Adapt. algo.	java DSL	/	
Karavadara[49]	Energy	Cores freq.	/	DVFS strategy	Algorithm	S-Net	
Gedik[50]	Latency, throughput	Parallelism degree	/	Bottleneck det.	Algorithm	C-Stream	
Schneider[51]	Throughput	Parallelism degree	/	Adapt. algo.	Scheduler	SPL	
De Sensi[13]	Efficiency	Parallelism degree, cores freq.	/	Linear regression	Nornir	FastFlow	
Vogel[54]	Latency	Parallelism degree	Feedback loop	Adapt. algo.	Algorithm	SPar	
Griebler[16]	Performance, energy	Parallelism degree, cores freq.	Feedback loop	Linear Regression/Adapt. algo.	SPar and Nornir	FastFlow	
Kahveci[56]	Throughput	Parallelism degree	/	Adapt. algo.	Algorithm	Joker	
Vogel[14]	Throughput, efficiency	Parallelism degree	Feedback loop	Adapt. algo.	Algorithm	SPar	
Vogel[42]	Throughput	Parallelism degree	Feedback loop	Adapt. algo.	Algorithm	SPar	
Gulisano[57]	Throughput	Tasks distr., nodes	/	Load thresholds	StreamCloud	Borealis	Clusters
Balkesen[58]	Latency	Tasks distr., nodes	/	Adapt. algo.	Algorithm	Borealis	
Heinze[59]	Resource util., latency	Task mig.	/	Threshold, R.L.	Algorithm	FUGU	
Gedik[36]	Throughput	Parallelism degree	/	Control algo.	algorithm	SPL IBM S.	
Wu[60]	Throughput	Task distr.	/	Adapt. algo.	DoDo	S4	
Chatzistergiou[61]	Throughput	Tasks distr.	/	Group-aware	Algorithm	Storm	
Martin[62]	Throughput	Nodes, tasks mig.	/	Threshold algo.	Algorithm	StreamMine3G	
Zacheilas[63]	Resource util., latency	Parallelism degree	/	Short path algo.	Esper	Storm	
Lohrmann[64]	Latency	Parallelism degree	Queuing th.	Scaling policy	Algorithm	Nephele	
Mayer[65]	Limit buffering	Parallelism degree	Queuing th.	Adapt. algo.	Algorithm	/	
Mayer[66]	Limit buffering	Parallelism degree	Queuing th.	Adapt. algo.	Algorithm	/	
Heinze[67]	Reduce cost	Nodes	/	Threshold algo.	Algorithm	FUGU	
Martin[68]	Fault tolerance	Tolerance scheme	Controller	Checkpoint algo.	Algorithm	StreamMine3G	
Zhang[69]	Latency	Batch and block size	/	Isotonic Regr.	DyBBS	Spark	
Liu[70]	Latency	Tasks distr.	/	Adapt. algo.	Algorithm	Storm	
Gil[71]	Throughput	Nodes, tasks distr.	/	Adapt. algo	Algorithm	S4	
Li[72]	Latency, throughput	Nodes	/	Adapt. algo.	Algorithm	Storm	
Kombi[73]	Latency	Parallelism degree	/	Time service	Autoscale	Storm	
De Matteis[75]	Latency	Parallelism degree	Control theory	Adapt. algo.	Algorithm	FastFlow	

(Continues)

TABLE 3 (Continued)

Approach	Goal	Adaptation actions/ entities	Theoret. tech.	Realization approach	Tool	Framework	E. E.
Cardellini[78]	Latency	Parallelism degree, tasks mig.	Feedback loop	Reinf. learn.	Algorithm	Storm	
Cardellini[76]	Response time	Task placem., parallelism degree	/	Additive Weight.	ODRP	Storm	
Cheng[80]	Throughput	Job parallelism level	/	Reinf. learning	A-scheduler	Spark Stream.	
Cheng[81]	Latency, throughput	Batch size, job parallelism level	Feedback loop	Fuzzy, reinf. learn.	A-scheduler	Spark Stream.	
Lombardi[82]	Throughput	Parallelism degree, nodes	/	Adapt. algo.	Elysium	Storm	
Kalavri[83]	Throughput	Parallelism degree	/	Controller	DS2	Flink, Timely	
Wang[84]	Throughput, latency	CPUs, tasks mig.	/	Adapt. algo.	Elasticutor	Storm	
Bartnik[85]	Latency	Parallelism degree	/	Adapt. protocol	Protocol	Flink	
Kombi[74]	Latency	Parallelism degree, Tasks distr.	/	Adapt. algo.	DABS	Storm	
Talebi[86]	Latency	Parallelism degree	MPC queuing	Adapt. algo.	Algorithm	/	
Das[87]	Latency	Batch size	Feedback loop	Control algo.	Algorithm	Spark Stream.	Clouds
Tudoran[88]	Latency	Batch size	/	Simple testing	Algorithm	JetStream	
Heinze[89]	Latency	Task mig.	Cost model	Bin packing	Algorithm	FUGU	
Mencagli[90]	Efficiency	Parallelism degree	Game th.	Adapt. algo.	Algorithm	/	
Liu[91]	Latency, throughput	Tasks distr., data speed	Feedback loop	Trial-and-error algo.	Algorithm	Storm	
Floratou[92]	Throughput	Operator instances	/	Adapt. algo.	Dhalion	Heron	
Venkataraman[93]	Latency, throughput	Batch size, nodes	/	Adapt. algo	Drizzle	Spark	
Tolosana[94]	Minimize Queues size	Nodes	/	Adapt. algo.	Algorithm	CometCloud	
Mai[95]	Throughput, latency	Nodes, batch size	Feedback loop	Adapt. algo.	Chi	Flare, Orleans	
Rajadurai[9]	Avoid downtime	Data distr., grain, nodes	/	Adapt. algo.	Algorithm	StreamJIT	
Fardbastani[96]	Load balance	data and tasks transfer	/	Adapt. algo.	Algorithm	CCEP	
Marangozova[97]	Resource util., latency	Nodes	/	Adapt. algo.	Algorithm	Storm	
Lombardi[98]	Resource util.	Nodes	/	Neural network	PASCAL	Storm	
Abdelhamid[99]	Throughput, latency	Parallelism degree, batch size	/	Adapt. algo.	Prompt	Apache Spark	
Russo[100]	Reduce cost	Parallelism degree	/	Markov, reinf. learn.	/	/	
Schor[101]	Throughput	Parallelism degree	/	Adapt. algo.	AdaPNet	POSIX	Accelerators
Vilches[102]	Throughput, energy	Stages Mapping	Queuing th.	Adapt. algo.	Algorithm	Intel TBB	
Mencagli[103]	Throughput, efficiency	Parallelism degree	Feedback loop	Fuzzy logic	Elastic-PPQ	FastFlow	
De Matteis[104]	Throughput, latency	Parallelism degree, batch	/	Adapt. algo.	Gasser	FastFlow	
Stein[105]	Latency	Batch size	Queuing th.	Adapt. algo.	Algorithms	SPar	

Abbreviations: adapt., adaptation; algo., algorithm; distr., distribution; E. E., Execution environment; freq., frequency; mig., migration; MPC, model predictive control; ODRP, optimal DSP replication and placement; placem., placement; propag., propagation; regr., regression; reinf. learn., reinforcement learning; sys., system; theoret. tech., theoretical technique; th., theory; util., utilization.

proposed in Section 4.1.1, in column "Goal" of Table 3 it is possible to note that there are several purposes of applying self-adaptiveness to stream processing. Thus, we propose the following catalog and definitions to organize the self-adaptation goals:

- *Throughput* (*maximize*) is the number of stream items/tasks processed in a given time interval. A high throughput tends to be a goal. Additionally, there are some studies that intend to allow the user to define throughput as a performance goal, which we refer to as *target throughput*.
- *Latency* (*minimize*): Latency in stream processing is a performance metric that refers to the time taken to process stream items/tasks. A lower latency tends to be better, which can be set as a *constraint* in some approaches.
- *Resources usage* (*maximize, limit, optimize*): In the context of this study, we refer to resources as computational power available for processing computations. Other terms like *resource utilization*, *system utilization* (abbreviated *sys. util.*) are considered synonymous and used interchangeably in this study. Some works attempt to utilize resources as much as possible. On the other hand, some approaches try to limit how much resources a given application uses in order to avoid interference in multi-tenant environments or for limiting energy consumption. *Energy* consumption is also a goal that we categorize as related to resources. There are also goals for optimizing (abbreviated *opt.*) the usage of resources in such a way that a performance goal is met with minimum resources, which is also related to limiting resource usage. This aforementioned resource optimization characterizes the goal of computing *efficiency*, which is also pursued by some approaches.
- *Buffering* (*limit, minimize)*: We refer to buffering as queuing stream items before processing them. The term *queue size* tends to mean a similar aspect in the stream processing applications. Limiting the buffer sizes impacts the performance of stream processing mainly in terms of latency, which is a potential optimization.
- *Cost* (*reduce*): On *pay-per-use* paradigms like cloud computing, the resources usage impacts directly the cost. Consequently, there are efforts that aim to reduce the cost of stream processing by self-adapting executions considering resource usage.
- *Fault* (*tolerance*): This aspect concerns the running stream processing applications, where avoiding faults is relevant. There are some approaches that autonomously adapt to the fault tolerance scheme for optimizing executions. Moreover, *avoid downtime* when running a stream processing application is a facet of fault tolerance considered by some approaches.
- *Load* (*balance*): Balancing the load of stream processing is an objective pursued for optimizing executions, which can achieve gains in terms of performance or resources. Some approaches explicitly mention load balance as a goal, where it is assumed that optimizing the load balance will provide gains to applications.

Throughput improvement of stream processing applications was the goal of 31 studies. Latency reduction was mentioned as a goal in 29 studies. Noteworthy, some of these studies attempt to reach a combination of optimal throughput and latency. In fact, a significant part of approaches has more than a single objective, often targeting a trade-off between different metrics.

Considering the motivation for a taxonomy discussed in Section 4, in this survey, it was possible to extend the revision for covering unique approaches focusing on additional applicabilities of self-adaptiveness. For instance, studies with new goals were found, such as minimize energy consumption,[13,16,49,53,102] avoid downtime,[9] and tolerate faults.[68] We believe that this is relevant to enable self-adaptiveness to be evaluated in the future for supporting new applicabilities.

5.3 | Adaptation actions and entities managed

Considering the classification of the scope of adaptation proposed in Section 4.1.1, the "adaptation-scope" section of Figure 4 shows the percentage of approaches that adapted a given scope. These results represent the sum of the percentages exceeds 100% because some approaches adapted more than one item, for instance, adapting the parallelism and the underlying resources. Noteworthy, the majority of approaches are applying adaptations at the application level, specifically the processing and data aspects. In some cases adapting the resources utilized may not be so important because it can be adapted transparently by the abstraction, like in a cloud environment where the application is adapted to reduce the costs and the resources are adapted by the lower layer of the provider.

Regarding the entities controlled in self-adaptive solutions (RQ2), in Table 3 it is possible to note a high number of different entities. In the environment, it is possible to manage nodes, CPUs, cores, frequency, and cores mapping. Moreover, in the application, it is possible to manage the placement, scheduling, parallelism degree, batches size, data distribution, and so forth. Thus, we propose the following catalog and description of entities self-adapted:

- *Parallelism degree* (a.k.a. degree of parallelism) is a generalization for adaptations at the system/application level related to the number of active processing elements. The parallelism degree is also referred to as the number of threads/processes and the number of replicas. The number of threads/processes is the degree of parallelism in applications running on multicore or cluster machines. Moreover, the number of replicas is the

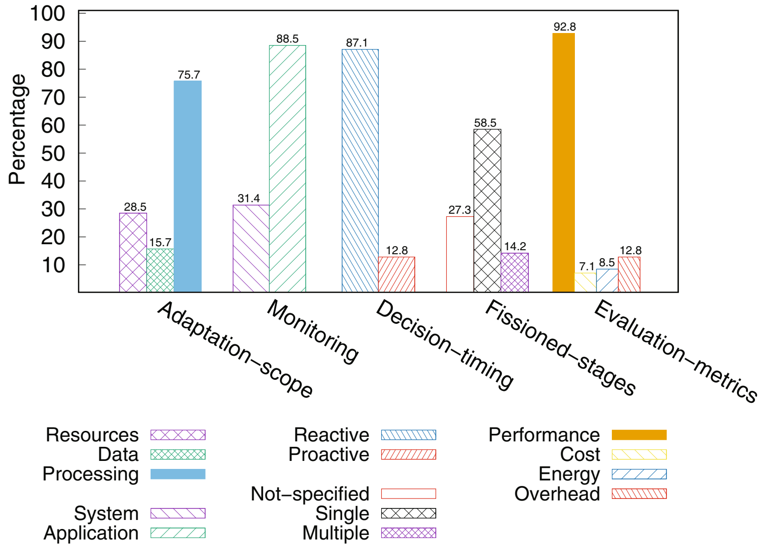

FIGURE 4 Results overview

number of entities processing in a given fissioned parallel stage, where each replica sometimes is related to one thread at the OS level. For the sake of precision, we have a subcategory inside the parallelism degree called *job parallelism level*, which is specific to some scenarios where parallelism is achieved by running multiple simultaneous jobs. The number of concurrent jobs can be self-adapted characterizing an autonomous *job parallelism level*. Other similar terms used in specific contexts for referring to the parallelism degree are *operator instances* and *operator parallelism*.

- *Data* are a generalization for adaptation concerning the data items. Adaptation of *batch and block size* concerns the changes performed at run-time in the data granularity. *Data speed* attempts to control the speed that data are ingested, such as the arrival time. *Data migration* corresponds to changing the place where data items are executed. *Data distribution* relates to changing the way that data items are assigned to computing elements. In data stream processing, a data item tends to be treated as a task. Consequently, *task distribution* is a generalization that we used when the distribution of task changes but it is not defined if each task corresponds to a data item.
- *Cores*: This category encompasses changes at the computing resources level, particularly in the CPUs. The *number of cores* corresponds to the number of active cores that can be changed in modern processors. Additionally, the *cores frequency* (abbreviated freq.) can be modified by setting fixed clock rates. *Mapping* refers to policies for mapping software threads to CPUs.
- *Nodes* is a resource adaptation that refers to the number of physical computing nodes where applications are executed. Usual adaptation actions are adding or releasing nodes.
- *Tolerance scheme* concerns the technique(s) used for assuring fault tolerance mechanisms in stream processing tools. A tolerance scheme has been implemented in distributed processing scenarios.

Different entities can be managed for pursuing the same metric (throughput, latency). However, the difference lays in how effective each entity is for optimizing the executions. The entities used are also highly related to the approaches' runtime, specific solutions may or may not support adapting a given entity at run-time. For instance, several approaches adapt the parallelism degree at run-time[36,50,54,90] while others have to change the graph topology to adapt to the parallelism degree.[9]

In Reference 9, the graph topology is transformed at run-time, but it remains unclear if the graph was transformed because of their runtime constraints or if their targeted advanced optimizations (e.g., efficiency). A runtime constraint can prevent adaptation at run-time. Thus, in Reference 9, the graph topology is transformed because this could be the only way to adapt the executions at run-time. We identify graph topology transformations as a complex adaptation action that has the potential to further optimize stream processing applications in terms of performance and efficiency.

5.4 | Monitoring on self-adaptation

Information about the executions is necessary to perform adaptation actions, where RQ3 examines which data is used according to the category described in Section 4.1.2. It is possible to note from the section "Monitoring" of Figure 4 that the majority of approaches are monitoring information from the application level, but there is also a reasonable number of studies that monitor low-level system indicators. Noteworthy, the section

"Monitoring" of Figure 4 shows the percentage of approaches that collected information from the applications and/or from the system, where the sum of the percentages exceeds 100% because some approaches monitored the application and the system.

The monitoring part is also related to the tools/technologies used, as in some cases, it is only possible to monitor the system, or only monitoring the application when system indicators are unreachable. It is also important to note that monitoring actions are performed to some extension in all approaches, adapting the execution without collecting information can be unfeasible. However, in the literature, we lack in-depth discussions of the limitations and advantages of specific monitoring statistics and the potential overhead that the monitoring routines can cause.

5.5 | Adaptation decisions

Regarding how the decision-making for adaptation is performed (RQ4) described in the proposed classification in Section 4.1.3, the section "Decision-timing" of Figure 4 shows that the timing of the majority of studies is reactive. The reason behind this it is challenging to predict the load of stream processing applications. Consequently, reactive approaches may eventually violate QoS, but they tend to respond better to fluctuating workloads. References[37,47,63,65,66,74,75,82,98] claimed to be proactive.

Another aspect from RQ5 is related to the *theoretical technique* used for decision-making, some utilized are feedback loops from control theory. However, the majority of works in the fourth column of Table 3 are filled with a slash "/" meaning that they do not mention the theoretical technique used when designing self-adaptiveness. The use of a slash is one to represent the lack of information in surveys.[106] In the categorization of *realization approaches*, we catalog algorithms designed for performing the decision-making with different goals, design goals, and entities controlled. These algorithms that are designed in specific solutions are called here *adaptation algorithm*, where there are some approaches that it is not possible to classify how the realization is performed. Yet regarding the decision-making, there are also more complex ones like heuristics and reinforcement learning, however, a low complexity is expected for a given approach to be computationally feasible on highly dynamic stream processing scenarios. For instance, in Reference 78, the authors concluded that some learning algorithms require a long time to find an optimal decision policy. Some approaches use trial-and-error for finding a configuration with the best performance. However, it tends to be less efficient as several suboptimal or poor configurations lead to performance losses. Additionally, under eventual temporal changes, several trials are required again.

5.6 | Self-adaptive parallelism in stream processing

Parallelism is commonly used for improving the performance of stream processing applications.[2,4] RQ5 concerns parallelism aspects of self-adaptive approaches proposed in the classification in Section 4.2. In the last column of Table 3, we present the parallel library or framework used, 13 works used Storm for providing self-adaptive properties. It is also notable that in some studies it remains unclear in which tool they designed and implemented the self-adaptive entities, the last column of Table 3 is filled with a slash "/." In such cases, it is assumed that self-adaptiveness was implemented on prototypes, which are not necessarily integrated with any existing runtime library/framework.

The number of fissioned stages (described in Section 4.2.3) is another relevant aspect related to RQ5. It is possible to note in the section "Fissioned-stages" of Figure 4 that the majority of studies focus on managing application graph topologies with a single fissioned stage. It is also notable that a part of the approaches does not describe well enough the characteristics of the applications used, for the sake of precision these approaches are classified as "Not specified." Importantly, there are also approaches for self-adaptiveness in applications with multiple fissioned stages.[36,43,45,46,64,82,90] However, the application having a multiple stage topology does not mean that such a robust graph is adapted at run-time. The multiple stage topology can be static while other aspects are adapted, such as the batch size[64] or tasks distribution. In this sense, multiple fissioned stages are utilized on specific scenarios, where the mechanism and the strategies' decision-making are limited in terms of generalization.

In this survey we are interested in specific aspects of parallelism exploitation, focusing particularly on self-adaptive parallelism abstractions. Additional parallelism details can be found in References 2 and 4. RQ6 and the category described in Section 4.2.5 relate to whether the existing approaches focus on providing parallelism abstractions to application programmers. Although nowadays we have frameworks providing high-level programming abstractions for stream processing,[4] a limited number of approaches[13,16,43,53,54,68,74,90,95] mentioned abstractions as relevant for using/implementing self-adaptation. Considering that it tends to be very complex and time-consuming for application programmers to achieve self-adaptation in their domain-specific applications, we believe that the tools/frameworks should come with ready-to-use abstractions. Examples are offering flags and parameters to enable users to provide hints on their objectives at a higher level. Such objectives could be met by intelligent and autonomous systems that seamlessly use entities to self-adapt executions.

Providing additional parallelism abstractions is one aspect that certainly will require more effort in the future. We expect that the specific characteristics of stream processing would need to be considered for assessing the feasibility of self-adaptive abstractions. Moreover, evaluating if the existing approaches are suitable for parallelism abstractions requires in-depth analysis, and new evaluation methodologies would need to be proposed.

5.7 | Validation metrics and variations

An often neglected aspect in literature approaches is the comprehensive validation, which is a concern covered by RQ7 and RQ8 and organized in taxonomy in Section 4.3. Importantly, the section "Evaluation-metrics" of Figure 4 evinces the percentage of approaches that considered a given metric in the evaluation. Hence, the sum of the percentage exceeds 100% because some approaches considered more than one metric, for example, covering the performance and the cost. In Figure 4, it is notable that performance is the most evaluated aspect of self-adaptive solutions. The approaches[63,67,68,86,100] considered in the evaluation of the relevance of the cost and the approaches[13,16,49,80,102] evaluated energy consumption.

Concerning the variations (RQ8) considered for evaluating the solutions, Table 4 evinces the results from the proposed taxonomy. A number of works were only tested with more than one application, but it is mostly unclear how different are the processing characteristics of those applications. Only a few studies considered the processing pattern of applications in their validation. The solutions mostly neglect the variability of resources available and environments. Moreover, several studies have no variations, meaning that the solution was validated with a single application, one workload, running in one environment. Evaluate a solution with different applications is relevant, mostly because each application has specific characteristics (processing behaviors, memory access, I/O, communication, etc.). Moreover, different workloads are relevant for evaluating the self-adaptiveness of algorithms because if the testbed has only one behavior, it is hard to estimate how the decision-making algorithms will behave under other conditions.

5.8 | Overhead measurement

Regarding RQ9 that related to the overhead category described in Section 4.3.3, the section "Evaluation-metrics" of Figure 4 provides results concerning the overhead measurement in the literature. The measurement of the overhead requires a comprehensive validation scenario, only the references[14,37,39,42,47,58,68,82,101] considered the overhead that can be caused by performing adaptation actions at run-time. Relevant aspects could be the resource utilization and performance of the application. The monitoring, self-adaptation algorithms, and reconfiguring entities for pursuing goals can also cause overhead. We argue that self-adaptive solutions should further consider the potential overhead caused, we believe that new validation methodologies for assessing the overhead should be proposed.

Overhead is also highly related to the environment and programming models used. For instance, it tends to be more complex to apply changes in distributed stream processing and when the adaptation requires state migration.[36,85] A relevant example of overheads is an adaptation to Storm's topologies, where changing the number of replicas causes application downtime.[79] In Reference 78, the reconfiguration applied in Storm caused performance losses, thus, they neglected such overhead by excluding the performance results up to 2 min after each reconfiguration. Consequently, for 2 min, there are no QoS guarantees. Thus, we argue that there is a need for better mechanisms and decision algorithms for improving stream processing applications for real-world scenarios.

5.9 | Results summary

Table 5 provides an overview of the research questions and the results aforementioned throughout this section.

6 | RESEARCH CHALLENGES

In Section 5, we reviewed and discussed the current approaches from the literature. In this section, we introduce and discuss important aspects to be enhanced in the future, such aspects are considered as open research challenges.

6.1 | Supporting self-adaptive parallelism in robust compositions

Considering that in this work we are focused on self-adaptation for parallel executions, it is relevant to cover parallelism adaptation in robust compositions (defined in Section 4.2.3). The section "Fissioned-stages" of Figure 4 introduced how many fissioned stages are used in the related literature. Adapting parallelism aspects at run-time in robust compositions is not a trivial problem. For instance, a simple strategy that only adapts the parallelism degree would need to take actions considering at least safety, load balancing between stages/compositions, and the amount of resources available. Consequently, such a solution would require several sensors and actuators with or without coordination between them.

Furthermore, the approaches available in the literature are still not presenting a self-adaptive solution that is generic and comprehensively validated. For instance, the IBM stream tools are tuned to their runtime library and constraints. Such a claim is supported by considering the high

TABLE 4 Self-adaptive validation

Approach	Application	Processing pattern	Input characteristics	Resources available	Execution environment
Schneider[43]	X	X			
Choi45, [45]	X			X	
Tang46, [46]	X				
Selva[39]	X				
Su38, [38]	X		X		
De Matteis37, [37,47]				X	
Gad[48]					
Karavadara[49]	X				
Gedik50, [50]	X	X			
Schneider51, [51]	X				X
De Sensi13, [13]	X	X			
Vogel54, [54]					
Griebler[16]	X				
Kahveci56, [56]	X				
Vogel14, [14]	X	X	X		
Vogel[42]	X				
Gulisano[57]					
Balkesen[58]		X			
Heinze59, [59]			X		
Gedik36, [36]	X				
Wu60, [60]					
Chatzistergiou[61]	X		X		
Martin[62]					
Zacheilas[63]					
Lohrmann[64]	X				
Mayer65, [65]	X				
Mayer[66]	X				
Heinze[67]	X	X			
Martin68, [68]	X				
Zhang69, [69]			X		
Liu[70]	X				
Gil[71]					
Li[72]			X		
Kombi[73]	X		X		
De Matteis[75]			X		
Cardellini[78]			X		
Cardellini[76]					
Cheng[80]					
Cheng[81]					

(Continues)

TABLE 4 (Continued)

Approach	Application	Processing pattern	Input characteristics	Resources available	Execution environment
Lombardi[82]	X		X		
Kalavri[83]	X	X	X		
Wang[84]	X		X		
Bartnik[85]	X				
Kombi et al.[74]	X		X		
Talebi[86]			X		
Das[87]		X		X	
Tudoran[88]			X		
Heinze[89]		X			
Mencagli[90]			X		
Liu[91]	X				
Floratou[92]			X		
Venkataraman[93]					
Tolosana[94]		X			
Mai[95]			X		
Rajadurai[9]	X		X		
Fardbastani[96]			X		
Marangozova[97]			X		
Lombardi[98]	X		X		
Abdelhamid[99]	X		X		
Russo[100]			X		
Schor[101]	X			X	
Vilches[102]	X	X			X
Mencagli[103]	X	X			
De Matteis[104]	X				
Stein[105]			X		

number of parameters that have to be set for using these tools and by the comparison between different approaches provided by De Matteis and Mencagli.[37] In De Matteis and Mencagli,[37] the strategy from Reference 36 was reproduced for comparison, where it achieved a poor performance by being unable to adapt under unbalanced workloads. The implication of this result provided in Reference 37 is that the strategy of Reference 36, which is a well-known solution for robust application compositions, lacks in terms of generality by performing poorly even with a simplistic composition of only one replicated stage. Kalavri et al.[83] found a similar outcome where Dhalion[92] was replicated and showed poor performance and slow convergence. Consequently, we argue that the scenario of robust application compositions demands a comprehensive evaluation of the literature's decision algorithms.

6.2 | Improving resources efficiency and performance with optimizations at run-time

As far as the entities found in the SLR are concerned, a potential optimization is to perform dynamic adaptations in the applications graph topologies (e.g., compositions), which can potentially self-optimize the application runtime in such a way that additional efficiency and flexibility is achieved. In this vein, only the work of Reference 9 was found that addresses graph adaptation at run-time, where the application is recompiled and changed without downtime using input duplication. However, this solution only performs such a complex optimization because runtime constraints make it unfeasible to adapt the parallelism degree at run-time, requiring program recompilation. However, dynamically changing the parallelism degree is possible in other runtimes without the need for recompilation.

TABLE 5 Summary of research questions and literature results

Question	Context	Overview
RQ1	Goals	Achieve goals such as latency and throughput are mostly pursued.
RQ2	Entities being self-adapted	Adaptation can be in the environment (e.g., nodes, cores, frequency) and at the application level (parallelism degree, placement, scheduling, batches).
RQ3	Information used	Most approaches are collecting information from the applications to utilize for decision-making.
RQ4	Decision-making timing	Most actions are reactive. Although feedback loops are widely used as a theory for designing self-adaptation, several approaches do not explain the theory used. There are a large number of adaptation algorithms for decision-making.
RQ5	Parallelism characteristics	Several frameworks are being used. Storm is still popular in stream processing. Most studies focus on applications with a single fissioned stage.
RQ6	Parallelism abstractions	A few approaches mentioned abstractions as relevant for using/implementing self-adaptation, which is a potential aspect to be considered in future efforts.
RQ7	Experiments	Performance is mostly considered as an evaluation metric.
RQ8	Variations in experiments	Most studies were tested with different applications. The variations in the validation of solutions certainly require more attention in the future.
RQ9	Measuring the overhead	Few studies considered the overhead caused by adaptation actions, which is a concern that arguably has to be included in new evaluation methodologies.

Considering that stream processing applications execute for long periods with fluctuations, we argue that further enhancements are needed for providing the flexibility for changing the applications graph's topology. Optimization in these aspects can potentially improve the performance (stages separation), reduce resource consumption (fusion), or achieving a trade-off between performance and resources. Also, we believe that a mechanism for adapting graph topologies combined with self-adaptive strategies could make it possible to detect and overcome bottleneck stages at run-time. The importance of the aforementioned aspects can be also seen for optimizing resource usage when running stream processing applications in modern environments (e.g., fog, edge). In such environments, the availability of computational resources tends to be more restricted than in highly used multicore machines.

6.3 | Improving parallel stream processing for running in dynamic environments

Our SLR also covered the important aspect related to the environment and architecture that stream processing applications are being run. Notable, considering the evolution of the architectures, there are already efforts using hardware accelerators (GPUs, FPGAs) for stream processing.[101-103] Additionally, distributed cluster environments are highly used. Multicores also have the potential for providing the performance level required by a significant part of stream processing applications. This is achievable considering the increasing number of cores and sockets available in a single machine.

In the SLR, it is notable that cloud environments are increasing in use for stream processing applications. Cloud computing can be seen as a flexible and dynamic execution environment, which can also be seen as a starting point for other environments like Fog and Edge. However, there are still challenges. For instance, Rajadurai et al.[9] demonstrated a peculiar issue of stream processing applications running in cloud environments. Live migration, which is a technique for moving a virtual machine from one physical node to another, causes downtime in stream processing applications. Downtime occurs due to the characteristics of these applications of constantly inserting new data that modifies the data saved in memory. A representation of this problem is provided by Rajadurai et al.[9] where the application throughput drops to 0 for several seconds. In the narrow scenario of Reference 9, they proposed techniques for mitigating downtime. However, a relevant aspect is that the potential downtime of stream processing applications running in cloud environments is not mentioned by other works targeting this kind of environment.

6.4 | Self-adaptiveness validation and overhead measurement

Table 4 highlights the evaluation aspects of approaches from the literature. With a critical view, it is possible to note that the validation of the proposed solutions is not receiving the necessary attention. Although there are available efforts for benchmarking the adaptiveness of stream processing systems,[107] we argue that we need further improvements in terms of methodologies and representative benchmarks for stream processing characteristics.[108] The majority of approaches consider only the performance, neglecting the potential overhead caused. Additionally, only a few works considered a comprehensive testbed, with different application processing characteristics, inputs, and running architectures and environments. In fact, we argue that no work yet considered a wide combination of these evaluation categories in order to characterize how the proposed solution would behave in different scenarios.

Although every work tends to have specific scenarios and goals, we believe that the impact of self-adaptiveness may cause in applications must be better measured. The validation must encompass different scenarios being as broad as possible. Proposing new evaluation/validation methodologies and guidelines is a potential opportunity to improve the validation of self-adaptive solutions, where it can potentially improve the QoS of applications using adaptiveness. These new approaches are expected to be representative for measuring self-adaptation performance, energy, resource utilization metrics, and overhead.

6.5 | Generalization and reproducibility of self-adaptive solutions

The conducted SLR has covered and extracted relevant information regarding the decision-making of existing self-adaptive solutions. Noteworthy, several approaches used as theoretical techniques the feedback loops adapted from control theory.[11] Another significant number of works also used queuing theory for modeling their solutions. The theories used have similarities, and one complements another (discussed in Section 2.3), but, notably, each approach tends to adapt suitable aspects to its specific scenario. Hence, there is a lack of considering the potential of generalizing and make the proposed solutions reproducible.

The proposed solutions could enable reproducibility by decoupling specific technical and low-level mechanisms from the potentially generalizable decision-making strategies. In this vein, the design of new solutions can be improved by modeling what can be generalizable, which would enable new solutions to reuse existing parts and only implementing specific runtime mechanisms.

Additionally, comprehensive evaluation methodologies would allow one to validate the literature's strategies to a broader context or different scenarios to determine whether new strategies or decision algorithms must be proposed. Significantly, new tools and languages being designed could include mechanisms to adapt parallel mechanisms, especially for applications with multiple fissioned stages. Hence, new tools could support features based on generically designed and comprehensively validated strategies for providing self-adaptation. For instance, Reference 13 proposes a framework for developing self-adaptive solutions. We argue that there are further opportunities for providing frameworks for modeling self-adaptive solutions on runtime libraries/tools.

7 | THREATS TO VALIDITY

There are also potential threats to the validity of the SLR to be considered. Similar to related SLR,[32] the search for studies can be considered limited. Finding all relevant studies is a known challenge in literature revisions. We attempt to mitigate this aspect by searching in different databases and libraries as well as performing multiple search rounds.

The search terms used can also limit the results, as generic terms could bring much more irrelevant results. However, the relevance of evaluating studies from other contexts is arguably low. Our search terms were validated considering their completeness with pilot searches evaluating whether previously known relevant studies were found.

Filtering studies and extracting data is another potential threat because different terms can be used for referring to a single characteristic. Consequently, we considered the general terminology of the area, where it is worth mentioning the catalog of similar terms provided by Hirzel et al.[1] The threats to the consistency of data extraction were potentially minimized with a multi-step revision and an explicit revision protocol. Moreover, some short abstracts were excluded as well as works that fail to provide detailed information about their context and proposed solution. Other works without technical aspects and experimental evaluation were also removed as they fail to provide a suitable validation.

8 | CONCLUSIONS

In this article, we presented a literature review of self-adaptation approaches applied to parallel stream processing. We also proposed a categorization of relevant aspects to comprehend the studies and their proposed solutions. These categorizations and evaluations of self-adaptiveness are aimed to contribute to researchers and practitioners in the adoption of self-adaptive. Consequently, self-adaptiveness can be a part of improving technical solutions and being a topic for future research. Although this article's scope only concerns stream processing due to its unique characteristics, we expect our results to be generalized and applied in other scenarios. For instance, self-adaptive parallelism can be exploited in applications where online, dynamic, and continuous adaptation is not needed. However, the strategies for monitoring and decision-making are still essential. Consequently, provide flexibility and modularity for future solutions is of paramount importance for their generalization and increase usage.

Considering the literature results, it is possible to observe that the research area is expanding, but several research challenges are still existing. Noteworthy, the provisioning of advanced self-adaptive approaches could provide enhancements to applications with realistic robust compositions, where we emphasize the need for providing abstractions for parallelism and execution aspects. Moreover, the validation and generalization of available solutions are limited, where we argue that comprehensive validation scenarios are very relevant for the acceptance and usage of self-adaptiveness.

It is also worth mentioning the demand for self-adaptive solutions supporting stream processing applications that will potentially run in dynamic environments (e.g., cloud, fog, edge). To be a key part of providing solutions for future computational challenges, the self-adaptive approaches are expected to be highly efficient in terms of performance or resources. In the future, we intend to contribute to the research area by tackling part of the existing open research challenges.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nivel Superior—Brasil (CAPES)—Finance Code 001, FAPERGS 05/2019-PQG project PARAS (No. 19/2551-0001895-9), FAPERGS 10/2020-ARD project SPAR4.0 (No. 21/2551-0000725-7), Universal MCTIC/CNPq No. 28/2018 project SPARCLOUD (No. 437693/2018-0). Open Access Funding provided by Universita degli Studi di Pisa within the CRUI-CARE Agreement.

DATA AVAILABILITY STATEMENT

The data analyzed in the current study are available from the corresponding author on reasonable request.

ENDNOTES

*https://www.scopus.com

†https://webofknowledge.com

‡https://dl.acm.org

§https://ieeexplore.ieee.org

ORCID

Adriano Vogel https://orcid.org/0000-0003-3299-2641

Dalvan Griebler https://orcid.org/0000-0002-4690-3964

Marco Danelutto https://orcid.org/0000-0002-7433-376X

Luiz Gustavo Fernandes https://orcid.org/0000-0002-7506-3685

REFERENCES

1. Hirzel M, Soulé R, Schneider S, Gedik B, Grimm R. A catalog of stream processing optimizations. *ACM Comput Surv*. 2014;46(4):46.
2. Röger H, Mayer R. A comprehensive survey on parallelization and elasticity in stream processing. *ACM Comput Surv*. 2019;52(2):36.
3. Andrade H, Gedik B, Turaga DS. *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge University Press; 2014.
4. de Assuncao MD, da Silva VA, Buyya R. Distributed data stream processing and edge computing: a survey on resource elasticity and future directions. *J Netw Comput Appl*. 2018;103:1-17.
5. Danelutto M, Mencagli G, Torquati M, González-Vélez H, Kilpatrick P. Algorithmic skeletons and parallel design patterns in mainstream parallel programming. *Int J Parallel Program*. 2021;49(2):177-198.
6. Mattson TG, Sanders BA, Massingill BL. *Patterns for Parallel Programming*. Addison-Wesley; 2005.
7. McCool M, Robison A, Reinders J. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier; 2012.
8. Vogel A, Mencagli G, Griebler D, Danelutto M, Fernandes LG. Towards on-the-fly self-adaptation of stream parallel patterns. 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP) Valladolid, Spain; 2021:89-93. IEEE.
9. Rajadurai S, Bosboom J, Wong WF, Amarasinghe S. Gloss: seamless live reconfiguration and reoptimization of stream programs. *ACM SIGPLAN Not*. 2018;53(2):98-112.
10. Shevtsov S, Berekmeri M, Weyns D, Maggio M. Control-theoretical software adaptation: a systematic literature review. *IEEE T Software Eng*. 2017;44(8):784-810.
11. Hellerstein J, Diao Y, Parekh S, Tilbury D. *Feedback Control of Computing Systems*. Wiley; 2004.
12. Krupitzer C, Roth FM, VanSyckel S, Schiele G, Becker C. A survey on engineering approaches for self-adaptive systems. *Pervasive Mob Comput*. 2015;17:184-206.
13. De Sensi D, De Matteis T, Danelutto M. Simplifying self-adaptive and power-aware computing with Nornir. *Future Gener Comput Syst*. 2018;87:136-151.
14. Vogel A, Griebler D, Danelutto M, Fernandes LG. Minimizing self-adaptation overhead in parallel stream processing for multi-cores. In: Schwardmann U, Boehme C, Heras DB, et al., eds. *Euro-Par 2019: Parallel Processing Workshops. Euro-Par 2019*. Lecture Notes in Computer Science. Vol 11997. Springer; 2020:30-41.
15. Vogel A, Griebler D, Fernandes LG. Providing high-level self-adaptive abstractions for stream parallelism on multicores. *Softw Pract Exp*. 2021;51(6):1194-1217.
16. Griebler D, Vogel A, De Sensi D, Danelutto M, Fernandes LG. Simplifying and implementing service level objectives for stream parallelism. *J Supercomput*. 2020;76(6):4603-4628.
17. Cole M. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Comput*. 2004;30(3):389-406.

18. Aldinucci M, Danelutto M. Skeleton-based parallel programming: functional and parallel semantics in a single shot. *Comput Lang Syst Struct*. 2007;33(3-4):179-192.
19. Chis AE, González-Vélez H. Design patterns and algorithmic skeletons: a brief concordance. In: Kołodziej J, Pop F, Dobre C, eds. *Modeling and Simulation in HPC and Cloud Systems*. Springer; 2018:45-56.
20. Toshniwal A, Taneja S, Shukla A, et al. Storm@ Twitter. Proceedings of the ACM International Conference on Management of Data; 2014:147-156. ACM.
21. Zaharia M, Xin R, Wendelland P, et al. Apache spark: a unified engine for big data processing. *Commun ACM*. 2016;59(11):56-65.
22. Carbone P, Katsifodimos A, Ewen S, Markl V, Haridi S, Tzoumas K. Apache Flink: Stream and Batch Processing in Single Engine. *Bull Tech Comm Data Eng*. 2015;36(4):28-38.
23. Reinders J. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media; 2007.
24. Aldinucci M, Danelutto M, Kilpatrick P, Torquati M. Fastflow: high-level and efficient streaming on multicore. In: Pllana S, Xhafa F, eds. *Programming Multi-core and Many-Core Computing Systems, Parallel and Distributed Computing*; Wiley; 2017:261-280.
25. del Rio AD, Dolz MF, Fernandez J, Garcia JD. A generic parallel pattern interface for stream and data processing. *Concurr Comput Pract Exp*. 2017;29(24):e4175.
26. Thies W, Karczmarek M, Amarasinghe S. StreamIt: a language for streaming applications. Proceedings of the International Conference on Compiler Construction; 2002:179-196.
27. Griebler D, Danelutto M, Torquati M, Fernandes LG. SPar: a DSL for high-level and productive stream parallelism. *Parallel Process Lett*. 2017;27(1):1740005.
28. Cheng B, Lemos R, Giese H, et al. Software engineering for self-adaptive systems: a research roadmap. In: Cheng B, De Lemos R, Giese H, Inverardi P, Magee J, eds. *Software Engineering for Self-Adaptive Systems*. Springer; 2009:1-26.
29. Weyns D. Software engineering of self-adaptive systems: an organised tour and future challenges. In: Dick Taylor R, Kang K, Cha S, eds. *Handbook of Software Engineering*; Springer; 2017:1-26.
30. Kephart J, Chess D. The vision of autonomic computing. *Computer*. 2003;36(1):41-50.
31. Klinaku F, Zigldrum M, Frank M, Becker S. The elastic processing of data streams in cloud environments: a systematic mapping study. Proceedings of International Conference on Cloud Computing and Services Science—Volume 1: CLOSER, INSTICC; 2019:316-323. SciTePress.
32. Qin C, Eichelberger H, Schmid K. Enactment of adaptation in data stream processing with latency implications—a systematic literature review. *Inf Softw Technol*. 2019;111:1-21.
33. Herodotou H, Chen Y, Lu J. A survey on automatic parameter tuning for big data processing systems. *ACM Comput Surv*. 2020;53(2):43.
34. Kehrer S, Blochinger W. Elastic parallel systems for high performance cloud computing: state-of-the-art and future directions. *Parallel Process Lett*. 2019;29(2):1-20.
35. Kitchenham B, Charters S. Guidelines for performing systematic literature reviews in software engineering. EBSE technical report. EBSE-2007-01; 2007.
36. Gedik B, Schneider S, Hirzel M, Wu KL. Elastic scaling for data stream processing. *IEEE Trans Parallel Distrib Syst*. 2014;25(6):1447-1463.
37. De Matteis T, Mencagli G. Keep calm and react with foresight: strategies for low-latency and energy-efficient elastic data stream processing. *ACM SIGPLAN Not*. 2016;51(8):13.
38. Su Y, Shi F, Talpur S, Wang Y, Hu S, Wei J. Achieving self-aware parallelism in stream programs. *Cluster Comput*. 2015;18(2):949-962.
39. Selva M, Morel L, Marquet K, Frenot S. A monitoring system for runtime adaptations of streaming applications. Proceedings of Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP); 2015:27-34. IEEE.
40. Schneider S, Hirzel M, Gedik B, Wu KL. Auto-parallelizing stateful distributed streaming applications. Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques; 2012:53-64. ACM.
41. Vogel A, Mencagli G, Griebler D, Danelutto M, Fernandes LG. Online and transparent self-adaptation of stream parallel patterns. *Computing*. 2021;1-19.
42. Vogel A, Griebler D, Danelutto M, Fernandes LG. Seamless parallelism management for multi-core stream processing. Proceedings of the International Conference on Parallel Computing (ParCo), Prague, Czech Republic. Vol. 36; 2019:533-542. IOS Press.
43. Schneider S, Andrade H, Gedik B, Biem A, Wu KL. Elastic scaling of data parallel operators in stream processing. IEEE International Symposium on Parallel & Distributed Processing; 2009:1-12. IEEE.
44. Gedik B, Andrade H, Wu KL, Yu PS, Doo M. SPADE: the system S declarative stream processing engine. Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data; 2008:1123-1134. ACM.
45. Choi Y, Li CH, Silva DD, Bivens A, Schenfeld E. Adaptive task duplication using on-line bottleneck detection for streaming applications. Proceedings of the 9th Conference on Computing Frontiers; 2012:163-172. ACM.
46. Tang Y, Gedik B. Autopipelining for data stream processing. *IEEE Trans Parallel Distrib Syst*. 2012;24(12):2344-2354.
47. De Matteis T, Mencagli G. Proactive elasticity and energy awareness in data stream processing. *J Syst Softw*. 2017;127:302-319.
48. Gad R, Kappes M, Medina-Bulo I. Local parallelization of pleasingly parallel stream processing on multiple CPU cores. IEEE Annual Information Technology, Electronics and Mobile Communication Conference; 2016:1-8. IEEE.
49. Karavadara N, Zolda M, Nguyen VTN, Knoop J, Kirner R. Dynamic power management for reactive stream processing on the SCC tiled architecture. *EURASIP J Embed Syst*. 2016;2016(1):14.
50. Sahin S, Gedik B. C-stream: a co-routine-based elastic stream processing engine. *ACM Trans Parallel Comput*. 2018;4(3):15.
51. Schneider S, Wu KL. Low-synchronization, mostly lock-free, elastic scheduling for streaming runtimes. *ACM SIGPLAN Not*. 2017;52(6):648-661.
52. Ni X, Schneider S, Pavuluri R, Kaus J, Wu KL. Automating multi-level performance elastic components for IBM streams. Proceedings of International Middleware Conference, New York, NY, USA; 2019:163-175. ACM.
53. De Sensi D, De Matteis T, Danelutto M. Nornir: a customisable framework for autonomic and power-aware applications. In: Heras DB, Bougé L, Mencagli G, et al., eds. *Euro-Par 2017: Parallel Processing Workshops. Euro-Par 2017*. Lecture Notes in Computer Science. Vol 10659. Springer; 2018:42-54.
54. Vogel A, Griebler D, De Sensi D, Danelutto M, Fernandes LG. Autonomic and latency-aware degree of parallelism management in SPar. In: Mencagli G, Heras DB, Cardellini V, et al., eds. *Euro-Par 2018: Parallel Processing Workshops. Euro-Par 2018*. Lecture Notes in Computer Science. Springer; 2019:28-39.
55. Griebler D, De Sensi D, Vogel A, Danelutto M, Fernandes LG. Service level objectives via C++11 attributes. In: Mencagli G, Heras DB, Cardellini V, et al., eds. *Euro-Par 2018: Parallel Processing Workshops. Euro-Par 2018*. Lecture Notes in Computer Science. Vol 11339. Springer; 2019:745-756.
56. Kahveci B, Gedik B. Joker: elastic stream processing with organic adaptation. *J Parallel Distrib Comput*. 2020;137:205-223.

57. Gulisano V, Peris RJ, Martinez MP, Soriente C, Valduriez P. Streamcloud: an elastic and scalable data streaming system. *IEEE Trans Parallel Distrib Syst*. 2012;23(12):2351-2365.
58. Balkesen C, Tatbul N, Özsu MT. Adaptive input admission and management for parallel stream processing. Proceedings of the ACM International Conference on Distributed Event-Based Systems; 2013:15-26. ACM.
59. Heinze T, Pappalardo V, Jerzak Z, Fetzer C. Auto-scaling techniques for elastic data stream processing. Proceedings of International Conference on Data Engineering Workshops (ICDEW); 2014:296-302. IEEE.
60. Wu X, Liu Y. Enabling a load adaptive distributed stream processing platform on synchronized clusters. IEEE International Conference on Cloud Engineering; 2014:627-630. IEEE.
61. Chatzistergiou A, Viglas SD. Fast heuristics for near-optimal task allocation in data stream processing over clusters. Proceedings of International Conference on Conference on Information and Knowledge Management; 2014:1579-1588. ACM.
62. Martin A, Brito A, Fetzer C. Scalable and elastic realtime click stream analysis using StreamMine3G. Proceedings of International Conference on Distributed Event-Based Systems, New York, NY, USA; 2014:198-205. ACM.
63. Zacheilas N, Kalogeraki V, Zygouras N, Panagiotou N, Gunopulos D. Elastic complex event processing exploiting prediction. Proceedings of International Conference on Big Data (Big Data); 2015:213-222. IEEE.
64. Lohrmann B, Janacik P, Kao O. Elastic stream processing with latency guarantees. Proceedings of International Conference on Distributed Computing Systems; 2015:399-410. IEEE.
65. Mayer R, Koldehofe B, Rothermel K. Meeting predictable buffer limits in the parallel execution of event processing operators. Proceedings of International Conference on Big Data (Big Data); 2014:402-411. IEEE.
66. Mayer R, Koldehofe B, Rothermel K. Predictable low-latency event detection with parallel complex event processing. *IEEE Internet Things J*. 2015;2(4):274-286.
67. Heinze T, Roediger L, Meister A, Ji Y, Jerzak Z, Fetzer C. Online parameter optimization for elastic data stream processing. Proceedings of ACM Symposium on Cloud Computing; 2015:276-287. ACM.
68. Martin A, Smaneoto T, Dietze T, Brito A, Fetzer C. User-constraint and self-adaptive fault tolerance for event stream processing systems. Proceedings of International Conference on Dependable Systems and Networks; 2015:462-473. IEEE.
69. Zhang Q, Song Y, Routray RR, Shi W. Adaptive block and batch sizing for batched stream processing system. IEEE International Conference on Autonomic Computing (ICAC); 2016:35-44. IEEE.
70. Liu Y, Shi X, Jin H. Runtime-aware adaptive scheduling in stream processing. *Concurr Comput Pract Exp*. 2016;28(14):3830-3843.
71. Gil-Costa VG, Hidalgo N, Rosas E, Marin M. A dynamic load balance algorithm for the S4 parallel stream processing engine. International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW); 2016:19-24. IEEE.
72. Li J, Pu C, Chen Y, Gmach D, Milojicic D. Enabling elastic stream processing in shared clusters. IEEE International Conference on Cloud Computing (CLOUD); 2016:108-115. IEEE.
73. Kombi RK, Lumineau N, Lamarre P. A preventive auto-parallelization approach for elastic stream processing. Proceedings of International Conference on Distributed Computing Systems (ICDCS); 2017:1532-1542. IEEE.
74. Kombi RK, Lumineau N, Lamarre P, Rivetti N, Busnel Y. DABS-storm: a data-aware approach for elastic stream processing. In: Hameurlain A, Wagner R, Morvan F, Tamine L, eds. *Transactions on Large-Scale Data- and Knowledge-Centered Systems XL*. Lecture Notes in Computer Science. Vol 11360. Springer; 2019:58-93.
75. De Matteis T, Mencagli G. Elastic scaling for distributed latency-sensitive data stream operators. Proceedings of Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP); 2017:61-68. IEEE.
76. Cardellini V, Grassi V, Presti FL, Nardelli M. Optimal operator replication and placement for distributed stream processing systems. *ACM SIGMETRICS Perform Eval Rev*. 2017;44(4):11-22.
77. Cardellini V, Presti FL, Nardelli M, Russo GR. Towards hierarchical autonomous control for elastic data stream processing in the fog. Proceedings of European Conference on Parallel Processing; 2017:106-117. Springer.
78. Cardellini V, Presti FL, Nardelli M, Russo GR. Decentralized self-adaptation for elastic data stream processing. *Future Gener Comput Syst*. 2018;87:171-185.
79. Cardellini V, Lo Presti F, Nardelli M, RG. Optimal operator deployment and replication for elastic distributed data stream processing. *Concurr Comput Pract Exp*. 2018;30(9):e4334.
80. Cheng D, Chen Y, Zhou X, Gmach D, Milojicic D. Adaptive scheduling of parallel jobs in spark streaming. IEEE INFOCOM Conference on Computer Communications; 2017:1-9. IEEE.
81. Cheng D, Zhou X, Wang Y, Jiang C. Adaptive scheduling parallel jobs with dynamic batching in spark streaming. *IEEE Trans Parallel Distrib Syst*. 2018;29(12):2672-2685.
82. Lombardi F, Aniello L, Bonomi S, Querzoni L. Elastic symbiotic scaling of operators and resources in stream processing systems. *IEEE Trans Parallel Distrib Syst*. 2018;29(3):572-585.
83. Kalavri V, Liagouris J, Hoffmann M, Dimitrova D, Forshaw M, Roscoe T. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. Proceedings of USENIX Symposium on Operating Systems Design and Implementation; 2018:783-798.
84. Wang L, Fu TZ, Ma RT, Winslett M, Zhang Z. Elasticutor: rapid elasticity for realtime stateful stream processing. Proceedings of the International Conference on Management of Data ACM; 2019:573-588.
85. Bartnik A, Monte BD, Rabl T, Markl V. On-the-fly reconfiguration of query plans for stateful stream processing engines. BTW 2019; 2019.
86. Talebi M, Sharifi M, Kalantari M. ACEP: an adaptive strategy for proactive and elastic processing of complex events. *J Supercomput*. 2020;77(5):4718-4753.
87. Das T, Zhong Y, Stoica I, Shenker S. Adaptive stream processing using dynamic batch sizing. Proceedings of the ACM Symposium on Cloud Computing; 2014:1-13. ACM.
88. Tudoran R, Nano O, Santos I, et al. Jetstream: enabling high performance event streaming across cloud data-centers. Proceedings of International Conference on Distributed Event-Based Systems; 2014:23-34. ACM.
89. Heinze T, Jerzak Z, Hackenbroich G, Fetzer C. Latency-aware elastic scaling for distributed data stream processing systems. Proceedings of International Conference on Distributed Event-Based Systems; 2014:13-22. ACM.

90. Mencagli G. A game-theoretic approach for elastic distributed data stream processing. *ACM Trans Auton Adapt Syst*. 2016;11(2):13.
91. Liu X, Dastjerdi AV, Calheiros RN, Qu C, Buyya R. A stepwise auto-profiling method for performance optimization of streaming applications. *ACM Trans Auton Adapt Syst*. 2018;12(4):24.
92. Floratou A, Agrawal A, Graham B, Rao S, Ramasamy K. Dhalion: self-regulating stream processing in heron. *Proceedings of the VLDB Endowment*. 2017;10:1825-1836.
93. Venkataraman S, Panda A, Ousterhout K, et al. Drizzle: fast and adaptable stream processing at scale. Proceedings of the 26th Symposium on Operating Systems Principles; 2017:374-389. ACM.
94. Tolosana-Calasanz R, Diaz-Montes J, Rana O, Parashar M. Feedback-control & queueing theory-based resource management for streaming applications. *IEEE Trans Parallel Distrib Syst*. 2017;28(4):1061-1075.
95. Mai L, Zeng K, Potharaju R, et al. Chi: a scalable and programmable control plane for distributed stream processing systems. *Proc VLDB Endow*. 2018;11(10):1303-1316.
96. Fardbastani MA, Sharifi M. Scalable complex event processing using adaptive load balancing. *J Syst Softw*. 2019;149:305-317.
97. Marangozova-Martin V, Palma N, Rheddane A. Multi-level elasticity for data stream processing. *IEEE Trans Parallel Distrib Syst*. 2019;30(10):2326-2337.
98. Lombardi F, Muti A, Aniello L, Baldoni R, Bonomi S, Querzoni L. PASCAL: an architecture for proactive auto-scaling of distributed services. *Future Gener Comput Syst*. 2019;98:342-361.
99. Abdelhamid AS, Mahmood AR, Daghistani A, Aref WG. Prompt: dynamic data-partitioning for distributed micro-batch stream processing systems. Proceedings of the International Conference on Management of Data, New York, NY, USA; 2020:2455-2469. ACM.
100. Russo GR, Cardellini V, Presti FL. Reinforcement learning based policies for elastic stream processing on heterogeneous resources. Proceedings of International Conference on Distributed and Event-Based Systems; 2019:31-42.
101. Schor L, Bacivarov I, Yang H, Thiele L. Adapnet: adapting process networks in response to resource variations. Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems; 2014:22. ACM.
102. Vilches A, Navarro A, Asenjo R, Corbera F, Gran R, Garzaran MJ. Mapping streaming applications on commodity multi-CPU and GPU on-chip processors. *IEEE Trans Parallel Distrib Syst*. 2015;27(4):1099-1115.
103. Mencagli G, Torquati M, Danelutto M. Elastic-PPQ: a two-level autonomic system for spatial preference query processing over dynamic data streams. *Future Gener Comput Syst*. 2018;79:862-877.
104. De Matteis T, Mencagli G, De Sensi D, Torquati M, Danelutto M. GASSER: an auto-tunable system for general sliding-window streaming operators on GPUs. *IEEE Access*. 2019;7:48753-48769.
105. Stein CM, Rockenbach DA, Griebler D, et al. Latency-aware adaptive micro-batching techniques for streamed data compression on graphics processing units. *Concurr Comput Pract Exp*. 2020;33(11):e5786.
106. Vogel A, Griebler D, Maron CA, Schepke C, Fernandes LG. Private IaaS clouds: a comparative analysis of OpenNebula, CloudStack and OpenStack. Proceedings of Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP); 2016:672-679. IEEE.
107. Hidalgo N, Rosas E, Vasquez C, Wladdimiro D. Measuring stream processing systems adaptability under dynamic workloads. *Future Gener Comput Syst*. 2018;88:413-423.
108. Maron CAF, Vogel A, Griebler D, Fernandes LG. Should PARSEC benchmarks be more parametric? A case study with Dedup. Proceedings of Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), Pavia, Italy: IEEE; 2019:217-221.

How to cite this article: Vogel A, Griebler D, Danelutto M, Fernandes LG. Self-adaptation on parallel stream processing: A systematic review. *Concurrency Computat Pract Exper*. 2022;34(6):e6759. doi: 10.1002/cpe.6759