Runtime Adaptation of Data Stream Processing Systems: The State of the Art

VALERIA CARDELLINI, FRANCESCO LO PRESTI, MATTEO NARDELLI, and GABRIELE RUSSO RUSSO, University of Rome Tor Vergata

Data stream processing (DSP) has emerged over the years as the reference paradigm for the analysis of continuous and fast information flows, which often have to be processed with low-latency requirements to extract insights and knowledge from raw data. Dealing with unbounded dataflows, DSP applications are typically long running and thus, likely experience varying workloads and working conditions over time. To keep a consistent service level in face of such variability, a lot of effort has been spent studying strategies for runtime adaptation of DSP systems and applications. In this survey, we review the most relevant approaches from the literature, presenting a taxonomy to characterize the state of the art along several key dimensions. Our analysis allows us to identify current research trends as well as open challenges that will motivate further investigations in this field.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Computer systems organization** → Self-organizing autonomic computing; • **Information systems** → **Stream management;**

Additional Key Words and Phrases: Data stream processing, adaptation, resource management

ACM Reference format:
Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. 2022. Runtime Adaptation of Data Stream Processing Systems: The State of the Art. *ACM Comput. Surv.* 54, 11s, Article 237 (September 2022), 36 pages.
https://doi.org/10.1145/3514496

1 INTRODUCTION

Our world is increasingly pervaded by “smart” devices, capable of capturing, tracking, and assisting almost every aspect of our life. This ubiquitous presence of devices at the edge of the network, from **Internet-of-Things (IoT)** sensors to wearable devices and smartphones, has fostered a unending growth in the amount of daily produced data, motivating the adoption of expressions like “Big Data” to characterize the resulting datasets in terms of extreme volume, velocity, and variety. Nonetheless, raw data are often of limited value compared to the knowledge and insights that analytics algorithms can extract from them, powering new or improved data-driven applications.

It is often the case that the potential value of data rapidly decreases after their collection, thus, timely processing is necessary. For example, log analysis software can automatically detect security attacks or faults in large-scale computing systems and prevent harm; to do so, these systems

Authors’ address: V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo Russo, Department of Civil Engineering and Computer Science Engineering, University of Rome Tor Vergata, via del Politecnico 1, 00133 Rome, Italy; emails: cardellini@ing.uniroma2.it, lopresti@info.uniroma2.it, {nardelli, russo.russo}@ing.uniroma2.it.

0360-0300/2022/09-ART237 $15.00
https://doi.org/10.1145/3514496

must analyze data as soon as possible, or any reaction could be late. **Data stream processing (DSP)** can be regarded as the reference paradigm for timely analysis of high-volume dataflows, revolving around the idea of processing data as soon as they are available to reduce latency and hence without (or before) storing them. DSP applications process data *streams*, ordered sequences of data units (often referred to as *tuples*, *events*, or *records*) associated with one or more *attributes* (or *fields*), which carry domain-specific information. For processing, data streams flow through a network of so-called *operators*, which apply specific transformations or functions (e.g., filtering, aggregation) and accordingly produce a new output stream [4]. Although the operations performed by single operators can be relatively simple, by chaining and inter-connecting multiple operators into a graph, DSP applications can solve possibly complex queries against the input stream. For instance, general-purpose DSP systems such as Apache Storm and Flink support the definition of queries with arbitrary logic; other systems focus on specific processing paradigms, such as **complex event processing (CEP)**, which aims at detecting high-level situations of interest (e.g., a fault in a manufacturing system) through the analysis of primitive event streams (e.g., sensor measurements).

Real-time stream processing is usually subject to several requirements, as explained by Stonebraker et al. [164], which impact the design and implementation of DSP systems. Just to name a few, data safety and availability must be guaranteed at all times, despite possible failures; processing must be automatically and transparently distributed across multiple processors and machines for the sake of scalability; clearly, systems must deliver low-latency responses in the face of high-volume data streams, introducing minimal overhead. To fully exploit parallel and distributed infrastructures and meet their requirements, DSP applications undergo various optimizations that impact, for example, the scheduling of operators to the available nodes [101], the choice of the parallelism level for each operator [145], and the structure of the application graph itself [73].

However, these optimizations, performed at development- or deployment-time, cannot guarantee a consistent service level for the whole lifetime of DSP applications, which, dealing with unbounded datasets, are kept in execution indefinitely and likely face varying working conditions over time. The long-running nature of DSP applications makes it essential for them to respond and adapt to variations in the working environment (e.g., by means of application elasticity [61, 145] or operator migration [111, 167]), to continue optimizing one or more objectives throughout their life cycle. Indeed, runtime adaptation of DSP applications and systems has received a significant amount of attention from researchers and practitioners so far. Looking at the scientific publications dealing with these issues (Figure 1), we can note that the interest for the topic has significantly increased during the past decade, in conjunction with the widespread development and adoption of Big Data oriented tools, and has consolidated as an active field of research.

The solutions presented in the literature so far have considered a broad spectrum of mechanisms, architectures, and methodologies to introduce adaptation capabilities in stream processing systems. The complexity of the resulting solution space has made it difficult to identify definitive and complete strategies to address the aforementioned challenges, especially as the existing works often target different computing platforms and rely on different assumptions. For this reason, in this work we review, analyze, and classify more than 140 scientific papers dealing with runtime adaptation of DSP systems, with the aim of developing a more mature understanding of both the challenges and the acquired experience in this field.

This survey makes the following key contributions. First, after introducing the key principles underpinning DSP systems, we describe the main challenges and the available mechanisms for their runtime adaptation (Section 2). As our key contribution, we then present a taxonomy of the state of the art based on the well-known "5W1H" (or "Six W's") investigation approach [83], which allows us to classify the most relevant publications from the literature and analyze the current

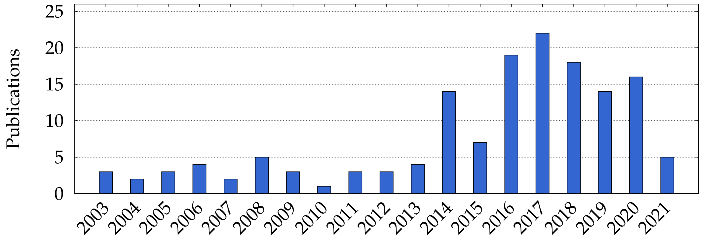


Fig. 1. Number of scientific publications dealing with runtime adaptation of DSP applications. Note: More papers published in 2021 are yet to appear at the time of this writing.

state of research (Sections 3 and 4). Based on our analysis of the state of the art, we outline a few directions for future research on the topic that aim at filling existing gaps in the literature and taking advantage of the opportunities provided by emerging computing landscapes (e.g., serverless, Edge computing) (Section 5). In the supplementary material, we also present a complementary literature review that focuses on the implementation of the adaptation solutions on top of existing DSP frameworks and their evaluation methodologies (Appendix B).

1.1 Related and Complementary Surveys

We briefly review related surveys that complement ours and can be relevant for readers interested in exploring other issues in the context of DSP, or diving deeper into particular aspects.

Important concepts underpinning the stream processing paradigm are presented by Babcock et al. [10] from the perspective of database management systems evolving into data stream management systems. Cugola and Margara [39] provide an overview of the different technologies for timely analysis of information flows, from active databases to CEP engines and general-purpose DSP systems, introducing a modeling framework for their analysis. More recently, the *Dataflow* model was presented by Akidau et al. [4], providing a unified model for computation over unbounded datasets (i.e., data streams) that aims to separate the logical notion of data processing from the underlying implementation.

The evolution of DSP systems and the associated research efforts are presented by Fragkoulis et al. [54]. They highlight that although the foundations of stream processing have remained largely unchanged over the years, early systems, mostly designed as extensions of relational query engines, have evolved into sophisticated and scalable engines, whose applicability exceeds the boundaries of data analytics. Dayarathna and Perera [42] review DSP systems in the broader field of *event processing*, discussing the architectural choices behind the most popular platforms and recent advancements in applications (e.g., online learning, graph analytics). We will provide essential background information in the next section, but we refer the reader to these works for detailed analysis of the DSP paradigm in general and the associated algorithmic and architectural issues.

A systematic literature review of the works dealing with runtime adaptation of DSP systems is presented by Qin et al. [139]. They particularly focus on the mechanisms available for adaptation, whereas we also study the architectural and methodological approaches for adaptation control. With regard to DSP performance management and adaptation, special emphasis has been devoted so far to the issues related to deployment management and particularly application placement and elasticity. Approaches for the initial application placement in distributed environments have been discussed by Lakshmanan et al. [101] and, more recently, by Tantalaki et al. [167], where adaptive strategies are reviewed as well. Salaht et al. [151] review research works that deal with service placement in the Fog/Edge computing scenario, including in their analysis some solutions for distributed DSP application placement. Operator parallelization and scaling are extensively reviewed by Röger and Mayer [145], discussing issues associated with both implementation and control of application elasticity. A compact analysis of elasticity issues in DSP systems, including historical background, is provided by Gulisano et al. [61]. Assunção et al. [9] also review solutions for elastic DSP, with particular emphasis on systems deployed in highly distributed computing environments. The broader spectrum of strategies for resource provisioning and management, including operator scaling and placement, is considered by Liu and Buyya [111], where a taxonomy of the existing solutions is presented.

These surveys complement ours, providing in-depth analysis of the strategies for deployment and resource management. In particular, the surveys of Assunção et al. [9] and Röger and Mayer [145] partially overlap with ours with regard to operator auto-scaling, which we discuss along with the other mechanisms. However, we take the more general perspective of runtime adaptation, which is not limited to deployment reconfiguration and encompasses a variety of mechanisms. For the same reason, our analysis does not comprise optimization techniques applied before application execution (e.g., initial operator parallelization or placement), which are instead discussed in some of the cited works.

Other surveys focus on specific aspects of application development and optimization. Hirzel et al. [73] present a catalog of optimization techniques applied to application graphs (e.g., operator fusion or reordering), discussing the impact and applicability of each one. Herodotou et al. [71] review and classify strategies for automatic parameter tuning (e.g., memory settings, I/O, and network behavior) for data-intensive frameworks, including DSP systems. To et al. [171] analyze the issues associated with state management in both batch and stream processing systems. Zhang et al. [197] instead focus on *hardware-conscious* DSP, reviewing solutions that leverage specialized hardware (e.g., FPGAs) for optimized execution, by means of ad hoc architectures and implementations.

2 BACKGROUND

In this section, we review the main concepts behind DSP systems and applications, especially with regard to their distributed execution. For this purpose, we will look at DSP applications at two different levels of abstraction. We will first present the *abstract* application model and then show how an *execution* model is derived from it. The abstract model, defined at development-time, is a high-level view of the application and its semantics; the execution model extends it to include lower-level information that is necessary to execute the application.

2.1 Abstract Application Model

The fundamental entities comprised in the abstract model are operators and streams. At this level, DSP applications are usually specified as a directed graph $G_{dsp} = (V, E)$, where V is a set of vertices comprising data sources, operators, and sinks (i.e., $V = V_{src} \cup V_{op} \cup V_{sink}$), and E a set of edges (i.e., streams flowing between vertices). Vertices with no incoming edges represent *data sources*, from which input streams originate. Similarly, vertices with no outgoing edges are named *sinks* and represent consumers of the produced results (e.g., dashboards). Note that at this level of abstraction, a source vertex may correspond to a multitude of physical sources (e.g., sensors) that collectively emit a single logical data stream.

Each operator $v \in V_{op}$ is associated with a processing function f_v that is applied to each incoming data unit. Functions may be as simple as filtering or parsing, or more complex, including joins of multiple streams or inference. Application programmers specify the function executed by each operator and the overall topology. To ease this task, DSP frameworks usually provide built-in processing primitives (e.g., filters, maps) and higher-level libraries for common use cases (e.g., graph analytics). An important property of operators regards whether their processing logic

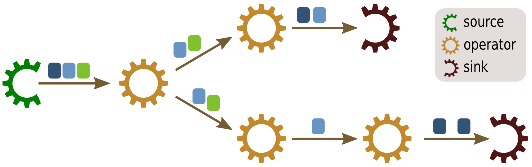


Fig. 2. Example of a DSP topology.

solely depends on the current input or on internal state [171] as well (e.g., partial results, events observed in the past). As such, we can classify operators as either *stateless* or *stateful*.

The resulting graph (Figure 2) is often referred to as application *topology*. Although topologies usually consist of **directed acyclic graphs (DAG)**, cyclic computation is increasingly supported by DSP frameworks (e.g., Flink). Allowing operator output to be (partially) fed back to the same operator is essential to ease the implementation of *iterative* computations (e.g., graph or **machine learning (ML)** algorithms). Furthermore, the graph model is frequently generalized to allow operators forward the same data stream to multiple downstream operators (e.g., to implement different queries on the same data). In this case, the data stream is modeled as a *hyperedge*.

2.1.1 Windows. A special class of stateful operators is represented by *windowed* operators [58], which slice up the incoming stream into chunks (i.e., *windows*) to be processed as a whole. For example, given a stream of e-commerce transactions, we may aim to compute the most frequently purchased items over the last hour; in this case, an operator would count the occurrences of each item within 1-hour-long windows of the transactions stream. Windowing is a necessary mechanism for some aggregation functions. Indeed, since the input stream is assumed to be "infinite," queries such as "find the event with maximum value for attribute A" are only applicable to finite chunks.

Windows can be defined in terms of time, tuple count, or sessions. *Time*-based windows group data from the same time period (e.g., "the last hour"), and their start and end are defined by timestamps. Such timestamps may be either *explicit* (set by data sources as tuple attributes) or *implicit* (set by the DSP system as tuples are received). Explicit timestamps are usually preferred for data associated with the occurrence of real-world events, but they also pose a few challenges, especially in presence of distributed sources. First, the stream ordering with respect to the explicit timestamps may differ from the actual order in which data enter the system. Furthermore, some events might be late or even lost, and it is not obvious how to determine whether a certain window is complete or straggler tuples should be awaited.

A simple approach to deal with stragglers hinges on timeouts, whose expiration causes any missing tuple to be deemed as lost. Unfortunately, setting a proper timeout is tricky, as large values negatively impact processing latency, whereas small values may force many delayed tuples to be discarded. Given the relevance of the issue, researchers have investigated more flexible strategies to cope with out-of-order data and stragglers. In particular, several DSP engines (e.g., Flink, Google Cloud Dataflow) rely on *watermarks* [15, 54] (or *landmarks*), which track the lowest timestamp that may yet appear in a stream. Relying on watermarks, any operator can immediately determine that a certain window is complete and proceed with the computation, as soon as the watermark passes the end of the window. Different watermark implementations have been considered [15], including by means of *punctuations* [174], which consist of special tuples injected into the data stream carrying progress information about one or more attributes (e.g., timestamp).

Alternative window definitions rely on counts or sessions. *Count*-based windows simply group a fixed number of consecutive data items (e.g., "the last 100 events"). *Session*-based windows are dynamically started and completed depending on some "activity" measures (e.g., windows are considered complete after no more events have been received for a certain time interval).

The number of events or time intervals define the size of the window. The *sliding interval* (or *stride*) instead defines the possible overlapping of windows. In particular, we distinguish *tumbling* (or *fixed*) windows and *sliding* windows. Tumbling windows define a *partitioning* of the input stream, as they never overlap (i.e., size and sliding interval coincide). Conversely, sliding windows can overlap with each other and single tuples may be included in multiple consecutive windows.

2.2 Execution Model

When it comes to running the application, the abstract model must be converted into an execution model, containing additional information on how each operator shall be executed. The execution model usually consists of a new graph $\bar{G}_{dsp}$, where vertices of the abstract model G_{dsp} are replaced by lower-level entities that the system can deploy for execution (e.g., sources and sinks may be connectors to external systems, whereas operators may correspond to processing threads). As for the abstract model, edges in the graph represent data streams flowing between vertices, corresponding at runtime to, for example, network connections or inter-process communication channels.

As many other software systems, DSP applications aim to exploit the parallelism provided by modern parallel and distributed infrastructures. In particular, operator execution leverages three forms of parallelism, namely task parallelism, pipeline parallelism, and data parallelism. *Task parallelism* is a natural consequence of the graph-based application model, where multiple queries on the same data streams can be performed in parallel by operators along different paths. Applications also enjoy *pipeline parallelism*, as operators along the same path process the stream concurrently (i.e., while an operator processes tuple i, its predecessor may be processing tuple $i + 1$). *Data parallelism* instead consists of executing multiple parallel instances of the same operator, each processing a portion of the incoming stream (usually on different processors), so that the operator can sustain higher data rates. For this purpose, in the execution model each operator $v \in V_{op}$ is replaced by $n \geq 1$ parallel *instances* (or *replicas*) $\bar{v}_1, \ldots, \bar{v}_n$.

Operator parallelization is the most popular modification applied when deriving the execution model [145], but several other techniques are available for such static application optimization (e.g., see [73]). For instance, to reduce communication overhead, sequences of two or more operators defined in the abstract model may be replaced by a single fused processing element in the execution model, with equivalent semantics (e.g., adjacent operators u and v may be replaced by x such that given an input tuple t, $f_x(t) = f_v(f_u(t))$).

2.3 Application Deployment and Execution

Once the execution model is available, DSP systems need to deploy the operator instances in the available computing nodes and start their execution. Operator instances are usually launched as concurrent threads or processes, enjoying the aforementioned pipeline parallelism. For this reason, this class of systems is also referred to as *pipelined* DSP systems, with the most notable alternative being represented by *micro-batched* stream processing [192] (used, e.g., in Spark Streaming). Systems adopting this paradigm exploit MapReduce-inspired batch processing techniques, which target large but finite data collections. To cope with unbounded dataflows, these systems split the input streams into small chunks (i.e., micro-batches) and apply batch processing techniques to one micro-batch at a time. The main drawback of this approach consists of the extra latency caused by data buffering, as tuples can only be processed when a micro-batch is complete. Traditional DSP frameworks belong to the group of pipelined systems, which is also the main focus of this work.

To start application execution, a *scheduler* component takes care of mapping the execution graph onto the computing infrastructure, associating each operator instance with a node. This process is known as *operator placement* and has significant impact on application **Quality-of-Service (QoS)**, as the available nodes may differ in capacity, reliability, and usage cost. Furthermore, the choice of nodes where instances must be executed implies a decision on the network links through which streams will flow. For instance, if a pair of adjacent operators is deployed in the same node (i.e., they are co-located), they will likely communicate through efficient inter-process communication mechanisms. Conversely, if operator instances are deployed in different nodes (or even in different data centers), data will need to traverse the network, incurring delay and possible loss.

Having to process unbounded datasets, notions of "completion" are not easily applicable to DSP applications, which usually execute for unbounded amounts of time. On the one hand, at some point applications must likely deal with failures in the underlying software and hardware stacks, which can lead to degraded performance or erroneous query results. For this reason, the presence of integrated fault tolerance mechanisms is a fundamental requirement for modern DSP systems (e.g., state checkpointing [24, 52], active replication [12, 70]). On the other hand, given their long-running nature, DSP applications must cope with varying conditions at runtime that make static optimizations unable to guarantee the desired service level in the long term.

First of all, application workloads are often variable and difficult to predict. With few exceptions, data streams originate upon the occurrence of events that can follow complex, non-deterministic dynamics. For instance, applications for social network analysis may be subject to sudden load peaks when new "trending topics" appear. Additional sources of performance variability come from the computing infrastructures, especially as DSP applications are increasingly moved out of traditional Cloud data centers and deployed in Fog/Edge platforms. These environments, offering computing resources located at the edge of the network and closer to data sources, are attractive to reduce latency, but they also force applications to face new issues such as increased resource heterogeneity, reduced processing capacity, unstable connectivity, and non-negligible network latency between nodes. In particular, network conditions hardily stay unchanged over time, possibly requiring operators to be migrated to different nodes to avoid performance degradation.

All of these aspects must be necessarily taken into consideration at runtime to optimally use the available resources. To handle such variability and keep a consistent service level, DSP applications should be able to self-adapt at runtime. In other words, applications need mechanisms to modify, for example, their own deployment configuration, the accuracy of the processing algorithms they execute, and the rules used to route data among operators in response to external changes. As we will discuss in the next section, several adaptation mechanisms exist for DSP applications that act on different properties or components of the system at runtime.

Adaptation mechanisms must be clearly complemented with suitable control policies, to decide when and how adaptation actions should be triggered at runtime, according to user-specified QoS requirements. Optimally planning adaptation is a difficult task, mostly because of the uncertainty that characterizes workloads and the lack of accurate application performance models. Furthermore, adaptation often comes at a price, in terms of overhead, as many application aspects can only be modified at runtime through suitable reconfiguration protocols, to preserve the integrity of data streams and operator internal state. As the overhead of such protocols may be significant (e.g., application must be paused), carefully planning adaptation is of paramount importance.

2.4 Runtime Adaptation Mechanisms

Adaptation mechanisms provide tools to alter the configuration and the behavior of DSP applications during execution. A large number of mechanisms have been considered in the literature so far and—as illustrated in Figure 3—we classify them into the following categories: topology adaptation, deployment adaptation, processing adaptation, overload management, fault tolerance adaptation, and infrastructure adaptation.

Topology adaptation (or, *query replanning*) mechanisms modify the DSP application topology, usually keeping the resulting semantics unchanged. As surveyed in the work of Hirzel

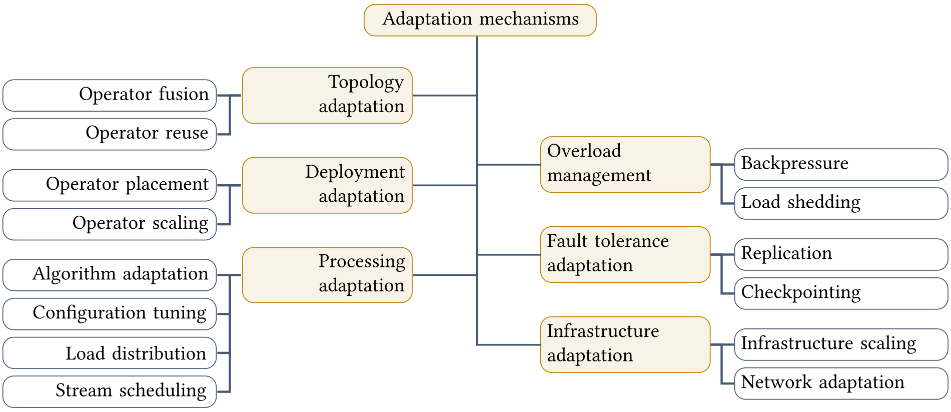


Fig. 3. Categorization of the main adaptation mechanisms.

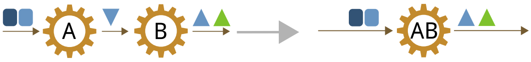
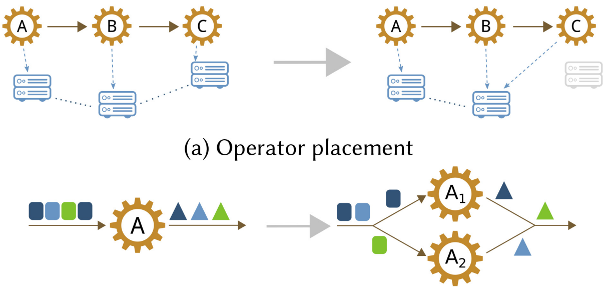


Fig. 4. Example of operator fusion.

et al. [73], several optimization techniques can be applied to DSP topologies even at development- or deployment-time. For runtime adaptation, two mechanisms have received particular attention, namely operator fusion and reuse. *Operator fusion* replaces a pipeline of two or more operators with a single one that carries out the same processing functions of the whole pipeline (Figure 4). The idea is to reduce the communication overhead due to data exchange, provided a single operator can efficiently handle the whole processing logic. *Operator reuse* is used in presence of multiple applications or queries that work on the same input streams and hence likely apply identical data transformations in the early stages of processing (e.g., parsing or filtering raw input data). To avoid redundant computation, DSP systems can let applications or queries reuse the same operator instances and share the produced output streams. To apply this technique at runtime, DSP systems must be able to verify opportunities for reuse as soon as users submit, update, or terminate their applications.

Deployment adaptation mechanisms act on the allocation of computing resources to DSP operator instances. These mechanisms have been widely investigated, for example, within the context of *placement* strategies for DSP, which is how application graphs should be mapped onto the computing infrastructure, deciding which node will host each operator instance. Operator placement is necessarily made when the application is first deployed, but operators may also be migrated at runtime (Figure 5(a)) in response to changes in the infrastructure (e.g., variations in the network, availability of new nodes). Another relevant mechanism is *operator scaling*, which elastically adjusts the amount of resources allocated to each DSP operator as needed, responding to workload variations. In particular, operators can be scaled either horizontally or vertically. *Horizontal* operator scaling, illustrated in Figure 5(b), leverages *data parallelism* to deploy parallel instances of the same operator, each processing a share of the input stream. By dynamically adjusting the number of active instances as needed, operators sustain large data volumes while avoiding resource overprovisioning. *Vertical* operator scaling instead does not alter the parallelism level and hinges on the dynamic allocation of computing resources (e.g., CPU time, memory) to the existing instances. In general, vertical scaling provides limited scalability compared to horizontal scaling, as the allocated resources cannot exceed the capacity of the computing node where operators are



(a) Operator placement

(b) Operator scaling

Fig. 5. Examples of deployment adaptation mechanisms.

currently deployed. Nonetheless, vertical scaling usually benefits from negligible adaptation overhead, whereas reconfiguring the operator parallelism level leads to significant overhead, because specific reconfiguration protocols must be used to preserve stream and internal state integrity.

Processing adaptation mechanisms directly act on the way data are processed, comprising several techniques such as algorithm adaptation, configuration tuning, load distribution, and stream scheduling. *Algorithm adaptation* mechanisms act on the algorithm executed by operators (e.g., trading-off computation accuracy with processing load). For instance, an operator may dynamically switch between exact and approximate computation depending on the workload. Similarly, *configuration tuning* techniques adjust configuration parameters at runtime, thus altering the behavior of the system. However, although algorithm adaptation changes the processing logic of operators, this class of mechanisms only impacts system parameters (e.g., operator buffer size) keeping the processing algorithm unchanged. Within this group, *dynamic batch sizing* is particularly relevant for micro-batched DSP systems, where a fundamental choice is how much data to include in each micro-batch, as larger batches improve resource utilization but lead to higher buffering latency.

Other mechanisms manipulate the stream themselves. In the presence of parallel instances of operators, *load distribution* (or *stream partitioning*) mechanisms change the way incoming data are routed to the various instances, aiming, for example, to balance the processing load. Whereas this task can be solved by means of traditional load balancing techniques in the presence of stateless operators, stateful operators require more attention. Indeed, to avoid altering the application semantics, each data unit may be required to be sent to a specific operator instance (e.g., based on a key attribute). For this reason, load distribution mechanisms may be forced to migrate portions of the operator internal state every time they change the data routing schemes.

Another mechanism available in this group is *stream scheduling*, which is determining the order in which computation on data has to be performed. This can be realized by altering the order in which tuples, groups of tuples, or micro-batches should be processed. Although many applications require the original data ordering to be preserved during computation, it is sometimes possible to increase resource utilization efficiency by processing locally buffered data in a different order. Furthermore, if operators are not executed concurrently over the computing infrastructure, a scheduling decision must also be made about which operator has to be executed at any time on the available processing units, even though the ordering of data is not changed.

Some adaptation mechanisms specifically target situations where DSP systems must face an excessive volume of input data. These *overload management* mechanisms aim at mitigating the overload to reduce performance degradation. The most relevant mechanisms in this group are backpressure and load shedding. *Backpressure* is a mechanism to propagate overload notifications from operators backwards to the data sources in the topology so that data emission rates can be throttled to alleviate overload. When this happens, tuples that cannot be immediately emitted by the sources are usually kept in buffers and not discarded. Conversely, *load shedding* techniques aim at reducing the processing load by dropping some input tuples. To do so, these mechanisms may try to identify “less interesting” data in the input stream, with respect to application-dependent criteria, so as to minimize the impact on results accuracy. It is worth remarking that other mechanisms (e.g., operator scaling) we have mentioned also help to deal with large volumes of data. However, overload management mechanisms differ from them, as they mainly represent “emergency tools” rather than strategies to avoid overload in the long term.

Another class of mechanisms acts on the *fault tolerance* strategies embedded in DSP systems. The key observation behind these mechanisms is that fault tolerance comes at the cost of additional computational or communication demand (e.g., extra operator load due to periodic state checkpointing). As such, these mechanisms dynamically trade off fault tolerance and computational overhead based, for example, on current workload. Examples of mechanisms in this group are adaptive replication and adaptive checkpointing. Active replication consists in running redundant replicas of DSP operators, which increase application resiliency and possibly reduce tail processing latency in case of timing issues [173], at the cost of possible increase of provisioning costs and state management overheads. In this survey, we focus on *adaptive active replication*, in which the degree of operator replication is dynamically adjusted based, for example, on the currently available computing resources. *Adaptive checkpointing* instead regulates the frequency and the granularity of state checkpoints and tuple acknowledgments that enable processing recovery in case of failures. Although in principle these operations should be performed as frequently as possible, adaptive mechanisms trade off the risk of data loss with checkpointing overhead.

The last group of adaptation mechanisms we identify copes with *infrastructure adaptation*, thus comprising all of those operations—possibly not specifically designed for DSP systems—used to manage the computing infrastructure. Among them, the most relevant mechanism for DSP systems is *infrastructure scaling*, which consists of elastically scaling the number of computing nodes in the infrastructure, for example, to accommodate a larger number of application components. Infrastructure scaling is often coupled with horizontal operator scaling, so as to dynamically add (or remove) computing nodes based on the current number of operator instances in use. Moreover, looking at the network level and thus at the exchange of data streams between distributed nodes, the advancements in the area of **software-defined networking (SDN)** have created opportunities to perform *network adaptation* (e.g., to dynamically allocate bandwidth to operators and applications).

3 BUILDING A TAXONOMY OF ADAPTIVE DSP SOLUTIONS

The great variety of mechanisms available for DSP adaptation have been extensively investigated by the research community. In this survey, we explore the most relevant solutions from the literature, analyzing how they cope with key questions in the adoption of the different mechanisms (e.g., when adaptation should be triggered, which metrics should be taken into account).

Our analysis considers more than 140 published research works on the topic, which have been reviewed and classified.[1] To give an overview of the wide spectrum of DSP adaptation approaches investigated so far, we classify the most relevant solutions along several dimensions, which are inspired by the *5W1H* (or *Six Ws*) pattern, which is widely used in the journalism domain: ***what***, ***why***, ***who***, ***when***, ***where***, and ***how***. As depicted in Figure 6, these questions helped us identifying the following relevant features.

[1] See Appendix A for information on how the publications have been selected.

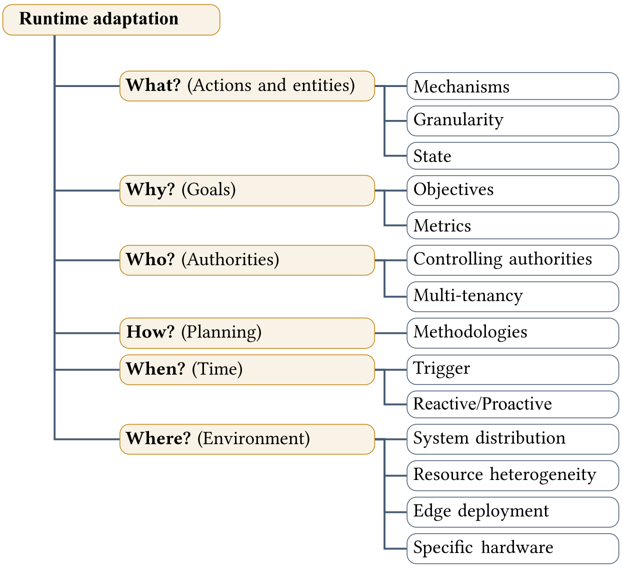


Fig. 6. Dimensions used to classify existing adaptation solutions, inspired by the *5W1H* approach.

What? This question deals with the actions performed to adapt DSP applications and the targeted entities. In particular, we first determine which adaptation *mechanisms* are exploited, hence identifying the type of actions performed at runtime (e.g., operator scaling). We also investigate the *granularity* at which actions are performed (e.g., tuple, operator), to better characterize the adapted entities. Furthermore, we verify whether the operator *internal state* is involved in the adaptation enactment, as it possibly represents an additional entity to take care of.

Why? This question investigates the motivation behind the design of the adaptation strategy. We characterize existing solutions looking at their *objectives*, which may consist, for example, in optimizing one or more QoS metrics, or satisfying some constraints. The set of considered *metrics* is a relevant aspect as well, given the variety of functional and non-functional attributes adopted in the literature.

Who? This question aims to identify the *authorities* responsible for decisions regarding the adaptation of DSP applications. In practice, we are interested in determining how decisions are made at runtime within the (possibly complex) architecture of large-scale DSP systems. This is a relevant aspect as not all control schemes are equally effective or scalable (e.g., centralized control schemes often suffer from scalability issues) Moreover, DSP systems may host applications falling under the responsibility of multiple *tenants*, thus, we check whether adaptation solutions explicitly consider multi-tenancy scenarios.

How? This question investigates the *methodology* adopted to evaluate and plan adaptation actions (i.e., to devise the adaptation control policy). As shown in the following, a wide spectrum of approaches have been exploited in the literature, ranging from simple heuristics to model-based approaches and ML techniques.

When? This question investigates time-related aspects of adaptation decisions. Two main issues must be addressed in this context. First, it is important to determine when adaptation should be *triggered*. For instance, adaptation actions could be planned periodically or only triggered upon the occurrence of particular events. Second, we distinguish between *reactive* and *proactive* solutions.

Where? This question deals with the computing environment targeted by each work, which can significantly impact the design and implementation of adaptation schemes. First of all, we

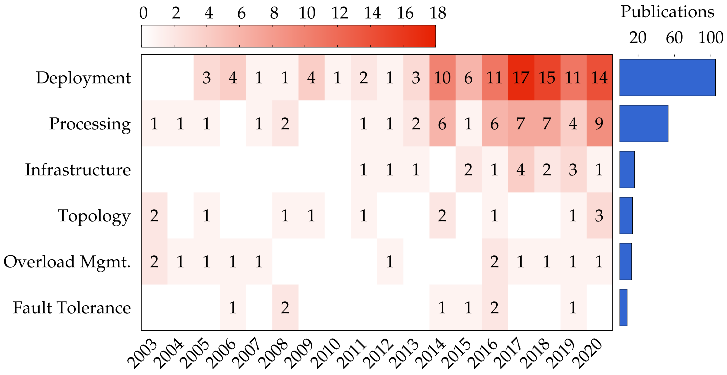


Fig. 7. Popularity of the different groups of adaptation mechanisms per year.

verify whether a *distributed* system is considered, and possibly whether geographical distribution is admitted. Furthermore, we check whether homogeneous computing resources are assumed to be available in the considered infrastructure, or instead *resource heterogeneity* is contemplated. We also verify whether *Edge-based deployments* are considered and whether the work assumes the availability of *specific hardware* (e.g., GPUs).

4 TAXONOMY OF ADAPTIVE DSP SOLUTIONS

Based on the approach introduced previously, we analyze and compare the most relevant research contributions in the area of self-adaptive DSP. A detailed classification of each reviewed work is reported in Appendix A, where we also provide an illustration of the complete taxonomy. In the following, we will discuss our main findings for every dimension considered in our taxonomy.

4.1 What: Adaptation Actions and Controlled Entities

4.1.1 Mechanisms. Adaptation mechanisms represent the actions available to adapt applications and their components at runtime. As already described in Section 2.4, for the sake of analysis, we have classified them into the following groups: topology adaptation, deployment adaptation, processing adaptation, overload management, fault tolerance adaptation, and infrastructure adaptation. As shown in Figure 7, not all groups have received the same attention so far within the research community. Indeed, deployment adaptation mechanisms have been explored way more than the other tools. Processing and infrastructure adaptation mechanisms are the most popular groups among the remaining ones, and the interest for them has increased during the past decade. The other groups have received a limited amount of attention so far.

Topology adaptation techniques are mostly used for static application optimization, but a few works have applied them for runtime adaptation. For instance, Lohrmann et al. [114], Madsen et al. [118] and Wang et al. [181] leverage operator fusion, combined with other mechanisms, to improve performance at runtime. They conveniently combine sequences of operators into “components” (i.e., fused operators), hence reducing (i) the number of processing entities (e.g., threads) required for execution and (ii) the amount of data exchanged between operators. Operator fusion and reordering at runtime have been first exploited in Aurora [2]; in particular, reordering is driven by a performance model that takes into account the operators’ execution time and selectivity (i.e., number of output tuples per input tuple). Jonathan et al. [85] instead exploit various runtime query replanning tools (e.g., reordering operators) to reduce network usage in wide-area stream processing systems. In particular, they focus on the ordering of aggregation operators, which possibly require a significant exchange of data between multiple geographical regions.

Reuse of operators is instead exploited to improve resource efficiency when multiple queries or applications need to apply identical operations on the same input data stream in several works [35, 91, 102, 142]. These solutions are able to automatically detect opportunities of reuse, verifying equivalence between operations and data streams. For instance, Chaturvedi et al. [35] target the specific scenario of IoT analytics, where dataflows generated by devices are likely processed by multiple streaming topologies for different purposes, and reuse can avoid resource wastage.

Deployment adaptation mechanisms are by far the most investigated ones, as illustrated earlier, with both operator placement and scaling being widely adopted. Changing the placement of operators at runtime is necessary to achieve consistent QoS in face of workload and infrastructure condition variations (e.g., increasing network congestion or the availability of new computing nodes may conveniently trigger new placement decisions). Indeed, in addition to the large number of solutions to the initial placement problem (e.g., [40, 45, 128]), adaptive placement strategies have been developed as well (e.g., [7, 48, 84, 116, 117, 138, 144, 184, 186]). For instance, Aniello et al. [7] design an online placement solution for Apache Storm, which migrates operators at runtime based on continuously monitored performance metrics. Luthra et al. [116] consider the placement problem for CEP operators in a dynamic users environment and show how different placement techniques may be required to fulfill QoS requirements of different applications, or during different time periods (e.g., “rush hours”). Therefore, they present a *transition* strategy to switch between different placement techniques at runtime.

A slightly different approach than placement to tackle the deployment adaptation is presented by Gu et al. [59], who consider the operators composition problem to select and connect already deployed operators into user-required DSP applications with QoS requirements. The selection occurs by adaptively probing a subset of candidates to discover an optimal composition.

A common issue faced by adaptive placement solutions is how to efficiently migrate operators whenever their deployment must be updated. As we will also discuss in the following, this task is particularly challenging in presence of stateful operators, whose internal state must be migrated as well. Aiming to reduce the migration overhead, several works present improved mechanisms to make DSP operator migrations smoother (e.g., [75, 125, 137]).

As elasticity is considered a key feature for modern DSP, a large number of works investigated solutions for operator scaling. We note that most of them focus on horizontal operator scaling (e.g., [27, 52, 53, 55, 57, 60, 98, 113, 115]). In practice, the number of parallel operator instances is adjusted by starting (or terminating) threads or processes where instances are executed. Unfortunately, not all of the most popular DSP frameworks offer native support for efficient horizontal operator scaling, with each parallelism adaptation causing significant reconfiguration overhead. For this reason, researchers have been also extending existing frameworks or studying different implementations to enable seamless operator scaling (e.g., see [16, 127, 159, 179]).

A smaller number of strategies involves vertical operator scaling (e.g., [44, 76, 121, 147, 161]), where the amount of computational resources allocated to operator instances is dynamically adjusted rather than the parallelism level. As mentioned previously, vertical scaling hinges on lower-level mechanisms to alter resource allocation or configuration as required, usually with negligible adaptation overhead. For instance, De Matteis and Mencagli [44] exploit dynamic voltage and frequency scaling of modern CPUs to realize a vertical scaling solution. Hoseiny Farahabady et al. [76] rely on Linux *cgroups* to allocate CPU shares to operators at operating system-level.

The main disadvantage of vertical scaling is the limited scalability it provides. Indeed, without increasing the number of instances for operators, the amount of extra computational capacity that can be supplied to operator instances on demand is limited (e.g., CPU frequency cannot be increased beyond the maximum value supported by hardware). To address this issue, a few works combine vertical and horizontal scaling. For example, De Matteis and Mencagli [44] exploit horizontal scaling for performance-oriented adaptation, whereas vertical scaling is used for energy-aware adaptation.

The group of *processing adaptation* mechanisms is the second most investigated in the literature. Among them, algorithm adaptation is used, for example, in several works [66, 94, 100, 104, 195]. For instance, Heintz et al. [66] consider a DSP system spanning Edge and Cloud data centers, and devise a strategy to adapt the amount of computation to be performed at the edge, taking into account both the amount of data sent over network and the "freshness" of data reaching the Cloud.

Configuration tuning can be used to adapt system configuration at runtime, as in done in several works [22, 38, 41, 114, 175]. For instance, Cammert et al. [22] adjust the size of time-based windows and time granularities on the basis of a detailed cost model; Cheng et al. [38] use a learning algorithm to adjust the scheduling parameters for a micro-batch streaming system. Lohrmann et al. [114] dynamically size operator output buffers based on current load; Tudoran et al. [175] instead optimize the size of the data batches transferred between operators in a geographically distributed DSP system;

Several research works investigate load distribution and routing strategies (e.g., [3, 23, 31, 50, 92, 95, 107, 143]). For instance, Rivetti et al. [143] present a solution to balance load among parallel instances of a stateless operator, accounting for variable tuple processing times. Katsipoulakis et al. [92] instead propose "holistic" stream partitioning strategies for stateful operators, where both load imbalance and processing overhead are considered. TelegraphCQ [31], an early-generation DSP framework, adaptively determines the data routing to operators on a tuple basis. It also provides load balancing through partitioning of the stream and the corresponding operator state by means of Flux [157], whose policy tries to maximize the benefit of rebalancing while minimizing the number of moved partitions.

Scheduling mechanisms are investigated in several works [18, 38, 51, 133, 177, 199]. This kind of adaptation is applied both to traditional and micro-batched stream processing. For instance, Bellavista et al. [18] present a priority-based tuple scheduling solution, where incoming tuples are reordered based on the priority level of their destination operator. Farhat et al. [51] target window-based operators, whose execution is frequently blocked, and exploit watermarks to robustly infer stream progress based on window deadlines and network delay, and schedule operator execution accordingly. Conversely, Palyvos-Giannas et al. [133] directly interact with the operating system scheduler to dynamically adjust priorities of multiple DSP applications and operators, based on their performance requirements. Cheng et al. [38] instead target micro-batched streaming systems and propose an adaptive scheduler for micro-batches.

When facing transient load peaks, *overload management* mechanisms can help in limiting performance degradation. Backpressure, which is considered as an overload symptom in some works (e.g., [53]), has been exploited as a mechanism as well in other works [6, 37]. For example, Chen et al. [37] present a backpressure controller that predicts the future cost of checkpointing and adjusts the flow rate to accurately control the input size during checkpointing, when processing capacity is reduced.

Load shedding has been extensively studied in the literature (e.g., [1, 2, 11, 89, 90, 93, 163, 168, 169]). For instance, Babcock et al. [11] formalize an optimization problem with the objective of minimizing its adverse impact on the results accuracy within the limits imposed by load constraints and study where to insert load shedding operators in the application graph G_{dsp}. Kalyvianaki et al. [89] present a feedback control based approach to satisfy latency constraints by dropping data during overload periods. Tatbul et al. [168] introduce two approaches for load shedding: one drops a fraction of the tuples in a randomized fashion, whereas the other drops tuples based on the importance of their content. Similarly, Katsipoulakis et al. [93] present a solution based on *concept-driven* load shedding, where the tuples to be dropped are selected so as to maximize processing accuracy.

Fault tolerance plays an important role in DSP. On the one hand, interruptions in stream computation can have a dramatic impact on latency; on the other hand, failures cannot be avoided, especially in distributed environments, thus efficient recovery mechanisms are necessary. A small number of research works have investigated approaches for adaptive fault tolerance [17, 46, 49, 70, 79, 81]. Among them, Bellavista et al. [17] and Heinze et al. [70] exploit active replication to guarantee fault tolerance and trade off replication degree with resource consumption. Similarly, Fang et al. [49] focus on active replication and integrate it with stream routing techniques to balance load among operator replicas while also minimizing the recovery time from faults. Hwang et al. [81] rely on active replication across a wide area and focus on replication transparency to deliver what non-replicated processing would produce without failures. Du and Gupta [46] adapt the completion timeout associated with tuples so as to quickly replay straggler data units and limit the increase in latency during recovery phases. Huang and Lee [79] build on a notion of *approximate fault tolerance*, whose idea is to mitigate backup overhead by adaptively issuing backups while ensuring that errors upon failures are bounded with theoretical guarantees.

At the *infrastructure level*, adaptation mainly consists in infrastructure auto-scaling, to (i) complement operator scaling and provision computing nodes as needed, and (ii) to adapt the amount of allocated resources as new applications are submitted for execution (or the running ones are stopped). As for operator scaling, a significant amount of effort has been spent on this issue (e.g., [52, 82, 115, 120, 141, 177]). Depending on the considered platform, infrastructure scaling is implemented by scaling the number of active **virtual machines (VMs)** (e.g., [82]) or containers (e.g., [120]). Infrastructure scaling is often coupled with operator scaling to achieve *multi-level* elasticity solutions (e.g., [115, 120]).

Aljoby et al. [5] exploit SDN to adapt the infrastructure at the network level. They dynamically provision network bandwidth for streams flowing between nodes over a multi-hop network, based on the demand monitored at application level.

4.1.2 Granularity. The granularity level of adaptation in the majority of the considered approaches is the single operator (e.g., [6, 14, 21, 86, 87, 112, 132, 158, 180]) or small groups of operators (e.g., [35, 62, 198]). For instance, Guo and Zhou [62] present a *component-based* solution to the operator scaling problem, where groups of operators are combined into so-called components based on the amount of data they exchange with each other, and scaling actions are applied on whole components instead of single operators. A few works also consider whole applications (e.g., [41, 66, 196]).

Several solutions, especially those acting on data streams (e.g., load distribution, shedding), perform adaptation with finer granularity, at level of single tuples (e.g., [3, 23, 50, 95, 168, 185]) or batches of tuples (e.g., [38, 177]). Solutions acting at the infrastructure level usually work with the granularity of the computing node (e.g., [47, 82, 176]) or the network link [5].

4.1.3 State. Operator internal state represents an additional challenge for adaptation, because its consistency must be preserved across configuration changes, and it might be necessary to move the state itself along with operators when deployment is modified [171]. Indeed, most of the existing solutions take into account the presence of stateful operators and design adaptation strategies consequently. In some cases, new mechanisms must be designed and implemented to overcome limitations of existing DSP frameworks (e.g., [27, 52, 179]). For instance, Fernandez et al. [52] present an integrated approach for auto-scaling and fast recovery of stateful operators, whereas Cardellini et al. [27] and Wang et al. [179] extend Apache Storm to allow for horizontal scaling of stateful operators. There are also a few works where state management is not included, either because optimizations for stateless operators are proposed (e.g., [143]) or support for stateful adaptation is left under the responsibility of application developers (e.g., [112]).

It is worth noting that the stateless or stateful nature of the operators under control should be taken into account when picking the adaptation mechanisms to use. Indeed, some mechanisms better suit stateless operators (e.g., load distribution enjoys more flexibility when streams can be rerouted without moving state; similarly, placement of stateless operators can be reconfigured more easily), whereas other mechanisms instead are mostly meaningful for stateful queries (e.g., adaptive state checkpointing).

4.2 Why: Adaptation Goals

4.2.1 Objectives. Adaptation actions are usually motivated by one or more goals, defined, for example, in terms of performance or operational costs. Specifically, adaptation aims at optimizing one or more metrics, satisfying some requirements, or a combination of both. For instance, just looking at placement adaptation solutions, we can find a variety of approaches. Li et al. [108] formulate a single-objective optimization problem, aiming to minimize application latency; Ottenwälder et al. [132] instead formulate a multi-objective problem, where conflicting metrics (e.g., network traffic, latency, and adaptation overhead) are considered; similarly Madsen et al. [117] and Silva Veith et al. [160] devise a multi-objective formulation and also add constraints to the problem (e.g., maximum migration overhead [117] and limited network and node capacity [160]).

4.2.2 Metrics. A broad spectrum of different metrics have been used to specify the adaptation objectives and requirements. We classify them as either application oriented or system oriented. *Application-oriented* metrics capture aspects of application operation that can be directly perceived by users (e.g., latency, processing accuracy); *system-oriented* metrics instead capture aspects of the system that can have impact on application but are usually observed at level of the underlying DSP system (e.g., resource utilization).

Among application-oriented metrics, the most popular ones are processing latency and throughput. *Latency* (or *response time*) plays an important role, as DSP applications are often required to process events with (near) real-time requirements, and many adaptation solutions rely on latency as a key performance metric (e.g., [55, 88, 89, 108, 178, 180, 196]). Latency refers to the time it takes to process data units since they enter the system, although slightly different definitions are used in practice. Indeed, latency may be defined at level of single operator (e.g., [55, 180]) or as end-to-end latency along source-to-sink paths in the application DAG (e.g., [88, 166]). Moreover, latency experienced by data units may be simply defined in terms of time spent waiting in buffers and being processed (e.g., [55], or [196], where batching time is also considered); or, alternatively, it can be defined as the difference between the current wall-clock time and the latest fully processed event timestamp (e.g., [178], where *watermarks* [4] are exploited). Latency is often used to formulate soft real-time constraints (e.g., [20, 37, 172, 189, 199]), requiring input tuples to be fully processed within given *deadlines.* In these works, adaptation is often aimed at minimizing deadline violations (i.e., *missed* deadlines in real-time terminology) [20, 172]. A similar approach is adopted by Zhou et al. [199], who associate each tuple with a utility value in a time-critical DSP system, assigning positive utility only to tuples processed within their deadline. A complementary metric to latency is *slowdown* (e.g., used in the work of Sharif et al. [158]), which is defined as the ratio of latency to the ideal processing time and can be more appropriate than latency in case of heterogeneous workloads.

Besides latency, another popular performance metric is *throughput* (e.g., [30, 82, 86, 87, 130, 131, 188]), which measures the number of data units processed per unit of time. Throughput is mainly adopted within operator scaling solutions (e.g., [86, 87]), where the number of parallel instances is adjusted in a way that allows application throughput to sustain the incoming data rate.

In addition to processing performance, we are sometimes interested in evaluating the "quality" of the results computed by the application, especially when using adaptation mechanisms that may have an impact on it. We indicate metrics used for this purpose as *accuracy* metrics, although in practice different metric definitions are used in the reviewed works [66, 93, 104, 195]. For instance, Le Quoc et al. [104] instead present StreamApprox, a framework for approximate stream processing, where achieved accuracy is estimated using statistical theory. Heintz et al. [66] also rely on algorithm adaptation to adjust the amount of computation performed on Edge nodes in a geo-distributed DSP system. They consider *staleness* as the reference metric, which measures the delay in getting the expected results. Therefore, accuracy in this context is not only about getting the exact results but also about the time we get those results.

Similarly, other works (e.g., [17, 136, 190]) look at the content of data streams, but they measure data loss instead of accuracy. For example, Zacheilas et al. [190] propose an operator scaling strategy that accounts for potential loss of data caused by resource underprovisioning. Bellavista et al. [17] introduce *internal completeness* metrics in their adaptive fault tolerance solution, where the amount of data lost in presence of different levels of replication are estimated.

Other metrics, such as resource cost and adaptation overhead, provide information about the downsides of performance and accuracy improvements. *Cost* (used, e.g., in several works [26, 69, 74, 134]) measures the expenses due to acquisition or usage of computing resources for running the DSP system. For example, Hochreiner et al. [74] design an elastic DSP system for IoT scenarios, where the cost of used computing resources is minimized.

As mentioned earlier, several adaptation mechanisms are characterized by a (possibly significant) *adaptation overhead*, which is taken into account (e.g., [20, 50, 67, 132, 179, 185]). For instance, Fang et al. [50] present a load distribution strategy for stateful operators, where the overheads of state migration are taken into account when planning adaptation actions. Similarly, Borkowski et al. [20] design a solution to the operator scaling problem, where the "cost" of reconfigurations, in terms of overhead, is minimized.

Performance-related and accuracy-related metrics are sometimes combined to define custom *utility functions* (e.g., [88, 99, 100, 162, 163, 199]). For instance, we already mentioned [199], where tuple utility depends on real-time constraint satisfaction. Another example is given by Kumbhare et al. [100], who use a utility function to combines resource cost and a so-called application value that depends on current processing accuracy.

Among system-oriented metrics, the most used one is *resource utilization* (e.g., [20, 27, 57, 60, 68, 72, 91, 120, 134, 152, 165, 176]), which captures the utilization level of a computing resource, usually CPU. As in different application domains, utilization is often used in conjunction with threshold-based adaptation policies, where actions are triggered whenever the utilization level violates a pre-defined threshold value (e.g., [27, 57, 74]). A related metric is *load imbalance*, which measures the load difference among parallel instances of an operator. This metric is especially used by load distribution solutions (e.g., [3, 50, 92, 191]), which often aim at minimizing this metric, as better load balancing leads to better performance.

Another classic performance index used for adaptation (e.g., [6, 106, 176]) is *queue length*, which measures the amount of data stored in system buffers (e.g., operator input queue), ready to be processed. For instance, Li et al. [106] use operator queue length, along with resource utilization, as key metrics to trigger operator scaling actions by means of a threshold-based policy.

Several works also look at the usage of communication resources, measuring the *network usage* of operators (e.g., [21, 48, 110, 186, 191]), which is the amount of traffic they exchange with each other. This metric is particularly relevant for systems deployed in (geographically) distributed environments, where delay and bandwidth may severely impact performance. For example,

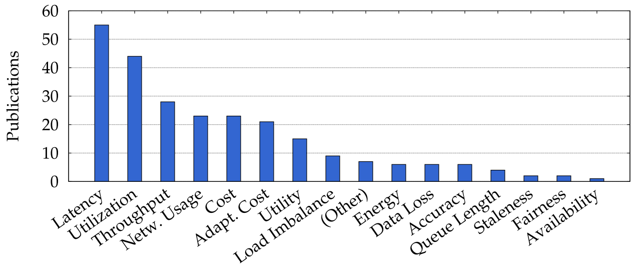


Fig. 8. Popularity of the different metrics among the reviewed publications.

Xu et al. [186] and Liu and Buyya [110], respectively, present T-Storm and D-Storm, which integrate traffic-aware solutions to the operator placement problem in Apache Storm.

Energy consumption has received growing interest over the past years as efforts toward sustainable computing have been promoted. A few works present adaptation strategies that exploit energy consumption as the key metric (e.g., [33, 43, 47, 166, 182]). For instance, Eibel et al. [47] and De Matteis and Mencagli [43] leverage dynamic voltage and frequency scaling to dynamically adjust the frequency of CPU cores where operators are executed, so as to trade off performance with energy consumption. Chao et al. [33] instead consider energy consumption while placing operators on mobile phones.

Other less popular metrics include availability and fairness. Application *availability* is taken into account by Chao and Stoleru [32] when placing operators on mobile phones with intermittent connectivity. Fairness is considered by Aljoby et al. [5] to allocate network capacity to different applications, and by Kalyvianaki et al. [90] to perform load shedding in a multi-tenant DSP system.

Figure 8 provides a graphical representation of the overall popularity of the aforementioned metrics. It is easy to realize that—as expected—few key metrics (i.e., latency, throughput, cost, utilization, adaptation overhead, network usage) are used far more frequently than the other ones.

4.3 Who: Controlling Authorities and Tenants

We now turn our attention toward the entities in charge of managing the adaptation process and the adapted applications. Specifically, we look at the *controlling authority*, which is responsible for making adaptation decisions, and the presence of *multiple tenants* within the DSP system, whose applications are possibly associated with different objectives and requirements.

4.3.1 Controlling Authority. Most approaches in the literature consider a *centralized* adaptation controller (e.g., [36, 48, 52, 55, 62, 82, 100, 117, 188]). In such a scheme, a single entity is responsible for the entire adaptation process. This centralized controller hence needs global information about the adapted applications and the underlying computing infrastructure, to make decisions about when and how adaptation actions should be performed. On the one hand, exploiting such a complete view of the system, centralized controllers are able to identify optimal adaptation policies (e.g., [55, 117]). On the other hand, scalability issues may arise when dealing with the complexity of the whole system, as the computational cost of the control algorithms may significantly increase with the number of involved applications, operators, and infrastructure elements. Moreover, from a fault tolerance perspective, a centralized controller represents a "single point of failure," whose faults inhibit adaptation capabilities for the whole system.

Decentralized control schemes (e.g., [90, 122, 131, 134, 138, 144]) overcome the scalability limitations of centralized approaches by distributing the adaptation responsibility to a multitude of controllers (e.g., a controller for each operator), which plan adaptation actions based on local, usually partial, information about the system. However, such a limited system view makes often difficult (or even impossible) for them to identify optimal adaptation policies, although some works (e.g., [122, 144]) propose optimal decentralized strategies. For instance, Rizou et al. [144] present a solution to the placement problem where each operator optimizes its own deployment; exploiting mathematical properties of the objective function, they are still able to identify the global optimum. Mencagli [122] relies on game theory to devise a distributed control strategy for operator scaling, where each operator makes decisions about its own parallelism level.

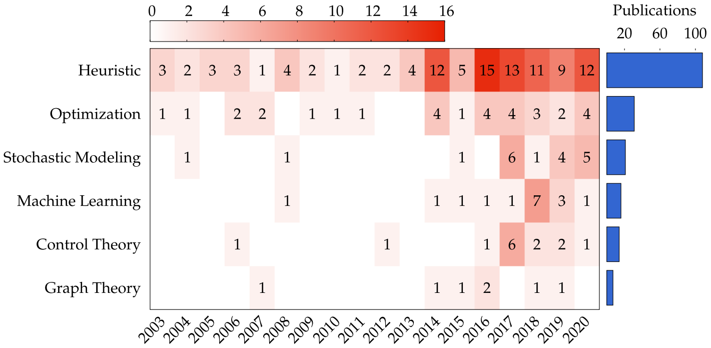
It is also worth noting that the choice of the control architecture to adopt is also influenced by the adaptation mechanisms that must be supported. Indeed, although it is particularly difficult to devise optimal decentralized strategies such as the operator placement or scaling problems, for other mechanisms, such as load shedding and load distribution, it is less critical to have a global system view, as they usually work at level of single buffers or operators.

A few works (e.g., [1, 6, 25, 43, 56, 59]) investigate *hierarchical* (or *hybrid*) control schemes, which are neither centralized nor fully decentralized. Having multiple controllers, often organized in a hierarchical fashion, it is possible to increase the scalability with respect to a single centralized controller while avoiding the lack of coordination of fully decentralized schemes. In particular, controllers at different layers of the hierarchy usually work with different time scales and granularity of control (e.g., components at the top of the hierarchy may rely on a coarse-grained view of the system, with lower layers taking care of finer adaptation control). For example, Amini et al. [6] present a two-layer vertical operator scaling and backpressure strategy: the first layer employs global optimization to compute and communicate resource allocation targets to resource controllers instantiated on each processing node; the second layer uses these allocation targets, along with local monitoring information, to inform upstream operators of the desired input rate. Similarly, Cardellini et al. [25] rely on a two-layered hierarchy to control operator scaling: at the top layer, controllers manage single DSP applications by coordinating decentralized controllers, which make adaptation decisions for single operators. Abadi et al. [1] consider a three-layered hierarchy: at the operator level, a local controller is responsible for load shedding; a neighborhood controller is responsible for load balancing the resources at a node with those of its immediate neighbors; and at the highest level, a global controller is responsible for making global optimization decisions and sending proper instructions (e.g., regarding the amount of load shedding) to lower-level controllers.

4.3.2 Multi-Tenancy. A DSP system may host multiple applications running concurrently. Hosted applications in turn may fall under the responsibility of either a single authority or a multitude of *tenants*. The former scenario is especially popular for DSP systems deployed in on-premise computing infrastructures, whereas the latter is commonly found, for example, in DSP platforms offered as Cloud Software-as-a-Service products. The majority of the reviewed works consider *single tenants*, often focusing on a single running application in their adaptation strategies. A few works tackle more complex *multi-tenancy* scenarios (e.g., [78, 88, 90, 136]). For instance, Kalim et al. [88] consider a DSP system with multiple applications and tenants, where operator scaling is used to dynamically allocate resources to the applications so as to satisfy all of their SLOs. Pham et al. [136] instead introduce differentiated *classes of service* in a DSP system, so as to accommodate the needs of multiple applications by means of load shedding and adaptive resource allocation.

4.4 How: Planning Methodology

The *planning methodology* identifies the techniques used to determine adaptation policies, which is to make adaptation decisions. Several approaches have been exploited so far in the literature. We will group them into the following classes: mathematical optimization and game theory, control



	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020
Heuristic	3	2	3	3	1	4	2	1	2	2	4	12	5	15	13	11	9	12
Optimization	1	1		2	2		1	1	1			4	1	4	4	3	2	4
Stochastic Modeling		1				1							1		6	1	4	5
Machine Learning						1						1	1	1	1	7	3	1
Control Theory				1						1				1	6	2	2	1
Graph Theory					1							1	1	2		1	1	

Fig. 9. Popularity of the different methodologies for adaptation control per year.

theory, graph theory, stochastic modeling (and, in particular, queueing theory), heuristics, and ML. Note that here we use the term *heuristic* in a broad sense, referring both to implementations of metaheuristics for adaptation optimization and custom algorithms (e.g., rule-based and, in particular, threshold-based scaling policies), which do not fall into any of the other categories. As such, this group embraces a large number of works, as demonstrated by Figure 9, where the popularity of the different methodologies over time is depicted. We also observe that besides resulting the most used techniques overall, optimization and heuristics were almost the only popular options in the past. More recently, other methodologies have been increasingly considered, especially control theory, ML, and stochastic modeling.

As we explained in Section 4.2, the motivations that lie behind adaptation can usually be expressed as an optimization problem, with one or more key metrics as objectives, and possibly constraints to be satisfied. Therefore, the most natural way to derive adaptation policies hinges on the modeling and resolution of the underlying optimization problem, by means of *mathematical optimization* tools. This approach is exploited, for example, to control various adaptation actions, such as operator placement (e.g., [117, 132, 144]) and scaling (e.g., [62, 113]), load distribution (e.g., [155, 191]), or micro-batch size tuning (e.g., [41]). For example, Madsen et al. [117] rely on a mixed-integer linear program formulation of the operator placement problem, where the objective to be minimized is a load imbalance function, with a maximum migration overhead constraint. Lohrmann et al. [113] instead focus on the operator scaling problem and cope with a nonlinear formulation, where they aim at minimizing the amount of allocated resources subject to a maximum latency constraint. The resulting problem is solved by means of gradient descent. *Robust optimization*, a field of optimization that deals with uncertainty in the data of optimization problems, has been so far only exploited by Lei and Rundensteiner [105] to design a load distribution approach that is resilient to workload fluctuations at runtime and hence can avoid operator migration overheads.

Given the multitude of entities usually involved in the adaptation process (e.g., operators, different applications), *game theory* represents a promising tool for the analysis of their interactions. Game theory hinges on the notion of game equilibria (their existence and possibly uniqueness) and how far the equilibria solutions are from the optimum (*price of anarchy*). However, the set of tools in this field has been so far scarcely adopted to self-adapt DSP applications. Mencagli [122] investigates this technique to drive DSP operator scaling in a decentralized fashion, where each operator is an agent that chooses its own parallelism level. A non-cooperative scenario is first considered, where agents are shown to reach the best equilibrium in a Pareto sense. Then, a cooperative scenario is studied, where an *incentive-based* mechanism is used to promote cooperation among agents, so as to get closer to system optimum. Balazinska et al. [13] present an approach based on distributed algorithmic mechanism design for managing load in federated DSP systems. It is based on private pairwise contracts pre-negotiated between participants, which set bounded prices for migrating load and specify the set of operators that each participant is willing to execute on behalf of others.

A few works exploit methods from *control theory* to devise adaptation policies. In this case, three main entities are identified: disturbance, decision variables, and system configuration. *Disturbances* represent dynamics that cannot be controlled, even though their future value can be predicted (at least in the short term), whereas *decision variables* map to the adaptation actions. Control-theoretic approaches are used in conjunction with a variety of adaptation mechanisms: operator scaling (e.g., [20, 44, 78, 123]), load distribution [123, 124], backpressure [37], and load shedding [89]. For instance, Mencagli et al. [123] rely on both *PID* controllers and *fuzzy logic* in their two-layered adaptive DSP solution. PID controllers regulate load distribution among parallel operator instances, whereas fuzzy logic controls scaling actions on longer time scales. Kalyvianaki et al. [89] instead design a discrete-time control algorithm for load shedding, which at each timestep selects the number of tuples to be processed so as to keep processing latency within a pre-defined value.

Being DSP applications usually modeled as DAGs, it is not surprising that *graph theory* has also been used to devise adaptation policies. In particular, it has been applied to drive operator scaling [87], operator placement [48, 132], and load distribution [23]. For example, Eskandari et al. [48] present a solution to the operator placement problem based on two-phase graph partitioning. In the first phase, they partition the application graph to decide which operators should be placed in the same computing node. Then, a second partitioning is used to assign operators to processes within each node, so as to minimize inter-node and inter-process communication. Ottenwälder et al. [132] deal with the placement problem as well and exploit a *time-graph* to model the migration plans associated with possible placement solutions for each operator. Based on this time-graph, they identify the best placement (and, hence, migration plan) solving a shortest path problem.

A few works exploit *stochastic modeling* tools to devise adaptation policies (e.g., [80, 82, 146, 163]). For instance, Slo et al. [163] present a load shedding solution for CEP systems, where shedding is driven by a probabilistic model. They aim at minimizing the impact of dropped events on accuracy while keeping latency within a defined bound. Imai et al. [82] and Runsewe and Samaan [146] both propose infrastructure scaling strategies for DSP systems running in the Cloud, which rely on predictions of future workloads. In the former work, an *ARMA* model is used for workload forecasting; in the latter, a *layered multi-dimensional hidden Markov model* is used, where the lower layer predicts resource utilization of single applications, and the top layer predicts the overall system load based on that information.

A particular class of models that is widely used for performance management is *queueing theory*, which has also been applied to DSP systems adaptation (e.g., [55, 76, 113, 147, 172, 180]). For example, to tackle the operator scaling problem, Fu et al. [55] model DSP applications as queueing networks, where each operator is associated with a GI/G/k station. The resulting model is used to estimate application latency and allocate resources accordingly. A similar approach is used by Lohrmann et al. [113], where, however, each operator instance is modeled as a GI/G/1 station, and Kingman's approximation [19] is exploited to estimate latency. Wang et al. [180] instead consider the operator placement problem, again modeling operators as GI/G/k stations. They use the resulting model to predict both application latency and throughput, resorting to the Allen-Cunneen formula for latency approximation [19]. Russo Russo et al. [147] present a vertical operator scaling solution that leverages **Markovian arrival processes (MAP)** [19] for online characterization of bursty workloads and the analytical resolution of the associated MAP/MAP/1 queueing models.

Most of the reviewed works rely on *heuristic* algorithms—in the broad sense explained earlier—to plan adaptation (e.g., [2, 7, 36, 57, 59, 62, 64, 102]). For example, Chatzistergiou and Viglas [36] present fast, linear-time heuristics for the operator placement problem, where they aim at minimizing inter-node traffic. The placement problem is also considered by Lakshmanan and Strom [102], whose goal is to minimize the end-to-end latency through an *ant colony* based heuristic approach. *Greedy* heuristics have been frequently exploited to drive adaptation (e.g., [7, 62–64]). For instance, Guo and Zhou [62] tackle the joint optimization of operator scaling and placement. Given the complexity of the resulting formulation, they resort to greedy resolution algorithms.

Among heuristic approaches, several works have investigated *threshold-based* algorithms (e.g., [27, 57, 60, 79, 95, 106, 141]), where adaptation actions are triggered whenever a certain metric becomes higher (or lower) than a pre-defined threshold. For example, Gulisano et al. [60] consider the operator scaling problem; they trigger scale-out actions whenever resource utilization exceeds a high utilization threshold, and scale-in actions when utilization is lower than a low utilization threshold. Kleiminger et al. [95] instead tackle the problem of distributing load between a local stream processor and the Cloud; to switch between local and remote processing, they monitor the operator input queue length, triggering adaptation when a maximum size threshold is exceeded. Ravindra et al. [141] consider a similar environment, where the DSP system spans a hybrid public-private Cloud. To trigger the allocation of new computing nodes and the switch between private and public Cloud deployment, they rely on a threshold defined in terms of maximum latency. Huang and Lee [79] present an adaptive fault tolerance solution, which relies on three user-configurable threshold parameters: (i) the maximum divergence between the current state and the most recent backup state, (ii) the maximum number of unprocessed non-backup items, and (iii) the maximum number of not-yet-acknowledged items.

The ever-increasing popularity of ML methods has not been ignored by the DSP community, and a few works have investigated the applicability of ML techniques to DSP adaptation (e.g., [82, 98, 115, 126, 190, 196]), focusing on different methodologies and adaptation mechanisms. For instance, Kombi et al. [98] exploit regression techniques to forecast the operator input rate and adjust its parallelism accordingly. To predict workload and resource utilization in the near future, Lombardi et al. [115] rely on artificial neural networks; these predictions are then used as the input for a threshold-based operator scaling policy. Isotonic regression is applied by Zhang et al. [196] to estimate the impact of different batch sizes in a micro-batched DSP system, and dynamically adjust the configuration based on the workload and operating conditions.

A branch of ML of particular interest is **reinforcement learning (RL)**, a collection of techniques to learn optimal behaviors in stochastic environments with respect to a set of available actions and associated rewards. RL has been applied to drive DSP adaptation in several works (e.g., [8, 25, 38, 68, 108, 115, 148, 160]). Heinze et al. [68] use RL to drive operator scaling considering a reward function based on operator resource utilization: the closer to a pre-defined target value, the higher the reward for the agent. Similarly, Cardellini et al. [25] use RL for operator scaling and specifically investigate model-based RL algorithms, which allow for significant reduction of the training phases. Operator scaling is also considered by Lombardi et al. [115], where RL is used to learn the optimal utilization thresholds for their policy. Cheng et al. [38] instead leverage RL to allocate resources to different jobs and tasks in a batched DSP system, using a performance-based reward function to discriminate good and bad choices. Li et al. [108] consider the operator placement problem, aiming to minimize the end-to-end latency. Dealing with very large state spaces, they exploit deep neural networks in conjunction with RL (*deep* RL). Silva Veith et al. [160] also tackle the placement problem using RL techniques; in particular, they consider a multi-objective formulation and exploit various algorithms, including *Monte Carlo Tree Search*, for its resolution.

4.5 When: Triggers and Time Horizon

In this section, we look at the aspects of adaptation related to time, which is *when* adaptation actions should be performed, and which time horizon should be used for planning.

4.5.1 Trigger. Adaptation actions can either be triggered by *periodic timers* or in response to particular *events*. Timer-based adaptation is simpler to design and implement, as it requires to set a single parameter (i.e., the adaptation activation interval) and guarantees that the adaptation policy keeps planning any required action over time. The interval between consecutive adaptation rounds usually ranges between few seconds and several minutes. Clearly, this interval should be short enough to allow the adaptation policy to quickly respond to condition changes; however, too frequent metrics collection and adaptation planning introduce overheads, so the activation interval must be set based on a trade-off between responsiveness and efficiency. Timer-based adaptation is adopted by most existing approaches (e.g., [1, 14, 53, 87, 100, 109, 152, 176]). For instance, Floratou et al. [53] and Kalavri et al. [87] present operator scaling approaches where the system periodically collects the required metrics from operators (e.g., throughput) and invokes an adaptation policy.

Other works (e.g., [2, 3, 35, 37, 90, 199]) consider event-triggered adaptation actions. To implement this kind of scheme, one or more types of event must be associated with the execution of an adaptation policy. In the literature, the most used event for this purpose is the arrival of a new tuple to a buffer, which is associated with load shedding, load distribution, and stream scheduling strategies (e.g., [2, 90, 92, 199]). For instance, Katsipoulakis et al. [92] present a load distribution strategy that is triggered every time an incoming tuple is detected. Chaturvedi et al. [35] instead present an operator reuse strategy where adaptation is triggered whenever a new application is submitted for execution or one is terminated. Differently from these works, Ottenwälder et al. [132] consider user-related triggering events. Indeed, in their geo-distributed CEP placement solution, operator migrations are planned and possibly executed in response to users' location changes.

4.5.2 Proactivity. A second important issue is the *time horizon* considered to plan adaptation actions. Specifically, *reactive* approaches look at information from the past to make adaptation decisions, thus possibly reacting to condition changes. Conversely, *proactive* strategies make decisions looking at a limited future time horizon, to adapt applications in advance. It is clear that proactive solutions are harder to realize, as they require predictions about future working conditions, and their efficacy depends on the accuracy of such predictions. It is therefore no surprising that most of the existing approaches rely on reactive adaptation schemes (e.g., [3, 30, 57, 60, 118, 124, 154, 155, 165, 195]). Examples of reactive policies are given by the aforementioned threshold-based heuristics, where actions are triggered by threshold violations, which are usually evaluated against latest available monitoring information (e.g., average resource utilization in the last minute).

A few works propose proactive adaptation strategies (e.g., [21, 43, 72, 78, 80, 82, 98, 99, 146, 190]), relying on several different techniques. A key challenge in designing proactive adaptation solutions is forecasting the application load in the near future, especially for operator and infrastructure scaling strategies. Zacheilas et al. [190] deal with this issue exploiting regression methods based on Gaussian processes for workload prediction; Imai et al. [82] and Hoseiny Farahabady et al. [78] rely on, respectively, ARMA and ARIMA forecasting models; Hidalgo et al. [72] model the incoming load using a Markov chain; a more complex state-based method is used by Runsewe and Samaan [146], where multi-layer hidden Markov models are considered. Buddhika et al. [21] instead design a custom data structure, called *prediction ring*, to track data stream arrivals and predict resource utilization. Prediction rings are similar to circular buffers, where exponential smoothing is used to update arrival rate estimates over time. The data structure is also used to compute an

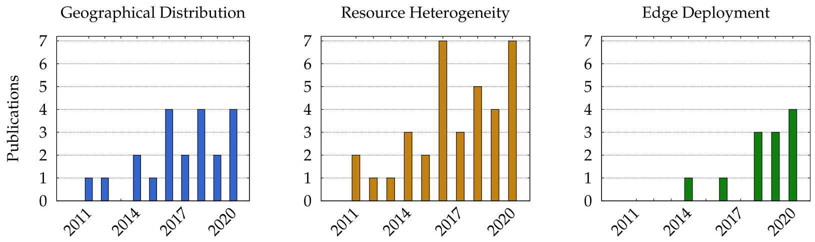


Fig. 10. Publications dealing with geographical distribution, resource heterogeneity, and Edge-based deployments throughout the past decade.

interference score that quantifies the impact of placing an additional operator instance alongside other instances on a given machine. Besides prediction, another issue is related to proactive control and optimization of adaptation actions. Some works (e.g., [43, 77]) exploit *model predictive control*, a control-theoretic approach that uses a model to predict the future system behavior over a limited prediction horizon. For instance, De Matteis and Mencagli [43] use model predictive control to control operator scaling and optimize a multi-objective cost function, which accounts for QoS violations, resource usage, and adaptation overhead. Similarly, Kumbhare et al. [99] propose a *lookahead optimization* approach, where a prediction model is used to solve an optimization problem over a sliding future time window, and accordingly control auto-scaling. They consider a utility maximization problem, with a constraint on the minimum application throughput to be guaranteed.

4.6 Where: Computing Environment

The last question we analyze is related to *where* adaptation is implemented. DSP systems are deployed in a variety of different environments, including parallel multi-core servers, Cloud infrastructures, and Fog/Edge platforms. As these computing environments exhibit very different characteristics, adaptation strategies usually make some assumptions about the environment they target. In particular, we characterize target environments looking at their degree of system distribution, the heterogeneity of the offered computing resources, the inclusion of nodes located at the Edge, and the availability of specialized hardware. It is clear that these aspects are partially correlated (e.g., solutions targeting Edge platforms are more likely to consider geographically distributed deployments and heterogeneous nodes). In recent years, increasing attention has been devoted to Fog/Edge platforms and geo-distributed settings in general, not only in the DSP domain. This trend is confirmed by our literature review, which shows a growing number of adaptation strategies dealing with this kind of environments as well as resource heterogeneity (Figure 10).

4.6.1 System Distribution. We first look at the degree of distribution of the computing environment. Some works target *single-machine* environments, exploiting the parallelism provided by multi-core and multi-processor architectures (e.g., [56, 86, 123, 124, 154, 156]). In these environments, the most investigated issues are operator scaling and stream scheduling, which enable efficient utilization of the hardware. For instance, Kahveci and Gedik [86] present *Joker*, a DSP runtime that is able to automatically scale the execution of Java-based multi-threaded DSP applications. Schneider and Wu [156] tackle the same problem for applications running on top of IBM Streams, presenting a solution to fully exploit the parallelism provided by machines with hundreds of cores. De Matteis and Mencagli [44] also target multi-threaded scenarios, additionally considering energy consumption in their resource allocation policy. Conversely, Fu et al. [56] consider the problem of scheduling the execution of operators in a time-sharing fashion on resource-constrained Edge nodes, where the number of available cores is likely smaller than needed.

Most of the reviewed works target (locally) distributed computing environments, where DSP systems can scale their execution on several nodes (e.g., [7, 20, 27, 48, 50, 67, 93, 109, 120, 196]). In addition to operator scaling, which is widely adopted also in single-machine environments, operator placement is particularly relevant for distributed DSP systems, where potential increases in computing capacity come at the cost of inter-node communication, whose performance impact might be significant. For instance, Eskandari et al. [48] and Wu et al. [184] propose placement solutions aiming to minimize the network traffic produced by operator instances, as well as Huang et al. [80], who focus on load distribution. Distributed infrastructures, especially those in the Cloud, often provide the additional benefit of being elastic (i.e., computing nodes can be provisioned as needed at runtime). Indeed, infrastructure-level scaling is investigated, for example, by Marangozova-Martin et al. [120] and Borkowski et al. [20], where it is coupled with operator scaling.

A few works (e.g., [30, 131, 132, 138, 144, 153, 175, 181, 193]) have investigated adaptation for DSP systems in geographically distributed environments, where network-related aspects such as delay, unreliability, and limited bandwidth play a key role. For this reason, operator placement is even more relevant in this context, as operator instances must be carefully assigned to computing nodes taking into account network aspects. This problem is tackled, for example, by Rizou et al. [144], where a distributed optimization algorithm is used to find a placement solution that minimizes the amount of data sent over network. Pietzuch et al. [138] also present a decentralized solution for the placement problem exploiting a physics-inspired model, which enables minimization of the network usage. Saurez et al. [153] specifically target Fog computing scenarios, and devise a runtime placement adaptation strategy to migrate operators and satisfy latency requirements.

4.6.2 Resource Heterogeneity. For distributed infrastructures, a key question is whether computing nodes are assumed to be *homogeneous* (i.e., they have identical technical specifications) or *heterogeneous.* So far, most of the solutions have targeted homogeneous environments, but a few works consider more challenging, heterogeneous settings (e.g., [46, 64, 91, 100, 131, 132, 148, 155, 180]). Among the various mechanisms, operator placement is particularly impacted by heterogeneity. For instance, Kalyvianaki et al. [91] present a placement model in which computing nodes can be equipped with different amounts of resources (e.g., CPU cores). Heterogeneous nodes also call for proper load distribution strategies, as instances deployed on more powerful nodes are expected to process more data. To this end, Du and Gupta [46] investigate a latency-based load distribution scheme for heterogeneous platforms, where load imbalance among operator instances is measured in terms of processing latency. Conversely, Schneider et al. [155] aim to minimize the time TCP connections between operators are blocked because of full buffers, which happens when an operator is overloaded. Kumbhare et al. [100] consider a Cloud DSP system with heterogeneous VMs and study an operator and infrastructure scaling strategy. To make decisions about the VMs to allocate and deallocate when needed, they associate each VM with a *weight*, which accounts for the amount of resources available on that VM, its cost, and the remaining time in the current billing cycle.

4.6.3 Edge Deployment. The idea of deploying DSP applications in Edge platforms is attractive to avoid moving user-generated data toward data centers, thus reducing latency and bandwidth consumption. There is a limited number of adaptation solutions dealing with Edge computing environments (e.g., [8, 32, 56, 131, 160, 193]), but their number has been growing over the past couple of years. For instance, O'Keeffe et al. [131] maximize the throughput achieved by applications running on top of IoT devices at the Edge, exploiting stream routing techniques inspired by backpressure routing in data networks. Aral et al. [8] consider a specific class of streaming applications, where incoming data streams are used to train ML models. They envision a distributed learning architecture, where local models are trained at the Edge and periodically sent to the Cloud, where they are aggregated and broadcasted. Their adaptive solution dynamically adjusts the frequency of updates from Edge to Cloud so as to keep model staleness under control.

4.6.4 Specific Hardware. Although most of the existing DSP systems are designed to run on commodity hardware, there is an increasing interest for *hardware-conscious* streaming systems, which can be optimized to fully exploit specific hardware architectures. In addition, although the number of DSP systems dealing with hardware other than general-purpose CPUs is large and growing, as surveyed by Zhang et al. [197], few of them involve runtime adaptation. Among the solutions reviewed in this work, several target specific hardware [5, 44, 47, 97]. For instance, Koliousis et al. [97] present SABER, a hybrid relational DSP engine for CPUs and GPGPUs, which dynamically schedules operators to the most suitable processor based on online observations. Aljoby et al. [5] instead target specific hardware at the network level, assuming the availability of SDN-enabled switches.

5 CHALLENGES AND RESEARCH PERSPECTIVES

Our review shows that a lot of ground has been covered on adaptive DSP. However, there are still areas in which we expect more research to be carried out in the next years, also based on the trends we highlighted earlier. In this section, we briefly outline the main open challenges and future directions we envision for this field.

5.1 Multiple Adaptation Mechanisms

Our classification shows that more than 70% of the considered solutions focus on a single adaptation mechanism, applied in isolation, and 90% of them consider no more than two mechanisms. The mechanisms most frequently exploited in conjunction are (i) operator placement and scaling, to optimize both the number and the deployment of operator instances (e.g., [26, 62, 182]), and (ii) operator and infrastructure scaling, to elastically provision computing resources based on application parallelism (e.g., [3, 20, 172]). To provide more general solutions, more effort must be spent to tackle the challenges of multi-mechanism adaptation. On the one hand, joint adaptation requires careful investigation of the possible interactions between different mechanisms. On the other hand, the problem of adaptation planning, which is often complex with a single mechanism, becomes even more challenging when multiple mechanisms must be jointly controlled.

Furthermore, our analysis also shows that some mechanisms have received much less attention that others, thus much is still to be done for a complete exploration of these tools. A few mechanisms, such as query replanning or configuration tuning, have been thoroughly investigated for static application optimization, but their adoption at runtime is still limited. Other mechanisms are likely to receive new (or renewed) interest thanks to technology advancements that enable their efficient adoption (e.g., widespread use of software containers will foster the adoption of vertical operator scaling techniques; the increasing support for SDN will boost the exploration of network-level adaptation). Similarly, we expect hardware-specific adaptation mechanisms like adaptive query compilation [97] to gain popularity, given the increasing availability of DSP systems able to efficiently exploit specialized computing platforms such as GPGPUs and FPGAs [197].

5.2 Integrated Support for Adaptation

Introducing adaptation into existing DSP frameworks often requires coping with inefficient support for application reconfiguration (e.g., [16, 75, 125, 137, 159, 179]), as most the frameworks have been designed with performance, ease of programmability, and fault tolerance as primary objectives. As DSP systems run in increasingly dynamic environments, we expect a shift in the role of application adaptability, which will become a key pillar in the design of future systems.

On the one hand, we expect DSP system architectures to become more decentralized, especially with regard to the control plane. By doing so, systems will gain flexibility and scalability, which will enable fine-grained control of the system components, from operator instances to buffers and network resources. Some effort has already spent in this aim, such as effort by Mai et al. [119], who proposed a programmable control plane for distributed DSP that enables scalable reconfiguration. On the other hand, as adaptation becomes a first-class citizen, we expect QoS objectives, which are usually provided as configuration parameters for adaptation, to become primary entities in the definition of DSP applications. Therefore, we expect extended programming models to enable the definition of parameters such as the minimum required throughput, availability, consistency, or the allowed inaccuracy at different stages of processing, along with the application topology.

5.3 Edge/Fog and Mobile Computing Platforms

As data streams often originate from devices at network edges (e.g., IoT devices), many applications would benefit from running closer to their sources, exploiting Fog and Edge computing environments. However, the most popular frameworks (e.g., Flink, Storm) are equipped with data processing layers designed for large and locally distributed server-based platforms. To effectively support real-time analytics at the edge of the network, this layer should be redesigned for constrained devices (e.g., with limited energy, or processing capacity), calling for specific lightweight frameworks. Although some effort has been recently spent in this aim (e.g., [56, 183, 187, 193, 194]), we expect this direction to be further explored as edge-oriented DSP frameworks are still far from the solidity of the established alternatives.

Application deployment also requires more attention in Edge/Fog environments, because of the higher heterogeneity and the more evident impact of network due to larger delays and limited bandwidth. Indeed, network-aware Edge placement strategies for DSP have been proposed both for offline (e.g., [140, 150]) and online (e.g., [32, 153, 160]) deployment optimization. Nevertheless, a significant gap exists between the richness of adaptation solutions for Cloud-based DSP systems and the set of those applicable to Fog/Edge platforms. For instance, although scaling-out is the most widely adopted mechanism to improve performance over Cloud-class resources, the resource scarcity of Edge devices calls for different resource acquisition strategies.

Moreover, moving DSP applications at the Edge possibly means dealing with *mobile* sources and *mobile* processing nodes, which can have a disruptive impact on application functionality. Initial effort has been spent to tackle the challenges associated with mobile DSP (e.g., energy consumption and unreliable network connectivity) (e.g., [32, 131, 170]), but there are still several issues to be addressed (e.g., smooth operator migration and specific fault-tolerance mechanisms).

5.4 Serverless DSP

Serverless computing [29] is increasingly popular thanks to the scalability and flexibility it promises, as well as its attractive pricing models. Researchers have started to investigate the use of serverless for data analytics (e.g., [129]), with the aim of relieving users from several operational concerns. However, in terms of performance, serverless computing environments have a few limitations that prevent seamless adoption for DSP. First, the stateless and ephemeral nature of serverless functions (i) forces operator internal state to be stored externally with possible overheads, and (ii) prevents operators to directly exchange data streams, having to resort to data stores for in-transit data. Although researchers have started addressing these issues (e.g., [96]), the major serverless platforms still suffer from these limitations. Furthermore, for real-time data analytics, another important issue is related to the *cold start* phenomenon [29], which can lead to latency spikes.

As research on serverless platforms continues, these limitations will be likely mitigated or removed. Nevertheless, DSP systems will have to be adapted or even redesigned to fully exploit serverless environments. Runtime application adaptation will require some effort as well, as the mechanisms and policies used so far will not necessarily fit the new processing paradigm.

5.5 Security Guarantees

DSP applications often cope with privacy-sensitive information or perform analytics tasks that may trigger safety-critical operations (e.g., anomaly detection in a manufacturing system). In either case, stream integrity and confidentiality must be guaranteed to avoid unintended (and possibly dangerous) behaviors. So far, security and privacy for DSP have received less attention compared to other aspects. Most of the related effort has been devoted to *access control* mechanisms for data streams (e.g., [28]) and *privacy preservation* techniques (e.g., [103]). A few works exploit specialized hardware features for increased security. For instance, Havet et al. [65] propose *SecureStreams*, combining a high-level dataflow programming model with low-level Intel *Software Guard Extensions* (SGX) to guarantee stream privacy and integrity. Park et al. [135] consider analytics on untrusted, resource-constrained Edge devices and present StreamBox-TZ, which offers strong data security and verifiable results by isolating computation in ARM-based *Trusted Execution Environments*. Differently, Chaturvedi and Simmhan [34] apply *Moving Target Defense*, where the key idea is varying system configuration (e.g., used port numbers, application topology) at runtime so that any prior information available to attackers becomes hardily usable.

Security and privacy aspects must be also considered when deploying the applications over distributed infrastructures. So far, security-related concerns have been mostly neglected by the literature on DSP application placement, with few exceptions (e.g., [149, 153]). We expect security aspects to be increasingly included in runtime adaptation solutions.

6 SUMMARY

We reviewed the existing approaches for runtime adaptation of DSP applications and systems. Relying on the “5W1H” approach, we presented a taxonomy of the most relevant solutions, which allowed us to identify past and present trends within this research area. A complementary taxonomy focused on the implementation and experimental evaluation of the solutions is available in Appendix B. Although a significant amount of work has been carried out on the topic, we identified a few gaps that still exist in the literature, especially with regard to recent trends (e.g., Fog/Edge-based application deployment). Based on these observations, we outlined some research directions that we expect to be pursued in the near future, to enhance current DSP systems and develop new ones.

REFERENCES

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, et al. 2005. The design of the Borealis stream processing engine. In *Proc. of CIDR’05*. 277–289.

[2] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: A new model and architecture for data stream management. *VLDB J.* 12, 2 (2003), 120–139.

[3] Ahmed S. Abdelhamid, Ahmed R. Mahmood, Anas Daghistani, and Walid G. Aref. 2020. Prompt: Dynamic data-partitioning for distributed micro-batch stream processing systems. In *Proc. of ACM SIGMOD’20*. ACM, New York, NY, 2455–2469.

[4] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, et al. 2015. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.* 8, 12 (2015), 1792–1803.

[5] Walid A. Y. Aljoby, Xin Wang, Tom Z. J. Fu, and Richard T. B. Ma. 2019. On SDN-enabled online and dynamic bandwidth allocation for stream analytics. *IEEE J. Sel. Areas Commun.* 37, 8 (2019), 1688–1702.

[6] Lisa Amini, Navendu Jain, Anshul Sehgal, Jeremy Silber, and Olivier Verscheure. 2006. Adaptive control of extreme-scale stream processing systems. In *Proc. of IEEE ICDCS’06*.

[7] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. 2013. Adaptive online scheduling in storm. In *Proc. of ACM DEBS'13*. 207–218.

[8] Atakan Aral, Melike Erol-Kantarci, and Ivona Brandic. 2020. Staleness control for edge data analytics. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 2 (2020), Article 38, 24 pages.

[9] Marcos D. de Assunção, Alexandre da Silva Veith, and Rajkumar Buyya. 2018. Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. *J. Netw. Comput. Appl.* 103 (2018), 1–17.

[10] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. 2002. Models and issues in data stream systems. In *Proc. of ACM PODS'02*. 1–16.

[11] Brian Babcock, Mayur Datar, and Rajeev Motwani. 2004. Load shedding for aggregation queries over data streams. In *Proc. of ICDE'04*. IEEE, Los Alamitos, CA, 350–361.

[12] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. 2008. Fault-tolerance in the Borealis distributed stream processing system. *ACM Trans. Database Syst.* 33, 1 (2008), Article 3, 44 pages.

[13] Magdalena Balazinska, Hari Balakrishnan, and Mike Stonebraker. 2004. Contract-based load management in federated distributed systems. In *Proc. of USENIX NSDI'04*.

[14] Cagri Balkesen, Nesime Tatbul, and M. Tamer Özsu. 2013. Adaptive input admission and management for parallel stream processing. In *Proc. of ACM DEBS'13*. 15–26.

[15] Edmon Begoli, Tyler Akidau, Slava Chernyak, Fabian Hueske, Kathryn Knight, Kenneth Knowles, Daniel Mills, and Dan Sotolongo. 2021. Watermarks in stream processing systems: Semantics and comparative analysis of Apache Flink and Google Cloud dataflow. *Proc. VLDB Endow.* 14, 12 (2021), 3135–3147.

[16] Mehdi M. Belkhiria, Marin Bertier, and Cédric Tedeschi. 2020. Group mutual exclusion to scale distributed stream processing pipelines. In *Proc. of IEEE/ACM UCC'20*. 247–256.

[17] Paolo Bellavista, Antonio Corradi, Spyros Kotoulas, and Andrea Reale. 2014. Adaptive fault-tolerance for dynamic resource provisioning in distributed stream processing systems. In *Proc. of EDBT'14*. 85–96.

[18] Paolo Bellavista, Antonio Corradi, Andrea Reale, and Nicola Ticca. 2014. Priority-based resource scheduling in distributed stream processing systems for big data applications. In *Proc. of IEEE/ACM UCC'14*. 363–370.

[19] Gunter Bolch, Stefan Greiner, Hermann de Meer, and Kishor S. Trivedi. 2006. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications* (2nd ed.). Wiley.

[20] Michael Borkowski, Christoph Hochreiner, and Stefan Schulte. 2019. Minimizing cost by reducing scaling operations in distributed stream processing. *Proc. VLDB Endow.* 12, 7 (2019), 724–737.

[21] Thilina Buddhika, Ryan Stern, Kira Lindburg, Kathleen Ericson, and Shrideep Pallickara. 2017. Online scheduling and interference alleviation for low-latency, high-throughput processing of data streams. *IEEE Trans. Parallel Distrib. Syst.* 28, 12 (2017), 3553–3569.

[22] Michael Cammert, Jurgen Kramer, Bernhard Seeger, and Sonny Vaupel. 2008. A cost-based approach to adaptive resource management in data stream systems. *IEEE Trans. Knowl. Data Eng.* 20, 2 (2008), 230–245.

[23] Matthieu Caneill, Ahmed El-Rheddane, Vincent Leroy, and Noël De Palma. 2016. Locality-aware routing in stateful streaming applications. In *Proc. of ACM/IFIP/USENIX MIDDLEWARE'16*. ACM, New York, NY, Article 4, 13 pages.

[24] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State management in Apache Flink®: Consistent stateful distributed stream processing. *Proc. VLDB Endow.* 10, 12 (2017), 1718–1729.

[25] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. 2018. Decentralized self-adaptation for elastic data stream processing. *Future Gener. Comput. Syst.* 87 (2018), 171–185.

[26] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. 2018. Optimal operator deployment and replication for elastic distributed data stream processing. *Concurr. Comp. Pract. Exp.* 30, 9 (2018).

[27] Valeria Cardellini, Matteo Nardelli, and Dario Luzi. 2016. Elastic stateful stream processing in storm. In *Proc. of HPCS'16*. IEEE, Los Alamitos, CA, 583–590.

[28] Barbara Carminati, Elena Ferrari, Jianneng Cao, and Kian Lee Tan. 2010. A framework to enforce access control over data streams. *ACM Trans. Inf. Syst. Secur.* 13, 3 (2010), Article 28, 31 pages.

[29] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2019. The rise of serverless computing. *Commun. ACM* 62, 12 (2019), 44–54.

[30] Javier Cerviño, Evangelia Kalyvianaki, Joaquín Salvachúa, and Peter R. Pietzuch. 2012. Adaptive provisioning of stream processing systems in the cloud. In *Proc. of IEEE ICDE'12*. 295–301.

[31] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, and Fred Reiss. 2003. TelegraphCQ: Continuous dataflow processing. In *Proc. of ACM SIGMOD'03*. 668.

[32] Mengyuan Chao and Radu Stoleru. 2020. R-MStorm: A resilient mobile stream processing system for dynamic edge networks. In *Proc. of IEEE ICFC'20*. 64–72.

[33] Mengyuan Chao, Chen Yang, Yukun Zeng, and Radu Stoleru. 2018. F-MStorm: Feedback-based online distributed mobile stream processing. In *Proc. of IEEE/ACM SEC'18*. 273–285.

[34] Shilpa Chaturvedi and Yogesh Simmhan. 2019. Toward resilient stream processing on clouds using moving target defense. In *Proc. of IEEE ISORC'19*. 134–142.

[35] Shilpa Chaturvedi, Sahil Tyagi, and Yogesh Simmhan. 2021. Cost-effective sharing of streaming dataflows for IoT applications. *IEEE Trans. Cloud Comput.* 9, 4 (2021), 1391–1407.

[36] Andreas Chatzistergiou and Stratis D. Viglas. 2014. Fast heuristics for near-optimal task allocation in data stream processing over clusters. In *Proc. of ACM CIKM'14*. 1579–1588.

[37] Xin Chen, Ymir Vigfusson, Douglas M. Blough, Fang Zheng, Kun-Lung Wu, and Liting Hu. 2017. GOVERNOR: Smoother stream processing through smarter backpressure. In *Proc. of IEEE ICAC'17*. 145–154.

[38] Dazhao Cheng, Xiaobo Zhou, Yu Wang, and Changjun Jiang. 2018. Adaptive scheduling parallel jobs with dynamic batching in spark streaming. *IEEE Trans. Parallel Distrib. Syst.* 29, 12 (2018), 2672–2685.

[39] Gianpaolo Cugola and Alessandro Margara. 2012. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.* 44, 3 (2012), Article 15, 62 pages.

[40] Gianpaolo Cugola and Alessandro Margara. 2013. Deployment strategies for distributed complex event processing. *Computing* 95, 2 (2013), 129–156.

[41] Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. 2014. Adaptive stream processing using dynamic batch sizing. In *Proc. of ACM SoCC'14*. Article 16, 13 pages.

[42] Miyuru Dayarathna and Srinath Perera. 2018. Recent advancements in event processing. *ACM Comput. Surv.* 51, 2 (2018), Article 33, 36 pages.

[43] Tiziano De Matteis and Gabriele Mencagli. 2017. Elastic scaling for distributed latency-sensitive data stream operators. In *Proc. of PDP'17*. IEEE, Los Alamitos, CA, 61–68.

[44] Tiziano De Matteis and Gabriele Mencagli. 2017. Proactive elasticity and energy awareness in data stream processing. *J. Syst. Softw.* 127 (2017), 302–319.

[45] Felipe R. de Souza, Alexandre da Silva Veith, Marcos D. de Assunção, and Eddy Caron. 2020. Scalable joint optimization of placement and parallelism of data stream processing applications on cloud-edge infrastructure. In *Service-Oriented Computing*. Lecture Notes in Computer Science, Vol. 12571. Springer, 149–164.

[46] Guangxiang Du and Indranil Gupta. 2016. New techniques to curtail the tail latency in stream processing systems. In *Proc. of DCC@PODC'16*. ACM, New York, NY, Article 7, 6 pages.

[47] Christopher Eibel, Christian Gulden, Wolfgang Schröder-Preikschat, and Tobias Distler. 2018. Strome: Energy-aware data-stream processing. In *Distributed Applications and Interoperable Systems*. Lecture Notes in Computer Science, Vol. 10853. Springer, 40–57.

[48] Leila Eskandari, Zhiyi Huang, and David M. Eyers. 2016. P-scheduler: Adaptive hierarchical scheduling in Apache Storm. In *Proc. of ACSW'16*. ACM, New York, NY, Article 26, 10 pages.

[49] Junhua Fang, Pingfu Chao, Rong Zhang, and Xiaofang Zhou. 2019. Integrating workload balancing and fault tolerance in distributed stream processing system. *World Wide Web* 22, 6 (2019), 2471–2496.

[50] Junhua Fang, Rong Zhang, Tom Z. J. Fu, Zhenjie Zhang, Aoying Zhou, and Xiaofang Zhou. 2018. Distributed stream rebalance for stateful operator under workload variance. *IEEE Trans. Parallel Distrib. Syst.* 29, 10 (2018), 2223–2240.

[51] Omar Farhat, Khuzaima Daudjee, and Leonardo Querzoni. 2021. Klink: Progress-aware scheduling for streaming data systems. In *Proc. of ACM SIGMOD'21*. 485–498.

[52] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter R. Pietzuch. 2013. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proc. of ACM SIGMOD'13*. 725–736.

[53] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. 2017. Dhalion: Self-regulating stream processing in Heron. *Proc. VLDB Endow.* 10, 12 (2017), 1825–1836.

[54] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. 2020. A survey on the evolution of stream processing systems. *CoRR* abs/2008.00842 (2020).

[55] Tom Z. J. Fu, Jianbing Ding, Richard T. B. Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. 2017. DRS: Autoscaling for real-time stream analytics. *IEEE/ACM Trans. Netw.* 25, 6 (2017), 3338–3352.

[56] Xinwei Fu, Talha Ghaffar, James C. Davis, and Dongyoon Lee. 2019. EdgeWise: A better stream processing engine for the edge. In *Proc. of USENIX ATC'19*. 929–946.

[57] Bugra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. 2014. Elastic scaling for data stream processing. *IEEE Trans. Parallel Distrib. Syst.* 25, 6 (2014), 1447–1463.

[58] Lukasz Golab and M. Tamer Özsu. 2003. Issues in data stream management. *ACM SIGMOD Rec.* 32, 2 (2003), 5–14.

[59] Xiaohui Gu, Philip S. Yu, and Klara Nahrstedt. 2005. Optimal component composition for scalable stream processing. In *Proc. of IEEE ICDCS'05*. 773–782.

[60] Vincenzo Gulisano, Ricardo Jiménez-Peris, Marta Patiño-Martinez, Claudio Soriente, and Patrick Valduriez. 2012. StreamCloud: An elastic and scalable data streaming system. *IEEE Trans. Parallel Distrib. Syst.* 23, 12 (2012), 2351–2365.

[61] Vincenzo Gulisano, Marina Papatriantafilou, and Alessandro Vittorio Papadopoulos. 2019. Elasticity. In *Encyclopedia of Big Data Technologies*. Springer.

[62] Qingsong Guo and Yongluan Zhou. 2017. CBP: A new parallelization paradigm for massively distributed stream processing. In *Database Systems for Advanced Applications*. Lecture Notes in Computer Science, Vol. 10178. Springer, 304–320.

[63] Qingsong Guo and Yongluan Zhou. 2017. Stateful load balancing for parallel stream processing. In *Euro-Par 2017: Parallel Processing Workshops*. Lecture Notes in Computer Science, Vol. 10659. Springer, 80–93.

[64] Zheng Han, Rui Chu, Haibo Mi, and Huaimin Wang. 2014. Elastic allocator: An adaptive task scheduler for streaming query in the cloud. In *Proc. of IEEE SOSE'14*. 284–289.

[65] Aurélien Havet, Rafael Pires, Pascal Felber, Marcelo Pasin, Romain Rouvoy, and Valerio Schiavoni. 2017. SecureStreams: A reactive middleware framework for secure data stream processing. In *Proc. of ACM DEBS'17*. 124–133.

[66] Benjamin Heintz, Abhishek Chandra, and Ramesh K. Sitaraman. 2020. Optimizing timeliness and cost in geo-distributed streaming analytics. *IEEE Trans. Cloud Comput.* 8, 1 (2020), 232–245.

[67] Thomas Heinze, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. 2014. Latency-aware elastic scaling for distributed data stream processing systems. In *Proc. of ACM DEBS'14*. 13–22.

[68] Thomas Heinze, Valerio Pappalardo, Zbigniew Jerzak, and Christof Fetzer. 2014. Auto-scaling techniques for elastic data stream processing. In *Proc. of IEEE ICDEW'14*. 296–302.

[69] Thomas Heinze, Lars Roediger, Andreas Meister, Yuanzhen Ji, Zbigniew Jerzak, and Christof Fetzer. 2015. Online parameter optimization for elastic data stream processing. In *Proc. of ACM SoCC'15*. 276–287.

[70] Thomas Heinze, Mariam Zia, Robert Krahn, Zbigniew Jerzak, and Christof Fetzer. 2015. An adaptive replication scheme for elastic data stream processing systems. In *Proc. of ACM DEBS'15*. 150–161.

[71] Herodotos Herodotou, Yuxing Chen, and Jiaheng Lu. 2020. A survey on automatic parameter tuning for big data processing systems. *ACM Comput. Surv.* 53, 2 (2020), Article 43, 37 pages.

[72] Nicolas Hidalgo, Daniel Wladdimiro, and Erika Rosas. 2017. Self-adaptive processing graph with operator fission for elastic stream processing. *J. Syst. Softw.* 127 (2017), 205–216.

[73] Martin Hirzel, Robert Soulé, Scott Schneider, Bugra Gedik, and Robert Grimm. 2013. A catalog of stream processing optimizations. *ACM Comput. Surv.* 46, 4 (2013), Article 46, 34 pages.

[74] Christoph Hochreiner, Michael Vögler, Stefan Schulte, and Schahram Dustdar. 2016. Elastic stream processing for the Internet of Things. In *Proc. of IEEE CLOUD'16*. 100–107.

[75] Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, John Liagouris, and Timothy Roscoe. 2019. Megaphone: Latency-conscious state migration for distributed streaming dataflows. *Proc. VLDB Endow.* 12, 9 (2019), 1002–1015.

[76] Mohammad R. Hoseiny Farahabady, Ali Jannesari, Javid Taheri, Wei Bao, Albert Y. Zomaya, and Zahir Tari. 2020. Q-Flink: A QoS-aware controller for Apache Flink. In *Proc. of IEEE/ACM CCGRID'20*. 629–638.

[77] Mohammad R. Hoseiny Farahabady, Hamid R. Dehghani Samani, Yidan Wang, Albert Y. Zomaya, and Zahir Tari. 2016. A QoS-aware controller for Apache storm. In *Proc. of IEEE NCA'16*. 334–342.

[78] Mohammad R. Hoseiny Farahabady, Albert Y. Zomaya, and Zahir Tari. 2017. QoS- and contention- aware resource provisioning in a stream processing engine. In *Proc. of IEEE CLUSTER'17*. 137–146.

[79] Qun Huang and Patrick P. C. Lee. 2016. Toward high-performance distributed stream processing via approximate fault tolerance. *Proc. VLDB Endow.* 10, 3 (2016), 73–84.

[80] Xi Huang, Ziyu Shao, and Yang Yang. 2020. POTUS: Predictive online tuple scheduling for data stream processing systems. *IEEE Trans. Cloud Comput.* To appear.

[81] Jeong-Hyon Hwang, Ugur Çetintemel, and Stan Zdonik. 2008. Fast and highly-available stream processing over wide area networks. In *Proc. of IEEE ICDE'08*. 804–813.

[82] Shigeru Imai, Stacy Patterson, and Carlos A. Varela. 2018. Uncertainty-aware elastic virtual machine scheduling for stream processing systems. In *Proc. of IEEE/ACM CCGRID'18*. 62–71.

[83] Changjiang Jia, Yan Cai, Yuen-Tak Yu, and T. H. Tse. 2016. 5W+1H pattern: A perspective of systematic mapping studies and a case study on cloud software testing. *J. Syst. Softw.* 116 (2016), 206–219.

[84] Aymen Jlassi and Cédric Tedeschi. 2020. Merge, split, and cluster: Dynamic deployment of stream processing applications. In *Proc. of IEEE/ACM CCGRID'20*. 71–80.

[85] Albert Jonathan, Abhishek Chandra, and Jon B. Weissman. 2020. WASP: Wide-area adaptive stream processing. In *Proc. of ACM/IFIP MIDDLEWARE'20*. ACM, New York, NY, 221–235.

[86] Basri Kahveci and Bugra Gedik. 2020. Joker: Elastic stream processing with organic adaptation. *J. Parallel Distrib. Comput.* 137 (2020), 205–223.

[87] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava C. Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three steps is all you need: Fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *Proc. of USENIX OSDI'18*. 783–798.

[88] Faria Kalim, Le Xu, Sharanya Bathey, Richa Meherwal, and Indranil Gupta. 2018. Henge: Intent-driven multi-tenant stream processing. In *Proc. of ACM SoCC'18*. 249–262.

[89] Evangelia Kalyvianaki, Themistoklis Charalambous, Marco Fiscato, and Peter Pietzuch. 2012. Overload management in data stream processing systems with latency guarantees. In *Proc. of FCW'12*.

[90] Evangelia Kalyvianaki, Marco Fiscato, Theodoros Salonidis, and Peter R. Pietzuch. 2016. THEMIS: Fairness in federated stream processing under overload. In *Proc. of ACM SIGMOD'16*. 541–553.

[91] Evangelia Kalyvianaki, Wolfram Wiesemann, Quang H. Vu, Daniel Kuhn, and Peter R. Pietzuch. 2011. SQPR: Stream query planning with reuse. In *Proc. of IEEE ICDE'11*. 840–851.

[92] Nikos R. Katsipoulakis, Alexandros Labrinidis, and Panos K. Chrysanthis. 2017. A holistic view of stream partitioning costs. *Proc. VLDB Endow.* 10, 11 (2017), 1286–1297.

[93] Nikos R. Katsipoulakis, Alexandros Labrinidis, and Panos K. Chrysanthis. 2018. Concept-driven load shedding: Reducing size and error of voluminous and variable data streams. In *Proc. of IEEE Big Data'18*. 418–427.

[94] Nikos R. Katsipoulakis, Alexandros Labrinidis, and Panos K. Chrysanthis. 2020. SPEAr: Expediting stream processing with accuracy guarantees. In *Proc. of IEEE ICDE'20*. 1105–1116.

[95] Wilhelm Kleiminger, Evangelia Kalyvianaki, and Peter R. Pietzuch. 2011. Balancing load in stream processing with the cloud. In *Proc. of IEEE ICDE'11*. 16–21.

[96] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic ephemeral storage for serverless analytics. In *Proc. of USENIX OSDI'18*. 427–444.

[97] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter R. Pietzuch. 2016. SABER: Window-based hybrid stream processing for heterogeneous architectures. In *Proc. of ACM SIGMOD'16*. 555–569.

[98] Roland Kotto Kombi, Nicolas Lumineau, and Philippe Lamarre. 2017. A preventive auto-parallelization approach for elastic stream processing. In *Proc. of IEEE ICDCS'17*. 1532–1542.

[99] Alok G. Kumbhare, Yogesh Simmhan, and Viktor K. Prasanna. 2014. PLAStiCC: Predictive look-ahead scheduling for continuous dataflows on clouds. In *Proc. of IEEE/ACM CCGrid'14*. 344–353.

[100] Alok G. Kumbhare, Yogesh L. Simmhan, Marc Frincu, and Viktor K. Prasanna. 2015. Reactive resource provisioning heuristics for dynamic dataflows on cloud infrastructure. *IEEE Trans. Cloud Comput.* 3, 2 (2015), 105–118.

[101] Geetika T. Lakshmanan, Ying Li, and Robert E. Strom. 2008. Placement strategies for internet-scale data stream systems. *IEEE Internet Comput.* 12, 6 (2008), 50–60.

[102] Geetika T. Lakshmanan and Robert E. Strom. 2008. Biologically-inspired distributed middleware management for stream processing systems. In *Middleware 2008*. Lecture Notes in Computer Science, Vol. 5346. Springer, 223–242.

[103] Do Le Quoc, Martin Beck, Pramod Bhatotia, Ruichuan Chen, Christof Fetzer, and Thorsten Strufe. 2017. PrivApprox: Privacy-preserving stream analytics. In *Proc. of USENIX ATC'17*. 659–672.

[104] Do Le Quoc, Ruichuan Chen, Pramod Bhatotia, Christof Fetzer, Volker Hilt, and Thorsten Strufe. 2017. StreamApprox: Approximate computing for stream analytics. In *Proc. of ACM/IFIP/USENIX MIDDLEWARE'17*. ACM, New York, NY, 185–197.

[105] Chuan Lei and Elke A. Rundensteiner. 2014. Robust distributed query processing for streaming data. *ACM Trans. Database Syst.* 39, 2 (2014), Article 17, 45 pages.

[106] Jack Li, Calton Pu, Yuan Chen, Daniel Gmach, and Dejan S. Milojicic. 2016. Enabling elastic stream processing in shared clusters. In *Proc. of IEEE CLOUD'16*. 108–115.

[107] Kejian Li, Gang Liu, and Minhua Lu. 2019. A holistic stream partitioning algorithm for distributed stream processing systems. In *Proc. of PDCAT'19*. IEEE, Los Alamitos, CA, 202–207.

[108] Teng Li, Zhiyuan Xu, Jian Tang, and Yanzhi Wang. 2018. Model-free control for distributed stream data processing using deep reinforcement learning. *Proc. VLDB Endow.* 11, 6 (2018), 705–718.

[109] Xiaofei Liao, Yu Huang, Long Zheng, and Hai Jin. 2019. Efficient time-evolving stream processing at scale. *IEEE Trans. Parallel Distrib. Syst.* 30, 10 (2019), 2165–2178.

[110] Xunyun Liu and Rajkumar Buyya. 2017. D-storm: Dynamic resource-efficient scheduling of stream processing applications. In *Proc. of ICPADS'17*. 485–492.

[111] Xunyun Liu and Rajkumar Buyya. 2020. Resource management and scheduling in distributed stream processing systems: A taxonomy, review, and future directions. *ACM Comput. Surv.* 53, 3 (2020), Article 50, 41 pages.

[112] Xunyun Liu, Amir Vahid Dastjerdi, Rodrigo N. Calheiros, Chenhao Qu, and Rajkumar Buyya. 2018. A stepwise auto-profiling method for performance optimization of streaming applications. *ACM Trans. Auton. Adapt. Syst.* 12, 4 (2018), Article 24, 33 pages.

[113] Björn Lohrmann, Peter Janacik, and Odej Kao. 2015. Elastic stream processing with latency guarantees. In *Proc. of IEEE ICDCS'15*. 399–410.

[114] Björn Lohrmann, Daniel Warneke, and Odej Kao. 2014. Nephele streaming: Stream processing under QoS constraints at scale. *Clust. Comput.* 17, 1 (2014), 61–78.

[115] Federico Lombardi, Leonardo Aniello, Silvia Bonomi, and Leonardo Querzoni. 2018. Elastic symbiotic scaling of operators and resources in stream processing systems. *IEEE Trans. Parallel Distrib. Syst.* 29, 3 (2018), 572–585.

[116] Manisha Luthra, Boris Koldehofe, Pascal Weisenburger, Guido Salvaneschi, and Raheel Arif. 2018. TCEP: Adapting to dynamic user environments by enabling transitions between operator placement mechanisms. In *Proc. of ACM DEBS'18*. 136–147.

[117] Kasper Madsen, Yongluan Zhou, and Jianneng Cao. 2017. Integrative dynamic reconfiguration in a parallel stream processing engine. In *Proc. of IEEE ICDE'17*. 227–230.

[118] Kasper Madsen, Yongluan Zhou, and Li Su. 2016. Enorm: Efficient window-based computation in large-scale distributed stream processing systems. In *Proc. of ACM DEBS'16*. 37–48.

[119] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, et al. 2018. Chi: A scalable and programmable control plane for distributed stream processing systems. *Proc. VLDB Endow.* 11, 10 (2018), 1303–1316.

[120] Vania Marangozova-Martin, Noël De Palma, and Ahmed El-Rheddane. 2019. Multi-level elasticity for data stream processing. *IEEE Trans. Parallel Distrib. Syst.* 30, 10 (2019), 2326–2337.

[121] Yuan Mei, Luwei Cheng, Vanish Talwar, Michael Y. Levin, Gabriela Jacques-Silva, Nikhil Simha, Anirban Banerjee, et al. 2020. Turbine: Facebook's service management platform for stream processing. In *Proc. of IEEE ICDE'20*. 1591–1602.

[122] Gabriele Mencagli. 2016. A game-theoretic approach for elastic distributed data stream processing. *ACM Trans. Auton. Adapt. Syst.* 11, 2 (2016), Article 13, 34 pages.

[123] Gabriele Mencagli, Massimo Torquati, and Marco Danelutto. 2018. Elastic-PPQ: A two-level autonomic system for spatial preference query processing over dynamic data streams. *Future Gener. Comput. Syst.* 79 (2018), 862–877.

[124] Gabriele Mencagli, Massimo Torquati, Marco Danelutto, and Tiziano De Matteis. 2017. Parallel continuous preference queries over out-of-order and bursty data streams. *IEEE Trans. Parallel Distrib. Syst.* 28, 9 (2017), 2608–2624.

[125] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Rhino: Efficient management of very large distributed state for stream processing engines. In *Proc. of ACM SIGMOD'20*. ACM, New York, NY, 2471–2486.

[126] Weimin Mu, Zongze Jin, Junwei Wang, Weilin Zhu, and Weiping Wang. 2019. BGElasor: Elastic-scaling framework for distributed streaming processing with deep neural network. In *Network and Parallel Computing*. Lecture Notes in Computer Science, Vol. 11783. Springer, 120–131.

[127] Hannaneh Najdataei, Yiannis Nikolakopoulos, Marina Papatriantafilou, Philippas Tsigas, and Vincenzo Gulisano. 2019. STRETCH: Scalable and elastic deterministic streaming analysis with virtual shared-nothing parallelism. In *Proc. of ACM DEBS'19*. 7–18.

[128] Matteo Nardelli, Valeria Cardellini, Vincenzo Grassi, and Francesco Lo Presti. 2019. Efficient operator placement for distributed data stream processing applications. *IEEE Trans. Parallel Distrib. Syst.* 30, 8 (2019), 1753–1767.

[129] Stefan Nastic, Thomas Rausch, Ognjen Scekic, Schahram Dustdar, Marjan Gusev, Bojana Koteska, Magdalena Kostoska, Boro Jakimovski, Sasko Ristov, and Radu Prodan. 2017. A serverless real-time data analytics platform for edge computing. *IEEE Internet Comput.* 21, 4 (2017), 64–71.

[130] Xiang Ni, Scott Schneider, Raju Pavuluri, Jonathan Kaus, and Kun-Lung Wu. 2019. Automating multi-level performance elastic components for IBM streams. In *Proc. of ACM/IFIP Middleware'19*. ACM, New York, NY, 163–175.

[131] Dan O'Keeffe, Theodoros Salonidis, and Peter R. Pietzuch. 2018. Frontier: Resilient edge processing for the Internet of Things. *Proc. VLDB Endow.* 11, 10 (2018), 1178–1191.

[132] Beate Ottenwälder, Boris Koldehofe, Kurt Rothermel, Kirak Hong, David J. Lillethun, and Umakishore Ramachandran. 2014. MCEP: A mobility-aware complex event processing system. *ACM Trans. Internet Technol.* 14, 1 (2014), Article 6, 24 pages.

[133] Dimitris Palyvos-Giannas, Gabriele Mencagli, Marina Papatriantafilou, and Vincenzo Gulisano. 2021. Lachesis: A middleware for customizing OS scheduling of stream processing queries. In *Proc. of ACM Middleware'21*. 365–378.

[134] Olga Papaemmanouil, Ugur Çetintemel, and John Jannotti. 2009. Supporting generic cost models for wide-area stream processing. In *Proc. of IEEE ICDE'09*. 1084–1095.

[135] Heejin Park, Shuang Zhai, Long Lu, and Felix X. Lin. 2019. Streambox-TZ: Secure stream analytics at the edge with trustzone. In *Proc. of USENIX ATC'19*. 537–554.

[136] Thao N. Pham, Panos K. Chrysanthis, and Alexandros Labrinidis. 2016. Avoiding class warfare: Managing continuous queries with differentiated classes of service. *VLDB J.* 25, 2 (2016), 197–221.

[137] Thao N. Pham, Nikos R. Katsipoulakis, Panos K. Chrysanthis, and Alexandros Labrinidis. 2017. Uninterruptible migration of continuous queries without operator state migration. *ACM SIGMOD Rec.* 46, 3 (2017), 17–22.

[138] Peter R. Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo I. Seltzer. 2006. Network-aware operator placement for stream-processing systems. In *Proc. of IEEE ICDE'06*. 49–60.

[139] Cui Qin, Holger Eichelberger, and Klaus Schmid. 2019. Enactment of adaptation in data stream processing with latency implications—A systematic literature review. *Inf. Softw. Technol.* 111 (2019), 1–21.

[140] Parisa Rahimzadeh, Jinsung Lee, Youngbin Im, Siun-Chuon Mau, Eric C. Lee, Bradford O. Smith, Fatemah Al-Duoli, Carlee Joe-Wong, and Sangtae Ha. 2020. SPARCLE: Stream processing applications over dispersed computing networks. In *Proc. of IEEE ICDCS'20*. 1067–1078.

[141] Sajith Ravindra, Miyuru Dayarathna, and Sanath Jayasena. 2017. Latency aware elastic switching-based stream processing over compressed data streams. In *Proc. of ACM/SPEC ICPE'17*. 91–102.

[142] Thomas Repantis, Xiaohui Gu, and Vana Kalogeraki. 2009. QoS-aware shared component composition for distributed stream processing systems. *IEEE Trans. Parallel Distrib. Syst.* 20, 7 (2009), 968–982.

[143] Nicolo Rivetti, Emmanuelle Anceaume, Yann Busnel, Leonardo Querzoni, and Bruno Sericola. 2016. Online scheduling for shuffle grouping in distributed stream processing systems. In *Proc. of ACM/IFIP/USENIX Middleware'16.*

[144] Stamatia Rizou, Frank Dürr, and Kurt Rothermel. 2010. Solving the multi-operator placement problem in large-scale operator networks. In *Proc. of IEEE ICCCN'10*. 1–6.

[145] Henriette Röger and Ruben Mayer. 2019. A comprehensive survey on parallelization and elasticity in stream processing. *ACM Comput. Surv.* 52, 2 (2019), Article 36, 37 pages.

[146] Olubisi Runsewe and Nancy Samaan. 2017. Cloud resource scaling for big data streaming applications using a layered multi-dimensional hidden Markov model. In *Proc. of IEEE/ACM CCGRID'17*. 848–857.

[147] Gabriele Russo Russo, Valeria Cardellini, Giuliano Casale, and Francesco Lo Presti. 2021. MEAD: Model-based vertical auto-scaling for data stream processing. In *Proc. of IEEE/ACM CCGRID'21*. 314–323.

[148] Gabriele Russo Russo, Valeria Cardellini, and Francesco Lo Presti. 2019. Reinforcement learning based policies for elastic stream processing on heterogeneous resources. In *Proc. of ACM DEBS'19*. 31–42.

[149] Gabriele Russo Russo, Valeria Cardellini, Francesco Lo Presti, and Matteo Nardelli. 2021. Towards a security-aware deployment of data streaming applications in fog computing. In *Fog/Edge Computing For Security, Privacy, and Applications*. Springer, 355–385.

[150] Hooman P. Sajjad, Ken Danniswara, Ahmad Al-Shishtawy, and Vladimir Vlassov. 2016. SpanEdge: Towards unifying stream processing over central and near-the-edge data centers. In *Proc. of IEEE/ACM SEC'16*. 168–178.

[151] Farah Aït Salaht, Frédéric Desprez, and Adrien Lebre. 2020. An overview of service placement problem in fog and edge computing. *ACM Comput. Surv.* 53, 3 (2020), Article 65, 35 pages.

[152] Benjamin Satzger, Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. 2011. ESC: Towards an elastic stream computing platform for the cloud. In *Proc. of IEEE CLOUD'11*. 348–355.

[153] Enrique Saurez, Kirak Hong, Dave Lillethun, Umakishore Ramachandran, and Beate Ottenwälder. 2016. Incremental deployment and migration of geo-distributed situation awareness applications in the fog. In *Proc. of ACM DEBS'16*. 258–269.

[154] Scott Schneider, Henrique Andrade, Bugra Gedik, Alain Biem, and Kun-Lung Wu. 2009. Elastic scaling of data parallel operators in stream processing. In *Proc. of IEEE IPDPS'09*. 1–12.

[155] Scott Schneider, Joel L. Wolf, Kirsten Hildrum, Rohit Khandekar, and Kun-Lung Wu. 2016. Dynamic load balancing for ordered data-parallel regions in distributed streaming systems. In *Proc. of ACM/IFIP/USENIX Middleware'16*. ACM, New York, NY, Article 21, 14 pages.

[156] Scott Schneider and Kun-Lung Wu. 2017. Low-synchronization, mostly lock-free, elastic scheduling for streaming runtimes. In *Proc. of ACM SIGPLAN PLDI'17*. 648–661.

[157] M. A. Shah, J. M. Hellerstein, Sirish Chandrasekaran, and M. J. Franklin. 2003. Flux: An adaptive partitioning operator for continuous query systems. In *Proc. of ICDE'03*. IEEE, Los Alamitos, CA, 25–36.

[158] Mohamed A. Sharaf, Panos K. Chrysanthis, Alexandros Labrinidis, and Kirk Pruhs. 2008. Algorithms and metrics for processing multiple heterogeneous continuous queries. *ACM Trans. Database Syst.* 33, 1 (2008), Article 5, 44 pages.

[159] Anshu Shukla and Yogesh Simmhan. 2018. Toward reliable and rapid elasticity for streaming dataflows on clouds. In *Proc. of IEEE ICDCS'18*. 1096–1106.

[160] Alexandre da Silva Veith, Felipe R. de Souza, Marcos D. de Assunção, Laurent Lefèvre, and Julio C. Santos dos Anjos. 2019. Multi-objective reinforcement learning for reconfiguring data stream analytics on edge computing. In *Proc. of ICPP'19*. ACM, New York, NY, Article 106, 10 pages.

[161] Rayman Preet Singh, Bharath Kumarasubramanian, Prateek Maheshwari, and Samarth Shetty. 2020. Auto-sizing for stream processing applications at LinkedIn. In *Proc. of USENIX HotCloud'20.*

[162] Ahmad Slo, Sukanya Bhowmik, and Kurt Rothermel. 2019. eSPICE: Probabilistic load shedding from input event streams in complex event processing. In *Proc. of ACM/IFIP Middleware'19*. ACM, New York, NY, 215–227.

[163] Ahmad Slo, Sukanya Bhowmik, and Kurt Rothermel. 2020. State-aware load shedding from input event streams in complex event processing. *IEEE Trans. Big Data.* To appear.

[164] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. 2005. The 8 requirements of real-time stream processing. *ACM SIGMOD Rec.* 34, 4 (2005), 42–47.

[165] Dawei Sun, Shang Gao, Xunyun Liu, Xindong You, and Rajkumar Buyya. 2020. Dynamic redirection of real-time data streams for elastic stream computing. *Future Gener. Comput. Syst.* 112 (2020), 193–208.

[166] Dawei Sun, Guangyan Zhang, Songlin Yang, Weimin Zheng, Samee Ullah Khan, and Keqin Li. 2015. Re-Stream: Real-time and energy-efficient resource scheduling in big data stream computing environments. *Inf. Sci.* 319 (2015), 92–112.

[167] Nicoleta Tantalaki, Stavros Souravlas, and Manos Roumeliotis. 2020. A review on big data real-time stream processing and its scheduling techniques. *Int. J. Parallel Emergent Distributed Syst.* 35, 5 (2020), 571–601.

[168] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. 2003. Load shedding in a data stream manager. In *Proc. of VLDB'03*. 309–320.

[169] Nesime Tatbul, Uğur Çetintemel, and Stanley B. Zdonik. 2007. Staying FIT: Efficient load shedding techniques for distributed stream processing. In *Proc. of VLDB'07*. ACM, New York, NY, 159–170.

[170] Abhishek Tiwari, Brian Ramprasad, Seyed H. Mortazavi, Moshe Gabel, and Eyal de Lara. 2019. Reconfigurable streaming for the mobile edge. In *Proc. of HotMobile'19*. ACM, New York, NY, 153–158.

[171] Quoc-Cuong To, Juan Soto, and Volker Markl. 2018. A survey of state management in big data processing systems. *VLDB J.* 27, 6 (2018), 847–872.

[172] Rafael Tolosana-Calasanz, Javier Diaz Montes, Omer F. Rana, and Manish Parashar. 2017. Feedback-control and queueing theory-based resource management for streaming applications. *IEEE Trans. Parallel Distrib. Syst.* 28, 4 (2017), 1061–1075.

[173] Geoffrey Phi C. Tran, John Paul Walters, and Stephen P. Crago. 2018. Reducing tail latencies while improving resiliency to timing errors for stream processing workloads. In *Proc. of IEEE/ACM UCC'18*. 194–203.

[174] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. 2003. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. Knowl. Data Eng.* 15, 3 (2003), 555–568.

[175] Radu Tudoran, Olivier Nano, Ivo Santos, Alexandru Costan, Hakan Soncu, Luc Bouge, and Gabriel Antoniu. 2014. JetStream: Enabling high performance event streaming across cloud data-centers. In *Proc. of ACM DEBS'14*. 23–34.

[176] Jan Sipke van der Veen, Bram van der Waaij, Elena Lazovik, Wilco Wijbrandi, and Robert J. Meijer. 2015. Dynamically scaling Apache Storm for the analysis of streaming data. In *Proc. of IEEE BigDataService'15*. 154–161.

[177] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J. Franklin, Benjamin Recht, and Ion Stoica. 2017. Drizzle: Fast and adaptable stream processing at scale. In *Proc. of ACM SOSP'17*. 374–389.

[178] Ke Wang, Avrilia Floratou, Ashvin Agrawal, and Daniel Musgrave. 2020. Spur: Mitigating slow instances in large-scale streaming pipelines. In *Proc. of ACM SIGMOD'20*. 2271–2285.

[179] Li Wang, Tom Z. J. Fu, Richard T. B. Ma, Marianne Winslett, and Zhenjie Zhang. 2019. Elasticutor: Rapid elasticity for realtime stateful stream processing. In *Proc. of ACM SIGMOD'19*. 573–588.

[180] Yidan Wang, Zahir Tari, Mohammad R. Hoseiny Farahabady, and Albert Y. Zomaya. 2017. Model-based scheduling for stream processing systems. In *Proc. of IEEE HPCC/SmartCity/DSS'17*. 215–222.

[181] Yidan Wang, Zahir Tari, Xiaoran Huang, and Albert Y. Zomaya. 2019. A network-aware and partition-based resource management scheme for data stream processing. In *Proc. of ICPP'19*. ACM, New York, NY, Article 20, 10 pages.

[182] Xiaohui Wei, Lina Li, Xiang Li, Xingwang Wang, Shang Gao, and Hongliang Li. 2019. Pec: Proactive elastic collaborative resource scheduling in data stream processing. *IEEE Trans. Parallel Distrib. Syst.* 30, 7 (2019), 1628–1642.

[183] Song Wu, Die Hu, Shadi Ibrahim, Hai Jin, Jiang Xiao, Fei Chen, and Haikun Liu. 2019. When FPGA-accelerator meets stream data processing in the edge. In *Proc. of IEEE ICDCS'19*. 1818–1829.

[184] Song Wu, Mi Liu, Shadi Ibrahim, Hai Jin, Lin Gu, Fei Chen, and Zhiyi Liu. 2018. TurboStream: Towards low-latency data stream processing. In *Proc. of IEEE ICDCS'18*. 983–993.

[185] Ying Xing, Stanley B. Zdonik, and Jeong-Hyon Hwang. 2005. Dynamic load distribution in the Borealis stream processor. In *Proc. of IEEE ICDE'05*. 791–802.

[186] Jielong Xu, Zhenhua Chen, Jian Tang, and Sen Su. 2014. T-storm: Traffic-aware online scheduling in storm. In *Proc. of IEEE ICDCS'14*. 535–544.

[187] Jinlai Xu, Balaji Palanisamy, Qingyang Wang, Heiko Ludwig, and Sandeep Gopisetty. 2022. Amnis: Optimized stream processing for edge computing. *J. Parallel Distrib. Comput.* 160 (2022), 49–64.

[188] Le Xu, Boyang Peng, and Indranil Gupta. 2016. Stela: Enabling stream processing systems to scale-in and scale-out on-demand. In *Proc. of IEEE IC2E'16*. 22–31.

[189] Le Xu, Shivaram Venkataraman, Indranil Gupta, Luo Mai, and Rahul Potharaju. 2021. Move fast and meet deadlines: Fine-grained real-time stream processing with Cameo. In *Proc. of USENIX NSDI'21*. 389–405.

[190] Nikos Zacheilas, Vana Kalogeraki, Nikolaos Zygouras, Nikolaos Panagiotou, and Dimitrios Gunopulos. 2015. Elastic complex event processing exploiting prediction. In *Proc. of IEEE Big Data'15*. 213–222.

[191] Nikos Zacheilas, Nikolas Zygouras, Nikolaos Panagiotou, Vana Kalogeraki, and Dimitrios Gunopulos. 2016. Dynamic load balancing techniques for distributed complex event processing systems. In *Distributed Applications and Interoperable Systems*. Lecture Notes in Computer Science, Vol. 9687. Springer, 174–188.

[192] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *Proc. of ACM SOSP'13*. 423–438.

[193] Ali Reza Zamani, Daniel Balouek-Thomert, Juan J. Villalobos, Ivan Rodero, and Manish Parashar. 2020. An edge-aware autonomic runtime for data streaming and in-transit processing. *Future Gener. Comput. Syst.* 110 (2020), 107–118.

[194] Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp Grulich, Sebastian Bress, Jonas Traub, and Voker Markl. 2020. The NebulaStream platform for data and application management in the Internet of Things. In *Proc. of CIDR'20*.

[195] Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A. Lee. 2018. AWStream: Adaptive wide-area streaming analytics. In *Proc. of ACM SIGCOMM'18*. 236–252.

[196] Quan Zhang, Yang Song, Ramani Routray, and Weisong Shi. 2016. Adaptive block and batch sizing for batched stream processing system. In *Proc. of IEEE ICAC'16*. 35–44.

[197] Shuhao Zhang, Feng Zhang, Yingjun Wu, Bingsheng He, and Paul Johns. 2019. Hardware-conscious stream processing: A survey. *ACM SIGMOD Rec.* 48, 4 (2019), 18–29.

[198] Yongluan Zhou, Beng Chin Ooi, Kian-Lee Tan, and Ji Wu. 2006. Efficient dynamic operator placement in a locally distributed continuous query system. In *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*. Lecture Notes in Computer Science, Vol. 4275. Springer, 54–71.

[199] Yongluan Zhou, Ji Wu, and Ahmed Khan Leghari. 2013. Multi-query scheduling for time-critical data stream applications. In *Proc. of SSDBM'13*. ACM, New York, NY, Article 15, 12 pages.

Received April 2021; revised December 2021; accepted January 2022