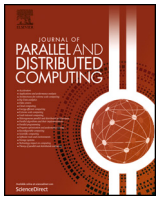
Journal of Parallel and Distributed Computing 199 (2025) 105041

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc



Latency-aware placement of stream processing operators in modern-day stream processing frameworks

Raphael Ecker [a], Vasileios Karagiannis [b], Michael Sober [c], Stefan Schulte [c,*]

[a] TU Wien, Vienna, Austria
[b] Center for Digital Safety & Security, Austrian Institute of Technology, Vienna, Austria
[c] Christian Doppler Laboratory for Blockchain Technologies for the Internet of Things, Hamburg University of Technology, Hamburg, Germany

ARTICLE INFO

Keywords:
Data stream processing
Compute continuum
Edge computing
Internet of Things
Apache storm

ABSTRACT

The rise of the Internet of Things has substantially increased the number of interconnected devices at the edge of the network. As a result, a large number of computations are now distributed in the compute continuum, spanning from the edge to the cloud, generating vast amounts of data. Stream processing is typically employed to process this data in near real-time due to its efficiency in handling continuous streams of information in a scalable manner. However, many stream processing approaches do not consider the underlying network devices of the compute continuum as candidate resources for processing data. Moreover, many existing works do not consider the incurred network latency of performing computations on multiple devices in a distributed way. To avoid this, we formulate an optimization problem for utilizing the complete compute continuum resources and design heuristics to solve this problem efficiently. Furthermore, we integrate our heuristics into Apache Storm and perform experiments that show latency- and throughput-related benefits compared to alternatives.

1. Introduction

Data stream processing is widely used for processing data in near real-time, e.g., in factory automation or banking scenarios [1]. While a stream processing application could theoretically be executed on a single computational resource, the scale and scope of many applications require distributing the stream processing operators on multiple computational resources. Distributing the operators also allows for parallelizing tasks, significantly scaling the operations. Originally, most approaches to enable distributed data stream processing relied on cloud-based resources [2]. However, more recently, the utilization of edge resources in addition to the cloud has gained a lot of attention [3].

Today, resources from the edge of the network to the cloud are seen as an edge-cloud compute continuum, allowing the use of computational resources anywhere in between [4]. These resources can be, e.g., single-board computers, sensor nodes, cloudlets at the edge of the network, routers and switches, or virtual machines in the cloud [5]. Unlike the cloud's centralized data centers, edge devices provide geographically distributed resources closer to the data sources and the users [4]. This is especially useful when the data sources are geographically distributed, e.g., in the Internet of Things (IoT), because it decreases the communication overhead regarding network latency and the amount of data to be transferred to the cloud.

When applying compute continuum principles in data stream processing, the overall processing response time depends on how the operators are arranged on the available distributed resources based on the location of the data sources. Thus, the geographical distribution of computational resources in the compute continuum affects the network latency of the communication between the operators [3]. This leads to the question of how operators should be distributed on available resources to meet the near real-time requirements of many stream processing applications, also known as the stream operator placement problem [6].

So far, various notable advancements have been proposed to perform efficient stream processing. The optimization of operator placement has been explored considering network usage aspects and by applying spring-based force models to estimate the latency between distributed nodes [7]. Also, novel approaches have shown improved operator placement decisions by leveraging fog computing resources and by considering resource availability [8,9].

Overall, many approaches to optimize the placement of stream processing operators have been introduced (this is also discussed in detail

* Corresponding author.
E-mail addresses: vasileios.karagiannis@ait.ac.at (V. Karagiannis), stefan.schulte@tuhh.de (S. Schulte).

https://doi.org/10.1016/j.jpdc.2025.105041
Received 19 February 2024; Received in revised form 21 October 2024; Accepted 20 January 2025
Available online 27 January 2025

in Section 7). Nevertheless, most of them do not consider the network latency between the operators. In addition, many existing approaches may not take full advantage of all the compute continuum which can include a variety of distributed computational resources. To address such limitations, this work aims to design a latency-aware stream operator placement approach tailored to the compute continuum. To this end, we formulate the placement of the operators on compute continuum resources as an optimization problem. Since the operator placement problem is typically NP-hard [10], we also focus on designing appropriate heuristics that approximate the optimal solution efficiently. Furthermore, we build a prototype based on Apache Storm and we perform an evaluation that shows great potential due to providing latency- and throughput-related benefits compared to alternatives.

In summary, this work provides the following contributions:

- We formulate an optimization problem for the placement of stream processing operators considering the latency between the distributed nodes of the compute continuum.
- We propose two heuristics for solving the proposed optimization problem, namely, Hill Climbing and Ant System, and we introduce a novel Hybrid heuristic.
- We implement the proposed heuristics using Apache Storm as the basis, and we provide thorough implementation details.
- We also implement custom mechanisms for resource monitoring and latency measurements in Apache Storm to provide the proposed heuristics with valuable input for the optimizations.
- Finally, we perform an extensive evaluation of the proposed approaches using numerous static and randomly generated stream processing scenarios, highlighting the benefits of the hybrid heuristic over existing alternatives.

The remainder of this paper is structured as follows: Section 2 discusses the prerequisites of our work. Afterward, Section 3 offers information regarding the utilized system model, and Section 4 presents the design of our latency-aware placement heuristics. Then, we provide implementation details in Section 5 and evaluate the proposed solution in Section 6. In Section 7, we provide an overview of related work and conclude this paper in Section 8.

2. Background

This section establishes the key concepts and terminology to define the stream operator placement problem. To this end, we provide an overview of data stream processing as a paradigm for structuring computations and applications out of independently executed operations (Section 2.1). Afterward, we define a system model for the compute continuum (Section 2.2). Finally, we formulate the stream operator placement problem (Section 2.3).

2.1. Data stream processing

Stream processing is a software engineering pattern for performing highly scalable computations with low latency [2]. In stream processing, calculations are made by performing transformations on input data. A stream processing application is built from a network of transformation operations, i.e., the operators, which can be executed concurrently. Inputs to the application are assumed to receive new data items to process continuously, and as such, the application also continuously calculates new outputs [11]. Any data to be processed is then streamed through this network of operations, with a stream of computational results exiting it. As such, this pattern allows for highly scalable software due to the inherent concurrency and provides near real-time computations. Any input must only be routed through and transformed by the network. Furthermore, the operations can be distributed by executing them on separate computational resources [2].

Stream processing applications are connected in a graph-based topology commonly represented as Directed Acyclic Graphs (DAGs). In such a graph, data items are transferred along the directed edges between nodes, representing the stream processing operators. The edges are, therefore, called data streams, which are considered unbounded sequences of data items continuously received to be processed. Nodes in the graph without incoming edges represent the data sources. These nodes provide the inputs for stream processing applications. Nodes without outgoing edges are the data sinks. In a typical scenario, most nodes in a stream processing DAG are operations that both receive and output data items. Each node may have multiple incoming and outgoing edges [2].

The execution of a stream processing application requires mapping data sources, data sinks, and operators onto computational resources. Multiple instances of the same operator can be deployed on different computational resources to allow for a high degree of scalability. While this is usually unproblematic for stateless operators, stateful operators require splitting the associated state into independent sets to allow concurrent operations. Apart from spawning new operators, scalable data stream processing also requires stopping existing ones that are no longer required. Hence, the means to split, merge, or synchronize the state of operators need to be provided [12].

Operators are not directly placed on computational resources but are managed by a stream processing engine [13]. Several engines exist, with Apache Flink [14],[1] Apache Spark Structured Streaming[2] and Apache Storm[3] [15] being well-known examples of open-source stream processing engines.

The processes (or threads) of a stream processing engine that execute the operators are referred to as workers. Workers manage the transfer of data, the data streams, to other workers or potentially between local operators. Additionally, they handle functionalities such as fault tolerance and state transfer [13].

2.2. The compute continuum

As discussed above, using compute resources from the cloud is often insufficient in scenarios where data items from distributed sources must be processed in near real-time. Accordingly, integrating computational resources at the edge of the network has become increasingly important in data stream processing in recent years [16]. Performing computations at the edge instead of the cloud reduces the limitations of network bandwidth and latency for applications running on end devices at the edge of the network [17].

In general, the idea of integrating computational resources at the edge and in the cloud as well as end devices is also known as the *compute continuum*. According computing models are known as, e.g., edge, fog, mist, or mobile cloud computing, covering different parts of the compute continuum [18]. For instance, the edge computing model [19] mainly omits compute resources from the cloud and aims at real-time data processing. In contrast, mobile cloud computing [20] uses the cloud to perform computation- and storage-intensive tasks, with the data coming from endpoints.

Within the paper at hand, we follow the *SPEC-RG Reference Architecture on The Compute Continuum* [18] to define which entities our system model (Section 3) addresses:

1. *Data Source:* We assume that endpoints like IoT devices generate data streams (in terms of single data items).
2. *Offload Target:* Computing operations (here: data stream operators) are offloaded to workers at the edge or in the cloud. Usually, the

[1] https://flink.apache.org/.
[2] https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html.
[3] https://storm.apache.org/.

goal is to use edge resources since they allow low-latency computations. However, if the amount of resources at the edge is insufficient, cloud resources might be a fallback solution.
3. *Architecture:* Our system model, as described in more detail in Section 3, makes use of a distributed architecture. The operators are distributed to multiple edge and cloud nodes. Further, network link latency estimation follows a decentralized approach.
4. *Offload Service:* Since data stream processing relies on data freshness, we aim to offload compute services (rather than storage services) to single nodes.
5. *Compute Capacity:* The overall compute capacity in a data stream processing topology can be moderate to very high. Usually, edge devices provide smaller amounts of resources (in terms of CPU, memory, and storage) than powerful cloud resources.
6. *Network Latency:* The network latency can be low to moderate due to using computational resources at the edge and in the cloud.
7. *Network Type:* Nodes may be connected using wired and wireless networks.

2.3. Stream operator placement problem

The stream operator placement problem defines the challenge of finding a solution for placing operators of a stream processing application on computational resources such that the solution fulfills specific criteria. It is a combinatorial problem of M operators and N computational resources with $O(N^M)$ potential solutions. Every resource has a set of capabilities or available capacities, which are used to identify valid placements. Depending on the problem definition, these resources may have identical capacities and capabilities. They are then considered homogeneous, while resources with different capacities and capabilities are heterogeneous. The definition of the placement problem and implementation decide what capacities and capabilities are used. For example, a resource's capability could be the availability of a GPU and, therefore, the potential to perform GPU-accelerated computations [6]. Samples of common capacity limitations are available processing performance, RAM, or bandwidth. Operators to be placed can similarly require a set of capacities and capabilities [21].

Based on these, constraints that must be satisfied for a valid placement can be defined, such as guaranteeing a minimum amount of resource capacities an operator requires to perform its function. Similarly, it is possible to define an optimization function, which assigns a score for each solution. A solver would then aim to optimize the score by, e.g., maximizing the stream processing application's throughput or minimizing its resource usage [2].

Finding an optimal solution for such placement problems is relatively trivial for small instances and can usually be completed quickly [22]. However, it can become a significant issue for large problem instances, such as with many workers or operators, because it is currently unknown if these problems are solvable in polynomial time. With the optimal operator placement problem being an instance of a more general NP-complete problem, its computation quickly becomes intractable with the size of the problem [21]. Variants have therefore been established around different trade-offs in guarantees on the quality of the solution and its computational cost, e.g., the usage of heuristics to solve the stream operator placement problem [6], which will also be the focus of the solution presented in the work at hand.

3. Design

This section presents the system model (in Section 3.1) and system design considerations (in Section 3.2). Additionally, we discuss dynamic resource utilization and adaptation (in Section 3.3) and the estimation of network latency (in Section 3.4), which play important roles in our heuristics.

3.1. System model

As outlined in Section 2.2, we follow the SPEC-RG Reference Architecture for The Compute Continuum [18]. We allow computing operations to be offloaded to edge and cloud nodes and assume that data is generated by distributed IoT devices. In addition, we assume a distributed system architecture, which we present in more detail in the following.

Related approaches often define a strict separation of resources into a multi- or often three-layer system model, including endpoint-, edge-, and cloud-based compute nodes. In contrast, in the work at hand, the system is viewed as a compute continuum in which the resource type is not considered relevant in the context of the operator placement decisions. Instead, a resource is characterized by the computational capabilities it can provide. This way, all available resources at the edge or in the cloud are considered, whether network devices, single-board computers, or powerful servers. In addition, all the network links between the system's resources are taken into account, which is essential to achieve an efficient distribution of operators, e.g., with reduced latency and increased throughput [23].

Resource-constrained IoT devices such as wireless sensor nodes are not considered full-fledged resources in our system. The reason for this decision is that such resources may be highly unreliable due to lossy wireless transmissions or limited battery lifespans. Nevertheless, since such devices are omnipresent in the compute continuum, we consider them through their connection to a *supervisor*, i.e., in a nearby full-fledged resource [24].

In general, the approach presented in the work at hand aims to apply to arbitrary modern-day data stream processing frameworks. Nevertheless, the implemented system is partially influenced by the selected framework. We have selected Apache Storm [15] since it provides sub-millisecond latencies [25] and offers interfaces to integrate new placement heuristics easily. This has made Apache Storm a popular choice for evaluating operator placement approaches, e.g., in [26–31].

Interestingly, the aforementioned supervisors are a concept provided natively by Apache Storm, although similar entities also exist in other data stream processing engines. In short, a supervisor is a component integrated into a resource (i.e., a host) responsible for local task allocation. If resource-constrained devices that can offer computational capabilities are connected to a host, the supervisor is responsible for assigning tasks to them.

One drawback of Apache Storm is that it does not provide extensive information on the latency from data transfers between operators. Therefore, in Section 3.4, we present an approach to estimating the latency of network links in Apache Storm.

3.2. Design considerations

Within the work at hand, we focus on the stream operator placement problem (Section 2.3) for the above-defined system model considering the following:

- *Scheduling Units:* In our work, scheduling is done for a complete set of operators defined within a DAG. The alternative would have been to do the scheduling for each operator. While this would be way easier from the perspective of the computational difficulty of finding an optimal solution, taking only one operator into account at a time may also lead to local maxima instead of the envisioned global maximum.
- *Adaptivity:* Placement techniques can be classified as either static or adaptive (also referred to as offline and online techniques). Static or offline schedulers perform operator placement once, typically at deployment time, and do not adjust these placements during runtime. This approach is more straightforward but can lead to suboptimal performance if the system conditions change, for example, due to variations in the number of available workers. On the other

hand, adaptive or online schedulers continuously monitor the system's state and dynamically adjust operator placements in response to changing conditions, such as fluctuating workloads or resource availability. Given that static approaches may result in outdated placements as the system evolves, we focus on adaptive scheduling to maintain optimal performance under dynamic conditions in this work.

- *Quality of Service Constraints:* In addition to optimizing some metrics with the scoring function, Quality of Service (QoS) constraints can be defined to guarantee a minimum standard in all solutions. These are commonly focused on limiting the cost or ensuring most of the available computational resources are utilized instead of idling. In our work, we optimize the latency and throughput of stream processing applications.
- *Heterogeneity:* Heterogeneity considers the support of computing resources with varying properties or capabilities, such as their computing performance or available storage [21]. Since nodes at the edge of the network could be very heterogeneous, the hybrid heuristic supports this.
- *Heuristics:* As mentioned in Section 2.3, the stream operator placement problem is NP-hard. Accordingly, we aim to provide heuristics for solving it.

We reuse these aspects when comparing our work to the related work in Section 7.

3.3. Resource monitoring and adaptation

An important aspect that needs to be considered in the design phase of stream processing optimization approaches is the dynamic variation of computational and network loads that might impact performance [32]. This variation can be an outcome of, e.g., newly discovered available computational resources that alter the system's computational capacity or unexpected network bottlenecks. To account for such variations, monitoring the computational and network resources of the system is crucial. Especially in environments characterized by fluctuating workloads and resource availability [33], considering varying computational resource capacity and utilization can be essential for maintaining optimal system performance. In our work, we consider computational resource utilization-related metrics (such as CPU and memory) in the problem formulation, as mentioned in Section 4.2, and also in the system implementation, as discussed later on in Section 5. Such considerations are taken into account at runtime during every iteration of the scheduler, leading to scheduling decisions tailored to the available computational resources of the system.

In addition to monitoring and adapting to computational resource utilization, we also consider network-related metrics. Assuming a general-purpose stream processing deployment, e.g., over a local network and the Internet, we have no control over the underlying network infrastructure. Therefore, we cannot assume knowledge of concrete network capacities and variation, as the target network (including the Internet) can be unpredictable [34]. Nevertheless, basic network estimations are considered to aid in dynamic network adaptations. Since latency between operators is crucial, as also discussed in Section 1, we devise an elaborate approach to take latency measurements, as discussed in Section 3.4, which is integrated with the problem formulation (in Section 4.2), and implemented, as explained in Section 5. Furthermore, network metrics such as latency and the number of emitted events are explicitly considered in the problem formulation. Regarding monitoring such metrics in practice, we discuss appropriate techniques in Section 5.2.

Therefore, dynamic resource monitoring (considering various metrics and resources) and adaptations are performed frequently since the proposed online scheduler runs periodically. This guarantees that computational and network resource changes are taken into account, leading to optimized scheduling, which is the focus of our work. To target unreliable and lossy networks with highly fluctuating resources, additional mechanisms for dynamic adaptations can be additionally implemented [35,36].

Algorithm 1: Process to update a supervisor's position in the latency cost space.

Input: $\mathbf{p}_{est} = \mathbf{p}_{curr}$ (current coordinate), ϵ_{min} (minimum required movement)
Output: updated estimated position $\mathbf{p}_{est}$

```
1  P ← selectPeers(G)
2  updatePositions(P)
3  measureLatencies(P)
4  Δ_max ← ∞
5  k ← 0
6  while Δ_max > ε_min and k < 100 do
7      m ← 0
8      foreach p ∈ P do
9          f ← force(p, p_est)
10         m ← m + f
11     end
12     p_est ← p_est + m
13     Δ_max ← ||m||
14     k ← k + 1
15 end
16 return p_est
```

3.4. Network link latency estimation

To collect latency estimations, we apply spring relaxation. A spring-based estimation rather than a matrix completion is preferred because a spring-based estimation naturally leads to a highly decentralized design without the need for coordination. This design is inspired by the work of Pietzuch et al. [7] (see Section 7).

In the introduced hybrid heuristic, each supervisor independently estimates its coordinates in a three-dimensional cost space. In this space, the distance between two coordinates correlates to the estimated latency between the supervisors. Initially, each supervisor's coordinates are randomized and adjusted based on periodic latency measurements [7].

Algorithm 1 shows the process of executing a periodic latency measurement. When a supervisor can communicate with other supervisors, these are referred to as peers $\mathcal{P}$. $\mathcal{P}$ is assumed to be known in the beginning to bootstrap the algorithm. First, a supervisor adjusts its position by assuming the coordinates of other peers $p \in \mathcal{P}$ are fixed (Lines 1-3). Spring relaxation is then applied between the supervisor and its peers to pull the supervisor into a position closer to the measured latencies (Lines 8-12). This is repeated iteratively until the total movement across all peers Δ_{max} falls below the minimum required movement ϵ_{min} or a maximum number of iterations k is reached (Line 6) [7]. The estimated position $\mathbf{p}_{est}$ is then returned (Line 16). After this process, the supervisor saves $\mathbf{p}_{est}$ in a key-value store and waits until the periodic latency estimation starts again to refine the coordinates based on new measurements or updated positions of peers. The interval of the periodic estimation is slightly randomized to prevent peers from potentially being synchronized, which can lead to cyclic updating based on outdated measurements.

To calculate the spring force in Line 9 of Algorithm 1, we present Algorithm 2. First, a unit vector $\mathbf{d}$ of the distance between the supervisors is calculated (Lines 1, 3). The force is defined as $k * log(\|\mathbf{d}\|/\ell_p)$, with k being a constant, ℓ_p the distance using the network ping, and $\mathbf{d}$ the distance in the latency space (Line 2). This force is then applied in the correct direction by multiplying it with $\mathbf{d}$ (Line 4). The logarithmic scale is used to prevent the forces between very distant positions from becoming too large compared to smaller forces [37,38].

Similar to Google's landmarking implementation, not all supervisors but only a small subset are used as peers [39]. In contrast, n random peers are used rather than selecting specific peers. After an estimation,

Algorithm 2: Process to calculate the force based on spring relaxation.

Input: p (peer), $\mathbf{p}_{est}$ (estimated position)
Output: $\mathbf{d}$ = movement update vector

$\mathbf{d} \leftarrow \mathbf{p}_p - \mathbf{p}_{est}$
$f \leftarrow k \cdot \log\left(\frac{\|\mathbf{d}\|}{\ell_p}\right)$
$\mathbf{d} \leftarrow \frac{\mathbf{d}}{\|\mathbf{d}\|}$
return $\mathbf{d} \leftarrow \mathbf{d} \cdot f$
return $\mathbf{d}$;

only the i peers with the lowest latency are kept, and j new random peers are selected, so the full selection of n peers exists again. This specific selection with a bias for more local resources is made as the accuracy of latency estimation for the closest resources matters the most for the operator placement. This is because these resources are far more likely to interact with each other once a placement occurs. After all, they offer low latencies. A network-aware operator placement heuristic that aims to reduce latencies then aims to use these resources. In contrast, the accuracy of the global latency estimation is less critical because having more distant resources interact hinders the aim of achieving lower latencies. Still, the j random peers ensure that the global estimation is somewhat accurate, even if ideally less relevant.

The implementation details of the network latency estimation are presented in Section 5.

4. Optimization problem

In this section, we present our latency-aware stream operator placement approach. First, we define a simplification of the system model in Section 4.1. Afterward, Section 4.2 formulates an optimization problem tailored to the system model discussed in Section 3. Then, Section 4.3 presents heuristics for solving this problem efficiently.

4.1. Simplification

The first step in efficiently solving the placement problem is simplification. The separation of workers on a supervisor in Apache Storms architecture is primarily logical. Each worker has access to all the resources of a supervisor. Similarly, multiple workers are unnecessary to achieve concurrency because workers run multiple threads. The documentation of Apache Storm even recommends only using one worker per topology on a supervisor, as multiple workers would introduce additional overhead for the communication between processes [40]. With this performance consideration in mind and without any requirement to distinguish between workers, there is no need to model each worker in the stream operator placement. Instead, operators can be assigned to a supervisor if one of its workers is already occupied by the same topology or at least one worker is still free. This greatly simplifies finding assignments and co-locating operators. Following this simplification, the actual optimization problem can be defined starting with the scoring function.

4.2. Formulation

The placement problem is formulated as a cost-minimization problem. The cost function is shown in Equation (1). This equation consists of four parameters with weights w_1 to w_4, which regulate the amount of influence of each parameter.

$$s(x) = w_1 * s_{lat}(x) + w_2 * s_{sup}(x) + w_3 * s_{co}(x) + w_4 * s_{event}(x) \quad (1)$$

$s_{lat}(x)$ is the highest estimated network latency accumulated during the processing of an event in the topology T. $T_{sources}$ and T_{sinks} denote all data sources and sinks of T, respectively. For any operator $s \in T_{sources}$, we define that $s_{lat}(s) = 0$. For the other operators in the topology, applies that $\forall o \in T : s_{lat}(o) = max(s_{lat}(p) + l_{p,o} : p \in o_{predeccesors})$ with $l_{a,b}$ being the estimated network link latency from the operator a to b. The latency score of the topology is the maximum among all sinks: $s_{lat}(T) = max(s_{lat}(s) : s \in T_{sinks})$.

$s_{sup}(x)$ is used to condense the placement of operators and is defined as $s_{sup}(x) = \frac{|supervisors \in x|}{|supervisors|}$. This parameter aims to reduce the total number of employed supervisors, allowing the temporary shutdown of redundant supervisors. This reduces unused resources.

$s_{co}(x)$ is a measure of the co-location of operators. If $p(o)$ is the placement of an operator o then $s_{co}(x) = \frac{|e_{o_1,o_2} \in T : p(o_1) \neq p(o_2)|}{|e_{o_1,o_2} \in T|}$.

Finally, $s_{event}(x)$ counts the events emitted over not co-located edges. With $t(e)$ being the events emitted on an edge e, $s_{event}(x)$ can be defined as $s_{event}(x) = \frac{\sum_{e_{o_1,o_2} \in T : p(o_1) \neq p(o_2)} t(e_{o_1,o_2})}{\sum_{e_{o_1,o_2} \in T} t(e_{o_1,o_2})}$. In practice, $t(e)$ might not be measured exactly because most stream processing frameworks (incl. Apache Storm) only have a metric about the emitted events for each operator. Nevertheless, $t(e)$ can be approximated by dividing the operator-based statistic by the count of outgoing edges. Thus, $s_{co}(x)$ relates to the general performance while $s_{event}(x)$ focuses on important operators being co-located.

All parameters have a range of $[0, 1]$ except for the latency $s_{lat}(x)$ that has a range of $[0, \infty]$. To account for this range, w_1 is set to $\frac{1}{1000000}$ so that it becomes insignificant and acts mostly as a deciding factor between similarly scored solutions. All the other weights, w_2 to w_4, are set to 1.

Regarding constraints, every operator is assigned to one supervisor, while each supervisor can have operators of one topology. Furthermore, the memory and CPU usage of the supervisors cannot exceed the provided capacities. Overall, the cost function favors co-located placements, while the constraints ensure the efficient use of the available computational resources. Hence, the utilization of a supervisor is allowed to reach up to 95% before it is considered a constraint violation.

4.3. Heuristics

We design three heuristics to solve the previously defined optimization problem: hill-climbing, ant system, and hybrid. All heuristics aim at using computational resources sparingly so that they can be executed iteratively for online scheduling. Thus, instead of striving to find an optimal solution (that may be too computationally intensive), our heuristics aim at approximating the optimal solution, which is preferred for real-time stream processing. To further reduce the computational overhead of the heuristics, a topology is rescheduled periodically, e.g., every 30 seconds, and only if there have been system updates.

4.3.1. Hill-climbing

Local search is based on performing small modifications on an existing solution iteratively to find better placements [41]. The initial solution for the search is either the current placement or a potentially highly inefficient placement from a greedy heuristic that mainly aims to satisfy the memory and slot availability constraints. In this local search implementation, for a modification to be accepted as a better solution, the amount of constraint violations has to be reduced or at least kept equal while improving the score. This is also often referred to as hill-climbing [41]. This means that a suboptimal placement will never be selected. Consequently, this heuristic might get stuck in local optima. Hill-climbing uses the following three operations to modify a placement:

1. Moving one operator to a different supervisor.
2. Swapping the placement of two operators which are not co-located.
3. Moving all operators of a supervisor to a different supervisor.

In theory, swapping and moving all operators is unnecessary because they can be expressed through multiple individual move operations. Due to co-location being highly relevant in multiple metrics of the scoring mechanisms, it is also often the cause of local optima. Swapping the

placements is highly effective at reducing the application's latency while moving all operators, which allows for the contraction of placements when a topology is experiencing a low load. In both cases, the loss of co-location would result in significantly worse scores if these modifications were to be executed in multiple steps instead, resulting in significant and undesirable local optima, which this form of local search would not overcome.

4.3.2. Ant system

This heuristic is based on the real-world collaborative pathfinding of ants [42]. A placement is represented by a path (in a graph) that starts at an operator, ends at a supervisor, and alternates between supervisors and operators until all operators are included in the path exactly once. Edges from the operators to the supervisors represent individual operator assignments. When the same edges are frequently chosen, they are given priority in future assignments based on a calculated pheromone. Notably, the order of the operators can affect the placement since the first operators are preferred. This can lead to local optima. To avoid these, the operator order is randomized for every placement, introducing a stochastic element that allows the algorithm to explore diverse solutions and avoid premature convergence.

Initially, the pheromone of each edge (i, j) is initialized with a configurable parameter p_i. A previous placement can also affect the initial pheromones by executing the pheromone placement p_c times for a path equivalent to the current placement. Then, m ants are generated, each with a random path. The best ant is updated, and the pheromones on the graph edges are modified accordingly. The iteration count and the number of iterations since the last score are used as exit conditions. When the algorithm ends, the best placement is returned.

The probability of an ant k at time t to move from a node i in the graph to j is shown in Equation (2). It consists of an a priori heuristic, for which a custom one is defined in Equation (3), and an a posteriori heuristic, i.e., the pheromone. α and β are used as parameters to weigh the importance of both elements [42].

$$p_{ij}^{k} = \begin{cases} \frac{|\mathcal{T}_{ij}(t)^{\alpha}| \cdot |\eta_{ij}(t)^{\beta}|}{\sum_{k \in \text{allowed movements}} |\mathcal{T}_{ik}(t)^{\alpha}| \cdot |\eta_{ik}(t)^{\beta}|} & j \in \text{allowed movements} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

The a priori heuristic in Equation (3) is used to find solutions with more co-locations of operators. To achieve co-location, the output value of this heuristic is multiplied by 2 if another operator of the topology is placed on the supervisor or if a predecessor or successor operator is placed on the supervisor. If the placement overloads the resources, the value is divided by 2. This way, the pheromone encourages or discourages certain decisions, guiding the exploration.

$$\eta_{ij} = 1 \cdot 2^{\text{pre or suc on } j} \cdot 2^{j \text{ already used}} \cdot \frac{1}{2^{\text{overloads } j\text{'s CPU}}} \cdot \frac{1}{2^{\text{overloads } j\text{'s mem}}} \tag{3}$$

Equation (4) shows the decay of the pheromone, which is based on the parameter ρ and the newly placed pheromone of all ants for an edge (i, j) [42]. In this ant system variant, a minimum pheromone amount p_m on each edge is enforced to ensure that the random exploration of alternate paths never stops [43].

$$\mathcal{T}_{ij}(t) = \max(\rho \cdot \mathcal{T}_{ij}(t-1) + \sum_{k=1}^{m} \Delta\mathcal{T}_{ij}^{k}, p_m) \tag{4}$$

The pheromone placed on an edge (i, j) by the ant k is defined in Equation (5). This equation uses a constant Q to scale the placed pheromone and S_k as the score of the ant k's solution. This ensures that a lower score results in more pheromones placed, attracting more ants to better solutions [42].

$$\Delta\mathcal{T}_{ij}^{k} = \begin{cases} \frac{Q}{S_k} & k^{th} \text{ ant uses edge (i,j) in path} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

To ensure that constraint violations are reduced, S_k is defined using a combined score, i.e., $S_k = score(k) + 10000 * constraintViolations(k)$.

4.3.3. Hybrid

The hybrid heuristic combines the benefits of hill-climbing and the ant system. The random solution construction of the ant system is more effective at exploring the solution space by avoiding getting stuck in local minima but also does not identify and exploit the smaller optimizations as greedily as hill-climbing. Hence, after the ant system finds a good solution, there may still be some small adaptions that can be made to improve the score greedily. For this reason, the hybrid search consists of running the ant system for its effective exploration, followed by hill-climbing to find and make those small adjustments. The execution time is halved for each heuristic to ensure that the execution of both heuristics in succession still fulfills the same design constraints. The idea of combining local search and ant systems is well known. However, it is usually directly applied within the ant system by optimizing each ant's path again before the pheromone is placed [41].

5. Implementation

This section discusses the details of the implementation of our Apache Storm-based prototype. In Section 5.1, we give an overview of the architecture. Afterward, Section 5.2 discusses the implementation of the metrics consumer, and Section 5.3 elaborates on the realization of the network latency estimation.

5.1. Architecture

As discussed, the presented approach is, per se, applicable to different data stream processing frameworks. Naturally, to integrate it into specific frameworks, some adaptations are necessary. In the following subsections, we describe how this has been done in the work at hand for Apache Storm. More precisely, the implementation has been done for Apache Storm 2.4.0 since this was the most recent version of the framework at the time of development. An overview of the general architecture can be found in Fig. 1. Our prototype is available at BitBucket[4] under the Apache 2.0 open-source license.

Naturally, we allow different hosts in the compute continuum to be used to deploy workers in Apache Storm. Each host runs a supervisor (Section 3), which controls the worker processes. The *Storm Metrics Consumer* is used to forward the metrics of a topology.

To access the necessary metrics for our placement heuristic, they have to be made accessible to Nimbus. Nimbus is a daemon running on the Apache Storm master node that distributes code in a Storm cluster, assigns tasks to worker nodes, and provides monitoring capabilities. To achieve this, the necessary metrics are sent to ZooKeeper, where they are accessible to the placement heuristic running in Nimbus.

In addition, we add a latency estimation component. It is installed and executed along with any supervisor but operates independently of them. To transfer this data to Nimbus and exchange data to compute the estimations, Redis, an in-memory key-value store, is used [44].

Apache Storm provides two Java interfaces to implement a placement logic. The *IStrategy* interface allows the creation of a new placement logic for Storm's standard *Resource Aware Scheduler* [28]. Still, it is limited to applying the placement at topology deployment time or if other external changes occur. Hence, we selected the *IScheduler* interface, which is called periodic and allows the implementation of an online scheduler.

As shown in Fig. 1, in addition to Storm's regular communication between Nimbus, ZooKeeper, and the supervisors, there are three additional flows of data related to the operator placement heuristic executed on Nimbus. When the *Storm Metrics Consumer* receives metric data from

[4] https://bitbucket.org/RaphaelEcker/dsgexposeoperatorplacement.

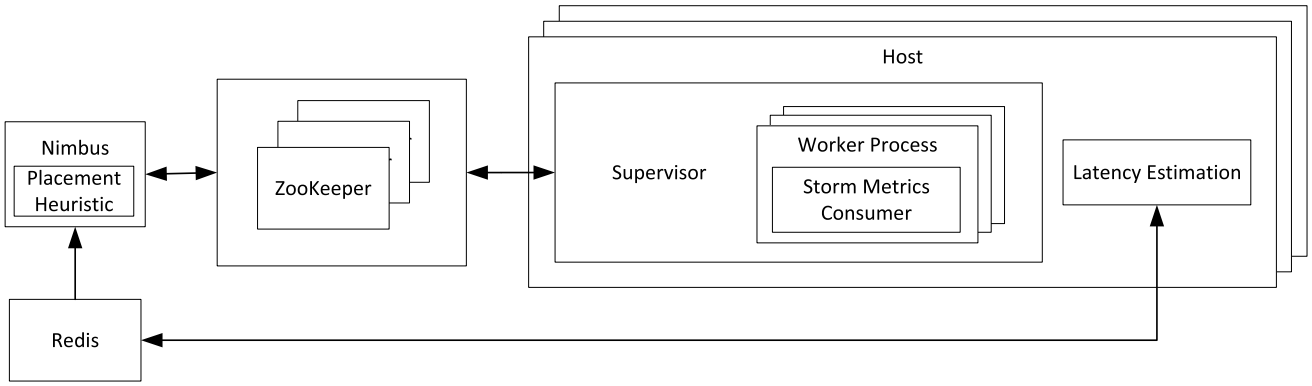


Fig. 1. Extended Architecture of Apache Storm.

operators, it is written to ZooKeeper. Additionally, the latency estimators running on each host retrieve the list of potential peers and their positions from Redis. They use this data to adjust their position estimation and store it on Redis. Finally, the placement heuristic on Nimbus only retrieves the data from ZooKeeper and Redis when a placement occurs, resulting in one-directional data flows. These three data flows, therefore, do not require synchronization, and the components do not directly interact with each other since Redis and ZooKeeper store the data until it is used.

One advantage of this architecture is that it allows for the implementation of the placement heuristic without requiring any modification of Apache Storm. All the necessary functionality already exists in Storm and can be extended by well-defined interfaces and configured or added externally as another component.

5.2. Metrics consumer

This section describes how to acquire access to resource metrics of Apache Storm operators, which is critical for facilitating dynamic monitoring and adaptation, as discussed in Section 3.3. For example, computational resource metrics, such as CPU utilization, can provide insights into the computational load of each node, allowing the detection of nodes nearing their processing capacity and enabling operator placement adaptation. Similar network metrics, such as bandwidth measurements, are essential to understanding the data transfer rate, affecting the overall system throughput and the efficiency of the communication links between operators. Therefore, such metrics can aid in optimized scheduling decisions and maintain performance in the face of fluctuating workloads and resource demands.

Interestingly, Apache Storm does not directly provide access to the metrics of a topology, which is why the *MetricsConsumer* is used to forward that data. Alternatively, Storm UI exposes the metrics with a Representational State Transfer (REST) API, but it would have to be repeatedly queried for potential metric updates rather than them being pushed [45]. In the end, the *MetricsConsumer*-based implementation also relies on Nimbus polling the data because the actual metrics are stored in ZooKeeper as described in the architecture. As a result, both implementation variants of accessing the metrics are highly similar, as they require the use of optional Storm components in their deployment, be it Storm UI or the *MetricsConsumer*.

The metrics collections interval has been reconfigured for online scheduling to ten seconds to match Storm's scheduling interval. Once the bandwidth usage of an operator has been measured, it can be used to calculate an average event size, such that the bandwidth usage can then be estimated even if the operator is co-located. Because of the difficulty of reliably acquiring up-to-date bandwidth information and the need for an initially bad placement, the decision was made to approximate bandwidth usage purely by considering the number of emitted events, as presented in the design.

Another limitation of Apache Storm is that, by default, the CPU usage is not monitored. Instead, Apache Storm provides the capacity metric, which is the percentage of time an operator is not idle. Considering that best practices avoid implementing operators with blocking functions, this directly means that capacity can signify some resource bottleneck. The cause of this bottleneck could be another resource, such as the storage or network, but in the most general case, it describes the CPU usage or its idle time. For this reason, the scoring function and constraints from the optimization problem definition are implemented using the capacity metric rather than the actual CPU usage. Capacity is, of course, a metric that is only relevant to the current computational resource. It is, therefore, transformed into a general CPU cost to support heterogeneous resources. The operator's capacity is multiplied by the supervisor's CPU capacity to compute the operator's CPU cost. This achieves point costs that can be considered in conjunction with the manually defined point-based capacities of supervisors using the existing functionality of Storm's Resource Aware Scheduler [28,46]. The manual supervisor CPU capacity configuration could be avoided by executing a benchmark on the start-up of the service to measure it directly instead.

5.3. Network latency estimation

The network latency estimation implementation follows the design and architecture considerations already made. Redis acts as a store for each supervisor's estimated positions and as a lookup table to find peers. The component is implemented using Java and uses the natively installed ping utility, if available, for increased accuracy in collecting latencies. To ensure the latencies of the estimation can be mapped to the supervisors, the same method to generate the supervisor's ID in Apache Storm is essentially reused. With both the latency estimation and Apache Storm identifying supervisors by their hostname, matching the data becomes trivial.

A simple method to estimate the latency of a stream processing application would be to sum or average all the individual latencies. While this can help identify better placements, it completely ignores how the latency is distributed across the application. As such, some paths could have very short or excessively long latency. For this reason, the latency of an application has been previously defined as the highest latency of any path through the DAG, i.e., the stream processing application.

An efficient method to calculate this is to set the latency to reach an operator to the maximum of all its predecessors and the network link latency to the current operator. By applying this iteratively to all operators once their predecessors have been calculated, the latency experienced at the sinks can be calculated and, therefore, be used to estimate a complete application's latency. The order in which the latency of the operators can be iteratively calculated stays identical unless the DAG is changed. As such, this order can be calculated once rather than being recomputed for every latency estimation. Furthermore, this order is a topological order for which well-known sorting algorithms exist, e.g., Kahn's algorithm [48]. Unfortunately, this only works for DAGs, but Apache Storm

Algorithm 3: Modification of the Eades, Lin, and Smyth heuristic to create topological orderings of cyclic graphs [47].

```
Input: G = (V, E) (directed, potentially cyclic graph)
Output: sets of vertices in topological order after removing minimal
        edges to break cycles
1  L, R ← empty stack
2  Q_src, Q_sink, Q_tmp ← empty queue
3  enqueue(sources(G), Q_src)
4  enqueue(sinks(G), Q_sink)
5  while G ≠ ∅ do
6      C ← ∅
7      while Q_src ≠ ∅ do
8          v ← dequeue(Q_src)
9          C ← C ∪ {v}
10         G ← G \ {v, edges of v}
11         foreach v' ∈ sources(G) do
12             enqueue(v', Q_tmp)
13         end
14         if Q_src = ∅ then
15             swap(Q_tmp, Q_src)
16             if C ≠ ∅ then
17                 push(C, L)
18                 C ← ∅
19             end
20         end
21     end
22     while Q_sink ≠ ∅ do
23         v ← dequeue(Q_sink)
24         C ← C ∪ {v}
25         G ← G \ {v, edges of v}
26         foreach v' ∈ sinks(G) do
27             enqueue(v', Q_tmp)
28         end
29         if Q_sink = ∅ then
30             swap(Q_tmp, Q_sink)
31             if C ≠ ∅ then
32                 push(C, R)
33                 C ← ∅
34             end
35         end
36     end
37     if G ≠ ∅ then
38         C_comp ← tarjan(G)
39         v ← maxVertexWithRemovedNeighbor(C_comp)
40         G ← G \ {v, edges of v}
41         C ← {v}
42         push(C, L)
43     end
44 end
45 while R ≠ ∅ do
46     C ← pop(R)
47     push(C, L)
48 end
49 return L
```

theoretically allows for the definition of cycles in a topology. Hence, our implementation needs to transform a cyclic directed graph into a DAG.

Removing the minimal number of edges from a cyclic directed graph to transform it into a DAG is known as the minimum feedback arc set problem. The heuristic by Eades, Lin, and Smyth was implemented to accomplish this aim because of its speed, simplicity, and capability to directly output a resulting topological ordering [47]. Unfortunately, the minimal feedback arc set problem only considers removing the minimum number of edges and not how this affects the pattern or structure of the graph. This does not match the goal of calculating metrics such as latency, for which the impact on these calculations should be minimized. Therefore, the selection of the node and, in turn, edges must be adapted when cycles are being resolved.

We also need to account for multiple cycles in a topology. Primarily, the heuristic should only remove necessary edges to eliminate cycles. Removing some cycles may allow additional vertices to be iteratively processed rather than requiring their out-of-order removal. This means an optimal order of which cycle to remove first needs to be found. So far, only cycles have been considered for their simplicity, but any more complex structure consisting of multiple cyclic elements must also be simplified. The important concept for such cyclic structures is that any vertex has a path to any other vertex, generally known as a strongly connected component [49]. This is also the problem for calculating latency because none of the structure's vertices can be considered the first or last element. As such, a mechanism for detecting and resolving strongly connected components is necessary.

Strongly connected components are well studied, and as such, algorithms to find them efficiently exist. In this work, we use Tarjan's algorithm [49] based on depth-first search and a runtime of $O(|V|+|E|)$ to not only identify all strongly connected components in a directed graph but also return them in inverse topological order. The strongly connected components can be returned in a topological order, even in a cyclic graph, because the strongly connected components can be used to transform the graph into a DAG.

The aforementioned algorithms provide the necessary prerequisites to create a heuristic for calculating a topological ordering for potentially cyclic graphs, which in turn allows calculating the latency of the stream processing application, although with only limited correctness. The complete pseudocode of the modified heuristic can be seen in Algorithm 3. It consists of the Eades, Lyn, and Smith heuristic (Lines 1-37, 40-49), modified to remove all viable sources or sinks as a set rather than individually (Lines 7-21, 22-36). Additionally, the logic of selecting a vertex to resolve a cycle was modified as described above. Tarjan's algorithm (Line 38) is applied to identify the first cycle in the DAG, which is the last component in the algorithm's output. Finally, a vertex in this cycle is selected for removal based on Eades, Lin, and Smyth's heuristic, requiring it to neighbor a previously removed vertex categorized as a source (Line 39).

5.4. Topology adaptation

Some existing stream processing frameworks, such as Apache Flink, may allow the automatic adaptation of the topology, e.g., by merging and splitting operators and using dynamic scaling [50,51]. Such actions can affect the utilization of the available resources, especially in environments with fluctuating workloads (this is also discussed in Section 3.3).

In this work, focusing on optimized scheduling, we consider various similar aspects, e.g., we prioritize the co-location of operators to optimize resource utilization, as discussed in Section 4.2, and we detect operators reaching the resource capacity limits and relocate them appropriately. Such conditions could also trigger topology adaptations, e.g., by merging co-locating operators performing the same processing tasks with a low processing rate or splitting operators reaching resource capacity. This may reduce the computational overhead.

However, although some stream processing frameworks like Apache Flink offer built-in support for topology adaptation, including merging and splitting, Apache Storm, the framework chosen for our implementation, does not support these features. Nevertheless, Apache Storm offers appropriate APIs for implementing custom scheduling approaches, which makes it a compelling candidate for our implementation. In this work, focusing on making optimized scheduling decisions, we do not consider topology adaptations, as this could be potentially handled natively by additional mechanisms offered by the host stream processing framework. Notably, during the experiments of Section 6, aiming to compare the proposed scheduling with alternatives, none of the examined approaches allows dynamic topology adaptations at runtime, thereby ensuring comparable evaluation results highlighting the bene-

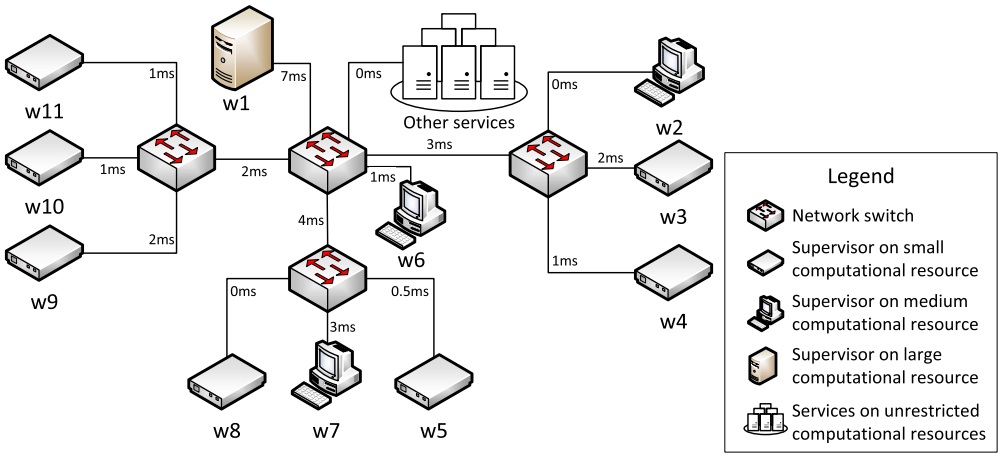


Fig. 2. Emulated network topology for the evaluation.

fits of the optimized scheduling decisions, which is the focus of the work at hand.

6. Evaluation

For the evaluation, we build a system for stream operator placement and run various experiments to compare the single heuristics and the hybrid heuristic with existing alternatives. We describe all the implemented approaches in the experimental setup in Section 6.1 and an elaborate description of the results of our experiments in Section 6.2.

6.1. Experimental setup

We experimented with various implemented approaches for this evaluation, which we integrated into Apache Storm. In total, we implement five scheduling approaches, namely, hill-climbing, ant system, the proposed hybrid, the default scheduler of Apache Storm, and the resource-aware scheduler [28,46,52]. These five specific approaches are motivated by their diverse strengths and trade-offs, offering a comprehensive view of the performance of stream operator placement strategies. Specifically, hill-climbing is chosen for its simplicity and efficiency in exploring local optimizations, providing a baseline for understanding how simple iterative improvements impact performance. The ant system is implemented due to its ability to explore a broader search space through collaborative pathfinding, offering insights into how stochastic optimization techniques can handle operator placement. The proposed hybrid approach, combining hill climbing and the ant system, is implemented to evaluate whether local and global search strategies together can yield better results than either alone, especially in the context of stream processing topologies.

Including the default and resource-aware schedulers is essential for benchmarking against existing, well-established methods. The default scheduler represents a basic, round-robin placement, and by comparing its results with heuristic-based approaches, we aim to highlight the potential performance benefits. On the other hand, the resource-aware scheduler incorporates resource constraints and is designed to optimize resource usage and reduce latency, making it an important benchmark for evaluating how our heuristic-based solutions compare to a more sophisticated baseline. Thus, by comparing these two approaches, we aim to evaluate the performance of our custom approaches compared to widely used alternatives. Overall, we consider that these five optimization alternatives are sufficient for investigating the behavior of the implemented approaches when making operator placement decisions. Additional aspects of the evaluation setup, including network configurations, test data, and data sources, are discussed below.

6.1.1. Network emulation

Since the compute continuum is a novel computing model, relevant environments using end devices, network, edge, and cloud resources are not readily available. For this reason, we have to rely on emulations to create a realistic network topology for this evaluation.

Fig. 2 shows our emulated network topology, including the link latencies. We use a switch-based network topology, usually used for smaller to medium-sized networks, such as a single factory or office building, which could have a private deployment of a compute continuum including cloud and edge resources. This topology includes workers $w1$ to $w11$, which are the Apache Storm supervisors running in Docker containers. The symbol *other services* in Fig. 2 represents ancillary services required for running the experiments (such as Nimbus, Zookeeper, and Redis). The resource capacities that are available for each supervisor are as follows: $w1$ has 1.5 logical cores of CPU and 2048 Megabytes (MB) of memory, $w2, w6$ and $w7$ have 1 logical core of CPU and 1024 MB of memory, and $w3$ to $w5$, as well as $w8$ to $w11$, have 0.3 logical cores of CPU and 768 MB of memory.

Notably, the resource capacities of the supervisors vary to represent different devices used in the compute continuum. The link latencies also vary to represent the distance of distributed computational resources. We use link latencies in the low single-digit milliseconds range since users of edge servers usually experience latencies below 10 milliseconds [53]. The network emulation is executed on a machine equipped with an Intel i7-4770k 3.5Ghz, four CPU cores, and eight threads using 16 GB DDR3 memory with 1600MHz. The machine uses Ubuntu 20.04.4 LTS as the operating system with version 2.4.0 of Apache Storm. Finally, we use Containernet to run the containers as hosts in the topology and the Quagga routing suite for routing [54].

6.1.2. Test data

Twelve different topologies are used for the evaluation and are listed in Table 1 for an overview. The topologies consist of three manually defined topologies (see Fig. 3) to test edge cases, seven randomly generated topologies (see Fig. 4) to evaluate the general cases, and two additional random topologies containing a cycle (see Fig. 5) to account for this special case. Further, each topology has only a single source and sink operation to allow for an easier comparison during benchmarking. Otherwise, multiple input and output rates would have to be considered for each topology. This is a common approach to test operator placement heuristics [22,10,55,56].

The topology M1 is merely a sequence of operations. M2 is a single operation for which ten instances exist and thereby tests a fan out of data and the load balancing of the individual instances. M3 represents the minimal possible topology with a single operation as a workload with only one instance.

Table 1
Overview of the main properties of the topologies being tested. The operations and operators include the data source and sink in their count.

Topology	Randomly Generated	Contains a Cycle	Operations	Operators
T-M1	✗	✗	6	6
T-M2	✗	✗	3	12
T-M3	✗	✗	3	3
T-R1	✓	✗	16	30
T-R2	✓	✗	7	12
T-R3	✓	✗	12	22
T-R4	✓	✗	15	28
T-R5	✓	✗	11	20
T-R6	✓	✗	11	20
T-R7	✓	✗	22	42
T-R8	✓	✓	8	14
T-R9	✓	✓	10	18

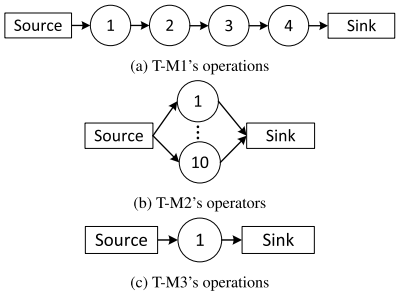


Fig. 3. Visualisations of manually defined topologies' operations or its instances in the special case of T-M2.

For the random topologies, Java's pseudorandom number generator *Random* has been seeded with the first 19 digits of π. Further, an additional $(n-1) * 1000$ random numbers are generated for the n^{th} topology to ensure a different random state for each topology.

The first step of generating a random topology is to select a random operation count between three and 24 with a discrete uniform distribution. The DAG generation starts with a single source and sink operation connected by an edge. Afterward, one of the edges is randomly selected, and either one operation or two, in the case of the diamond pattern, are inserted and replaced with the original edge. The single operation insertion is generated with a 60% probability and the diamond pattern with a 40% probability because equal probabilities tend to create too wide graphs.

Additionally, Apache Storm sends emitted events to all following operations, and, as such, a split is always a duplication of the output rate. Hence, we consider the selectivity of the individual operations to ensure that some random topologies are generated where some output more and others receive fewer events than they receive as input, representing both kinds in the evaluation. Additionally, the selectivity of the individual operations is essential to ensure the generation of applications that output more or fewer events than they receive as input. The topology is then iteratively grown until it reaches the exact operation count.

6.1.3. Test driver

The test driver is a separate component that generates input data for the stream processing application. Each input event, including generated data, contains a timestamp, which is used to track latency. During the processing of the topology, this timestamp is copied unchanged throughout the topology, and is included in the output. When the test driver receives an output tuple, it compares the timestamp of the output with the current time to compute the latency. In addition to latency, the test driver records metrics such as throughput, computational resource utilization, and scheduling information acquired from Redis.

The test driver operates with four threads: one for generating data, two separate threads for sending and receiving events to and from the

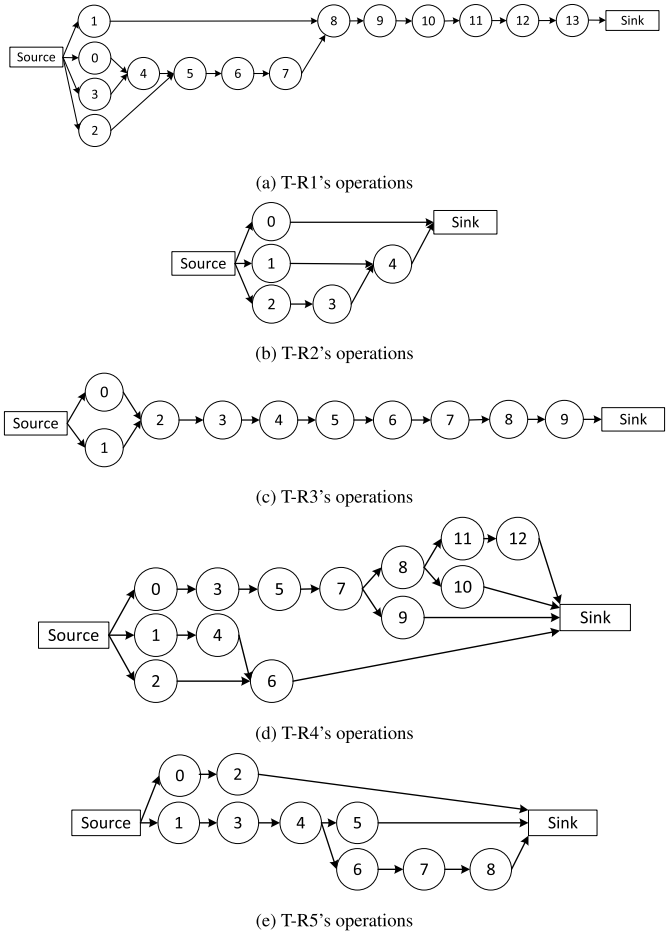


Fig. 4. Visualisations of the randomly generated topologies' operations.

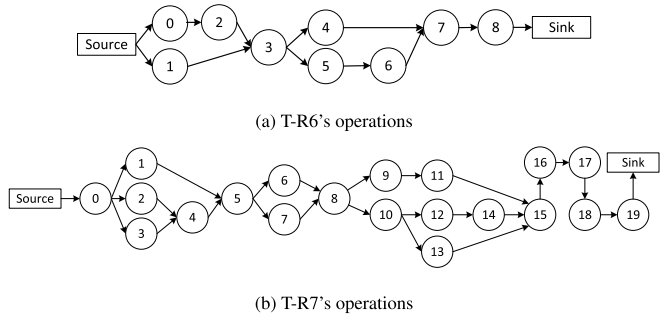


Fig. 5. Visualisations of the randomly generated topologies' operations (Cntd.).

Table 2
Average throughput factor increase over the default scheduler at the same CPU reservation for each topology without a cycle and their operator count.

Topology	Ops	Hill-Climbing	Ant System	Hybrid	Resource Aware Scheduler
T-M1	6	3.35 ($\sigma = 1.37$)	4.10 ($\sigma = 0.29$)	4.15 ($\sigma = 0.18$)	3.45 ($\sigma = 0.00$)
T-M2	12	4.32 ($\sigma = 4.08$)	2.68 ($\sigma = 3.28$)	3.19 ($\sigma = 2.03$)	1.94 ($\sigma = 0.00$)
T-M3	3	7.24 ($\sigma = 2.55$)	5.28 ($\sigma = 4.36$)	8.46 ($\sigma = 0.00$)	10.44 ($\sigma = 0.00$)
T-R1	30	1.79 ($\sigma = 0.62$)	1.70 ($\sigma = 0.67$)	1.68 ($\sigma = 0.64$)	1.81 ($\sigma = 0.55$)
T-R2	12	2.19 ($\sigma = 1.03$)	2.19 ($\sigma = 0.95$)	2.08 ($\sigma = 0.97$)	2.91 ($\sigma = 0.00$)
T-R3	22	2.04 ($\sigma = 0.58$)	1.90 ($\sigma = 0.50$)	2.24 ($\sigma = 0.60$)	2.37 ($\sigma = 0.60$)
T-R4	28	2.60 ($\sigma = 1.01$)	2.07 ($\sigma = 1.24$)	2.06 ($\sigma = 0.59$)	1.80 ($\sigma = 0.91$)
T-R5	20	2.10 ($\sigma = 1.02$)	1.51 ($\sigma = 0.97$)	2.15 ($\sigma = 0.90$)	2.13 ($\sigma = 1.30$)
T-R6	20	1.80 ($\sigma = 0.46$)	1.71 ($\sigma = 0.71$)	1.95 ($\sigma = 0.70$)	2.09 ($\sigma = 0.32$)
T-R7	42	2.08 ($\sigma = 0.72$)	1.75 ($\sigma = 0.57$)	1.92 ($\sigma = 1.00$)	1.63 ($\sigma = 0.79$)

stream processing application via sockets, and one dedicated thread to aggregate and output the recorded metrics.

Each experiment generates a constant throughput rate, followed by measuring the latency and throughput of the processing. This is sufficient to ensure the application has processed all events and supplied a constant rate of input events for measuring latency and throughput. For the static schedulers, such as the default scheduler and the Resource-Aware Scheduler, operators are placed once at topology submission. In contrast, for the adaptive schedulers, such as hill-climbing, ant system, and hybrid, the placement of operators is repeated three times to account for their ability to adjust during runtime. The evaluation begins with a low constant input rate by generating a queue of 5,000 tuples, followed by an increased input rate with a queue of 10,000 tuples to test the scale-out behavior, and finally, a return to the initial input rate to observe the scale-in performance. The adaptive schedulers are given two minutes to adjust the operator placement for every input, except during the scale-out phase, where an extended ten-minute period is used to test the stability of the placements. It is critical to ensure that Storm has completed all placement modifications and that the application is fully functional to acquire accurate measurements of the performance metrics. For the same reason, the test driver waits until all events are processed before measuring throughput, ensuring no queued events are pending before transitioning to the following constant input rate stage.

6.2. Experimental results

In this section, we examine the results of our experiments. Initially, we discuss the observed throughput and examine the runtime of computing single placements. Finally, we select some topologies to discuss throughput and latency in detail.

6.2.1. Throughput

In our experiments, we measure the maximum throughput using the test topologies. Additionally, we examine the maximum sustainable throughput, which describes the highest possible throughput at which the service can offer consistently low latency.

Table 2 provides a brief overview of the observed general throughput. We calculate the average throughput of all placements created by one of the heuristics at a specific resource usage. We then put it into relation to the default scheduler to calculate the factor by which the throughput has been improved. If a direct reference throughput does not exist, linear interpolation or the closest value is used if it is outside the range of results. By averaging all these throughput improvement factors, an overview of the performance of a scheduler on a specific topology can be provided.

First, there is a stark difference between the manually defined topologies (T-M1–T-M3) and the randomly generated ones (T-R1–T-R9). In general, the manually defined topologies are much smaller, and, as such, the Resource Aware Scheduler places them in all configurations on the largest computational resource, resulting in effectively a single placement. In contrast, the adaptive placement heuristics are more varied, as they also consider scaling down to a less capable resource. Additionally, larger throughput increases relative to the default scheduler are generally measured because all other schedulers optimize for co-location and are aware of the heterogeneous operator requirements and resource capabilities.

When comparing the solutions on the random topologies, a trend can be observed in which the Resource Aware Scheduler (average $\mu = 2.11$) generally outperforms hill-climbing (average $\mu = 2.09$), followed by the hybrid approach (average $\mu = 2.01$), and finally the ant system (average $\mu = 1.83$). These values represent the average of the throughput improvement factors shown in Table 2, specifically for the random topologies (i.e., T-R1 to T-R7). Additionally, the average standard deviations of our proposed heuristics are quite similar overall, indicating consistency across the experiments. Another observable trend is that our heuristics perform better on topologies with a larger number of operators, such as T-R4 and T-R7 while showing comparable performance on T-R1 and T-R5. Conversely, the Resource Aware Scheduler achieves some significant gains on smaller topologies. This is because, on smaller topologies, fewer operators compete for resources, which increases the potential for co-location and results in higher throughput.

A potential explanation for this is the optimizations of Apache Storm's ackers, which are a part of the optional guaranteed message processing mechanism. The proposed solutions handle them like all the other operators, while the Resource Aware Scheduler and default scheduler spread them evenly. This behavior can lead to not all operators being co-located with ackers. In higher throughput situations, this could then create additional messaging overhead.

Table 3 shows a head-to-head comparison of placements in relation to the Resource Aware Scheduler, where identical amounts of resources are allocated. The results show a similar pattern, with the proposed solutions performing better on larger topologies and highlighting some placements in T-R7 where the Resource Aware Scheduler performed unexpectedly worse, with even the default scheduler achieving higher throughput. Additionally, it shows a more consistent performance for the ant system and hybrid approach relative to the Resource Aware Scheduler's placements. On average, hill-climbing ($\mu = 1.11$) and hybrid search ($\mu = 1.03$) outperform the Resource Aware Scheduler in this comparison, and the ant system achieves a lower average relative performance of 0.89.

If similar comparisons are made with the maximum sustainable throughput, then hybrid search ($\mu = 8.46$) outperforms the Resource Aware Scheduler ($\mu = 7.76$), followed by hill-climbing ($\mu = 7.64$) and the ant system ($\mu = 7.56$). In this comparison, the average standard deviations are particularly high, close to the actual averages for all solutions, indicating inconsistent maximum sustainable throughput for the default scheduler that forms the basis of this comparison. In the head-to-head comparison to the Resource Aware Scheduler, this throughput ordering repeats, but with much closer results: hybrid search ($\mu = 1.01$) outper-

Table 3
Average throughput factor increase over the Resource Aware Scheduler at the same CPU reservation for each topology without a cycle and their operator count. Some entries are empty because no direct equivalent placement to compare is available.

Topology	Ops	Hill-Climbing	Ant System	Hybrid	Default Scheduler
T-M1	6	1.45 ($\sigma = 0.00$)	1.15 ($\sigma = 0.00$)		0.29 ($\sigma = 0.00$)
T-M2	12	0.89 ($\sigma = 0.00$)		0.73 ($\sigma = 0.00$)	
T-M3	3	0.89 ($\sigma = 0.00$)			0.10 ($\sigma = 0.00$)
T-R1	30	0.98 ($\sigma = 0.24$)	1.08 ($\sigma = 0.02$)	1.14 ($\sigma = 0.00$)	0.70 ($\sigma = 0.00$)
T-R2	12	0.61 ($\sigma = 0.00$)	0.97 ($\sigma = 0.00$)	0.89 ($\sigma = 0.00$)	0.34 ($\sigma = 0.00$)
T-R3	22	0.94 ($\sigma = 0.04$)	0.92 ($\sigma = 0.11$)	0.96 ($\sigma = 0.08$)	0.42 ($\sigma = 0.13$)
T-R4	28	1.11 ($\sigma = 0.03$)	0.84 ($\sigma = 0.28$)	1.03 ($\sigma = 0.32$)	0.83 ($\sigma = 0.70$)
T-R5	20	0.68 ($\sigma = 0.14$)	0.72 ($\sigma = 0.07$)	0.95 ($\sigma = 0.19$)	0.98 ($\sigma = 0.89$)
T-R6	20	0.87 ($\sigma = 0.19$)	0.75 ($\sigma = 0.23$)	0.96 ($\sigma = 0.09$)	0.42 ($\sigma = 0.00$)
T-R7	42	2.61 ($\sigma = 2.67$)	0.96 ($\sigma = 0.45$)	1.27 ($\sigma = 0.46$)	2.31 ($\sigma = 2.61$)

Table 4
Average runtimes of the proposed heuristics (in milliseconds).

Topology	Ops	Hill-Climbing	Ant System	Hybrid	Resource Aware cheduler
T-M1	6	9 ($\sigma = 2.25$)	75 ($\sigma = 7.88$)	78 ($\sigma = 4.67$)	16 ($\sigma = 2.08$)
T-M2	12	45 ($\sigma = 5.94$)	99 ($\sigma = 18.35$)	130 ($\sigma = 19.84$)	21 ($\sigma = 3.39$)
T-M3	3	4 ($\sigma = 0.55$)	35 ($\sigma = 1.82$)	38 ($\sigma = 2.83$)	16 ($\sigma = 2.70$)
T-R1	30	558 ($\sigma = 125.68$)	450 ($\sigma = 38.86$)	841 ($\sigma = 31.95$)	25 ($\sigma = 4.34$)
T-R2	12	65 ($\sigma = 14.52$)	131 ($\sigma = 21.14$)	197 ($\sigma = 26.40$)	20 ($\sigma = 3.16$)
T-R3	22	231 ($\sigma = 49.60$)	285 ($\sigma = 36.29$)	534 ($\sigma = 60.07$)	23 ($\sigma = 3.32$)
T-R4	28	179 ($\sigma = 59.91$)	447 ($\sigma = 80.54$)	717 ($\sigma = 80.49$)	24 ($\sigma = 3.59$)
T-R5	20	138 ($\sigma = 46.62$)	306 ($\sigma = 56.16$)	428 ($\sigma = 32.54$)	22 ($\sigma = 3.03$)
T-R6	20	198 ($\sigma = 75.98$)	274 ($\sigma = 67.17$)	472 ($\sigma = 40.59$)	21 ($\sigma = 3.47$)
T-R7	42	1029 ($\sigma = 28.00$)	661 ($\sigma = 84.76$)	1020 ($\sigma = 26.41$)	26 ($\sigma = 9.38$)
T-R8	14	185 ($\sigma = 61.83$)	377 ($\sigma = 41.18$)	355 ($\sigma = 63.65$)	19 ($\sigma = 2.60$)
T-R9	18	263 ($\sigma = 77.06$)	408 ($\sigma = 125.23$)	480 ($\sigma = 140.60$)	21 ($\sigma = 3.00$)

forms the Resource Aware Scheduler followed by hill-climbing ($\mu = 0.92$) and the ant system ($\mu = 0.87$).

To summarise the considerations on the throughput, hybrid search tends to outperform Apache Storm's build-in Resource Aware Scheduler on average across the random topologies by 1-3% on the median maximum throughput and maximum sustainable throughput. Hill-climbing performs better on the median max throughput by 11% and worse on the maximum sustainable throughput by 8%, and the ant system has 11-13% less. More generally, the proposed solutions achieve slightly higher throughput metrics on larger topologies and less on smaller ones, and T-R7 is a topology on which the Resource Aware Scheduler did not perform well.

6.2.2. Runtime

Table 4 compares the average runtime in milliseconds to compute a single placement. Unfortunately, the default scheduler does not report such a statistic. Still, its runtime is likely lower than that of the Resource Aware Scheduler since the functionality of the default scheduler is naturally less complex. The Resource Aware Scheduler provides the lowest runtimes, except on the smallest topologies. This result is expected because the proposed heuristics perform iterative adjustments and consider latency and data exchange in the placement problem. On most topologies, the hybrid solution requires the time of hill-climbing in addition to the ant system, which is also expected because it consists of running both solutions after each other. The ant system has a higher constant cost because it has a high minimum iteration count to counteract the probabilistic solution construction.

In contrast, it scales better than the other proposed solutions because a constant number of ants or solutions is computed rather than the quadratically growing neighborhood in hill-climbing. The pheromone can also focus the search on a smaller search space over time. Hill-climbing and the hybrid search were limited by the configured runtime limit of one second only on one random topology. The imposed design requirement on the runtime limit is very strict. However, it did not have a particularly noticeable effect on the quality of the placements. In comparison, the Resource Aware Scheduler uses a configurable timeout of 60 seconds by default. Ultimately, hill-climbing works well for smaller to medium topologies, while the ant system performs well on larger topologies.

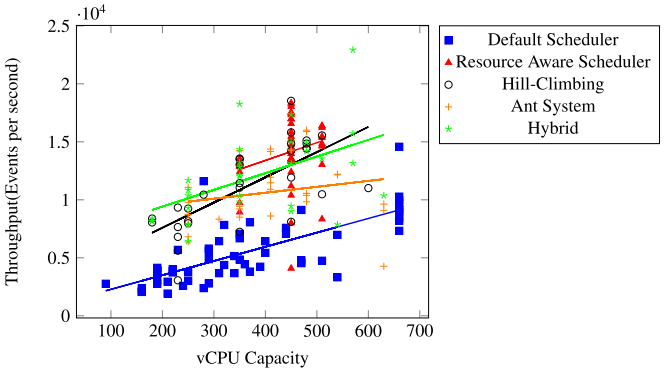


Fig. 6. Scatter plot of the median maximum throughput of individual placements for all placement heuristics with trend lines for T-R3. A vCPU capacity of 100 corresponds to the consumption of an entire core.

6.2.3. Individual results

In the following, we discuss some individual use cases to provide a better understanding of the average case and some special cases. Fig. 6 shows the median maximum throughput for T-R3, representing the average case. As can be seen in the figure, the default scheduler clearly shows the worst results and forms a cluster of its own. The other schedulers form a cluster near the center. Additionally, clusters on the same vertical line from individual schedulers are recognizable. These are es-

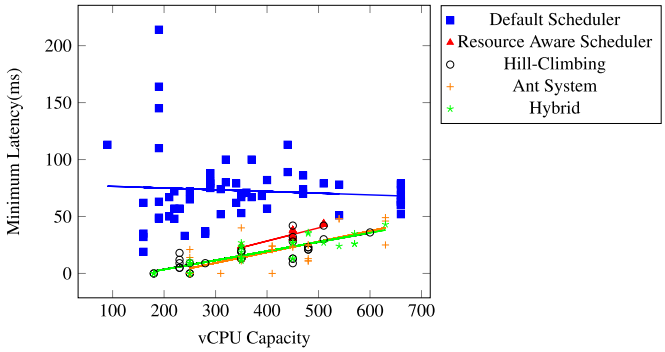


Fig. 7. Scatter plot of the minimum latency of individual placements for the placement heuristics with trend lines for T-R3. A vCPU capacity of 100 corresponds to the consumption of an entire core.

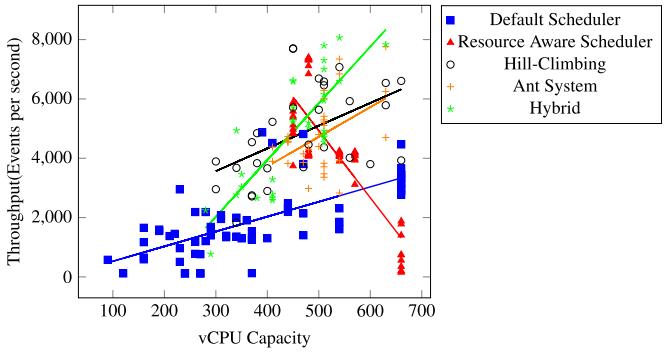


Fig. 8. Scatter plot of the median maximum throughput of individual placements for all placement heuristics with trend lines for T-R7. A vCPU capacity of 100 corresponds to the consumption of an entire core.

sentially similar or identical placements and show that the measured performance has a large variance even with identical placements. They are most notable for the Resource Aware Scheduler, which only produces a few placement variations, and the default scheduler, which has a line in the bottom right that utilizes all eleven computational resources.

It can be seen that the round-robin placement of the default scheduler results in relatively consistent but worse performance. A large part of this is that at practically any resource utilization, co-location of operators is unlikely. Most placements are, therefore, highly inefficient but more consistent. The Resource Aware Scheduler shows the contrast of similarly dense groupings of placements, but ones that largely benefit from co-location and avoid resource bottlenecks because of its awareness of heterogeneity. While the Resource Aware Scheduler usually does not produce the best-performing placements, they are consistently among the best performing, resulting in its generally above-average performance. The most varied performance is showcased by the proposed heuristics.

The least resource-consuming placements outside the default scheduler are generally created by hill-climbing and, in turn, the hybrid approach. In contrast, the ant system that does not perform a greedy search is far more widespread in performance and resource utilization.

In Fig. 7, the minimum measured latency is shown for the same topology. Again, it is straightforward to tell that the default scheduler performs significantly worse than the other approaches. This is because most operators are not co-located and use more network links. The figure shows many similarities to the throughput.

After discussing the general case, we examine the results of some special cases. In T-M3, only three operators are in the topology. Therefore, when accounting for co-locality, their placements are identical. Still, the Resource Aware Scheduler achieves higher throughput. The reason for this is the additional metrics consumer deployed with the proposed solutions to collect the topologies' metrics and indirectly forward them to the placement heuristic. While the workload should be minor, it is still an extra operator compared to the static heuristics, where Storm has not been configured to include the extra operator in the topologies. The actual performance difference shows an 11% throughput loss. In the larger topologies, this difference is less noticeable and is mainly lost in the high variance the metric usually has.

T-R7 is interesting because as the heuristics scale out, the Resource Aware Scheduler's throughput tends to drop, as can be seen in Fig. 8. Unfortunately, no exceptional cause could be identified with certainty as the source of this behavior. Similar throughput reductions, but far less significant, occur for the Resource Aware Scheduler on T-R4, T-R5, and potentially T-R1. These throughput reductions occur when the smaller resources are starting to be utilized, which indicates a problem with communication overhead, but at the same time, this is not observed on T-R6. Only the ant system has consistent reductions in the throughput as it scales out, which are less severe and affect fewer topologies. Further, these are the largest topologies and highlight the limitations of the greedy heuristic.

Finally, concerning the cyclic topologies T-R8 and T-R9, it is difficult to directly compare them because the placement heuristics operated at different scales. As such, only the functionality itself could be asserted for all placement heuristics, with the caveat that Apache Storm can deadlock any processing if a cyclic topology is overloaded, turning this into a relatively insignificant feature.

The evaluation shows the benefits and feasibility of an adaptive placement heuristic based on latency estimations. The default scheduler was significantly outperformed in any metric except for the increased computational cost of scheduling. Further, the hybrid approach and hill-climbing perform close to or better in every metric except the runtime than the Resource Aware Scheduler without the risk of misconfiguration of the scheduler. While the Resource Aware Scheduler generally achieves better throughput, it cannot do so consistently. If these cases are considered, hill-climbing achieves 11% higher maximum throughput, while the hybrid approach has 1-3% better maximum sustainable and median maximum throughput.

Similarly, average latency improved by 2% for hill-climbing and worsened by 4% for the hybrid search. The largest difference is the minimum latency, where all proposed solutions achieve reductions of at least 22% on average, at the cost of increasing the maximum latency. Hill-climbing has a 5% larger maximum latency, with hybrid search at 7%. The performance generalizes to all tested topologies reasonably well, and no problematic cases can be identified. Both the hill-climbing and hybrid heuristics perform particularly well, but hill-climbing seems preferable because of a far simpler implementation with fewer parameters to tune and offers better runtime performance. Additionally, while the ant system generally performs worse in most experiments, it does offer better scalability.

7. Related work

While several approaches to solve the stream operator placement problem have been proposed, to the best of our knowledge, no related approach delivers all of the features presented in the work at hand.

An important prerequisite for the presented placement approach is to be able to estimate latency values between nodes. This is used to favor co-locations, as discussed in the problem formulation in Section 4.2. To estimate these values, position-based methods and techniques for filling out suitable latency matrices have been presented [59]. Typically, approaches that rely on positions in a defined space are also referred to as embeddings, which aim to estimate the current coordinates of any computational resource in the network [60]. Spring-based forces can push resources apart or pull resources together in the defined space based on estimated latency measurements to estimate positions efficiently. The computations leading to these estimations can be repeated iteratively to update the latency based on current measurements and integrate these

Table 5
Feature Comparison of Related Approaches.

	Pietzuch et al. [7]	Rizou et al. [55]	Cardellini et al. [8,9]	Ottenwälder et al. [57]	Prosperi et al. [23]	Veith et al. [56]	Hiessl et al. [3]	Lambert et al. [58]	Hybrid
Online Scheduler	☑	☑	☑	☑	☐	☐	☑	☐	☑
Heterogeneous Resources	☑	☐	☑	☐	☐	☑	☑	☐	☑
Resource Optimization	☒	☐	☒	☐	☐	☒	☒	☐	☑
Awareness: Latency	☑	☑	☑	☑	☑	☑	☑	☐	☑
Optimization: Latency	☑	☑	☑	☑	☑	☑	☑	☐	☑
Awareness: Bandwidth	☑	☑	☑	☑	☑	☑	☐	☑	☑
Optimization: Bandwidth	☑	☑	☑	☑	☑	☐	☐	☑	☑

new values to a matrix [61]. An alternative method, matrix completion, aims to estimate only missing values of a matrix M that includes the latency measurements [59]. While these works provide foundational work on estimating latency, our work adapts such latency estimations to stream processing for the compute continuum and provides critical implementation details and respective evaluation results.

The *Stream-based Overlay Network* (SBON) [7] is a seminal approach applying spring-based forces to estimate latency values in the three-dimensional Euclidean space. Interestingly, this work also presents an approach to optimize the placement of operators on computational resources by considering network usage in addition to latency. However, this paper presents early work without details regarding integration into cloud and edge computing environments. Also, since this solution focuses on the network, heterogeneous resource demands are only indirectly optimized, i.e., if the resource constraints are fulfilled, the nearest node will be selected. A similar approach is presented by Rizou et al. [55], where the goal is to minimize network utilization instead of latency. In contrast, the work at hand takes into account the distributed nodes of the compute continuum as well as other heterogeneous computational resources that can be represented by the supervisors, as discussed in Section 3.3, and provides an evaluation based on experiments conducted over a network of various edge and cloud nodes. This evaluation showcases benefits applicable to the compute continuum that have not been explored to this extent in previous works.

The SBON approach has also been extended, e.g., by Cardellini et al. [8,9], which provide the most related approaches from the literature to the work at hand. Cardellini et al. extend SBON by implementing a distributed scheduler into Apache Storm to consider fog resources. In addition, a virtual placement is proposed, arranging resources in the latency space, which then aids in finding the closest physical node based on utilizing the available computational resources. Nevertheless, during these estimations, only the latency and data rate are considered for the spring-based force minimization, which means that the focus is on optimizing network metrics. Interestingly, the work investigates operator placement, aiming to maximize resource utilization by consolidating placements on computational resources. This also increases the occurrence of operator co-locations, leading to improved communication between operators 6.

MigCEP is a method for the placement and migration of operators intended to be used by infrastructure providers [57]. MigCEP places operators to satisfy latency constraints and creates a migration plan for each operator in advance to reduce network utilization. This migration plan consists of transfers of an operator to new computational resources that meet latency requirements while minimizing the total cost of migrations. Every transfer has a defined start time and deadline that rely on predicted mobility patterns showing the movement of data sources and sinks. While this work offers valuable insights for placing operators, our work offers a more comprehensive approach to optimized scheduling and integration stream processing framework, extending the state of the art with implementation details and results from a widely used stream processing framework.

Prosperi et al. [23] present *Planner*. Planner can identify subgraphs in a processing topology to be deployed in the cloud or at the edge. This approach makes the assumption that data is created at the edge, and that data from the edge should be sent to the cloud. This assumption defines where data sources and sinks need to be placed, i.e., at the edge and the cloud, respectively. Planner aims at transferring data from the edge to the cloud while maximizing the utilization of edge resources, rather than reducing resource utilization of all the available resources. A related approach is also presented by Veith et al. [56], who identify regions in the stream processor topology first and then apply strategies to place operators in the cloud or at the edge accordingly. In contrast, we formulate an optimization problem considering all the available resources, and we do not restrain the location of operators, allowing more flexibility that can be especially useful when deploying topologies on the various nodes of the compute continuum.

Hiessl et al. [3] propose an approach to minimize the latency between operators by considering the highest response time of all paths in a topology based on network latency and processing delay. Interestingly, this approach limits the number of operators to one per computational resource, which aids in finding optimized solutions. However, this constraint prevents optimized solutions that include co-locations of operators. In our work, we avoid this limitation by formulating an optimization problem that includes co-locations to target the various available nodes of the compute continuum.

Lambert et al. [58] assume that the data transfer rate is a bottleneck in stream processing topologies and present an optimization approach to maximize throughput sustainably. This approach effectively spreads the data and computations across the network to achieve higher throughput. Nevertheless, reducing latency is not a major goal of this work, an aspect we consider in the work at hand.

To further show how our work complements the state of the art, Table 5 provides an overview of the most important related works, following the same order as the related studies discussed above. In this table, ☑ indicates that a feature is provided; ☐ indicates that a feature is not provided or not discussed; ☒ is used to indicate that a feature is provided partially. The selected features include the following. Online Scheduler indicates that the scheduling optimization occurs repeatedly. Heterogeneous Resources indicates that various types of resources are considered, e.g., how our work also considers heterogeneous resource-constrained devices represented by the supervisor. Resource Optimization indicates that the optimization is done for heterogeneous resources. The sign ☒ for Resource Optimization means that the resources are considered constraints but not directly optimized. Finally, we differentiate between optimization targets and awareness of latency and bandwidth. Awareness indicates that a metric is seen as a constraint but not necessarily directly optimized, while Optimization indicates that this metric is optimized.

Interestingly, Table 5 shows that many approaches take into account latency, although they typically focus on network-related metrics without also considering resource optimization. Only a limited number of approaches consider resource optimization. Notably, only the work at hand directly optimizes resource consumption instead of merely considering resources as constraints. In addition, none of the discussed approaches explores the proposed approaches of this work, i.e., ant-based and hill-climbing heuristics, to solve the problem.

Finally, since this is an extended version of our previous work [24], we highlight some key differences in the following. Compared to [24], the work at hand presents a prototype based on Apache Storm accompanied with implementation details and extensively evaluates the proposed solution. This evaluation also includes a detailed explanation of the experimental setup and provides a throughput and runtime analysis of different test cases. Additionally, the evaluation discusses various specific results of notable test cases while [24] discusses only a single test case. Furthermore, we describe details regarding background information necessary to understand the applied concepts, and we provide a more extensive discussion of the related work distinguishing the work at hand from existing literature.

8. Conclusion

In this paper, we formulate an optimization problem for the placement of stream operators that is designed for processing IoT data in the compute continuum. This optimization problem takes into account the potentially distributed computational resources of the compute continuum and favors the co-location of operators, which can lead to lower processing latency. Furthermore, we design heuristics to solve this optimization problem efficiently. To evaluate these heuristics, we conduct several experiments using an Apache Storm-based prototype comparing the proposed optimized scheduling with alternatives, and we show various benefits considering, e.g., throughput, latency, and resource utilization. In future work, we plan to decentralize the placement mechanism and extend our work by considering additional metrics and constraints. In addition, as fluctuating workloads and network capacities might occur in the IoT, we plan to further model and evaluate such aspects, leading to a better understanding of the behavior of optimized scheduling during highly dynamic conditions.

CRediT authorship contribution statement

Raphael Ecker: Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Data curation, Conceptualization. **Vasileios Karagiannis:** Writing – review & editing, Writing – original draft, Supervision. **Michael Sober:** Writing – review & editing, Writing – original draft. **Stefan Schulte:** Writing – review & editing, Writing – original draft, Supervision.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Michael Sober reports financial support was provided by Christian Doppler Research Association. Stefan Schulte reports financial support was provided by Christian Doppler Research Association. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development as well as the Christian Doppler Research Association is gratefully acknowledged.

Data availability

Data will be made available on request.

References

[1] C. Axenie, R. Tudoran, S. Bortoli, M.A.H. Hassan, C.S. Sánchez, G. Brasche, Dimensionality reduction for low-latency high-throughput fraud detection on datastreams, in: 18th IEEE International Conference on Machine Learning and Applications, IEEE, 2019, pp. 1170–1177.
[2] M.D. de Assunção, A.D.S. Veith, R. Buyya, Distributed data stream processing and edge computing: a survey on resource elasticity and future directions, J. Netw. Comput. Appl. 103 (2018) 1–17.
[3] T. Hiessl, V. Karagiannis, C. Hochreiner, S. Schulte, M. Nardelli, Optimal placement of stream processing operators in the fog, in: 3rd IEEE International Conference on Fog and Edge Computing, IEEE, 2019, pp. 1–10.
[4] IEEE, IEEE Standard 1934-2018 for Adoption of OpenFog Reference Architecture for Fog Computing, Aug. 2018.
[5] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, J.P. Jue, All one needs to know about fog computing and related edge computing paradigms: a complete survey, J. Syst. Archit. 98 (2019) 289–330.
[6] P. Varshney, Y. Simmhan, Characterizing application scheduling on edge, fog and cloud computing resources, Softw. Pract. Exp. 50 (5) (2020) 558–595.
[7] P.R. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, M.I. Seltzer, Network-aware operator placement for stream-processing systems, in: 22nd International Conference on Data Engineering, IEEE, 2006, p. 49.
[8] V. Cardellini, V. Grassi, F.L. Presti, M. Nardelli, On qos-aware scheduling of data stream applications over fog computing infrastructures, in: 2015 IEEE Symposium on Computers and Communication, IEEE, 2015, pp. 271–276.
[9] V. Cardellini, V. Grassi, F.L. Presti, M. Nardelli, Distributed qos-aware scheduling in storm, in: 9th ACM International Conference on Distributed Event-Based Systems, ACM, 2015, pp. 344–347.
[10] M. Nardelli, V. Cardellini, V. Grassi, F.L. Presti, Efficient operator placement for distributed data stream processing applications, IEEE Trans. Parallel Distrib. Syst. 30 (8) (2019) 1753–1767.
[11] J.B. Dennis, D. Misunas, A preliminary architecture for a basic data flow processor, in: 2nd Annual Symposium on Computer Architecture, ACM, 1974, pp. 126–132.
[12] C. Hochreiner, M. Vögler, S. Schulte, S. Dustdar, Elastic stream processing for the Internet of Things, in: 9th IEEE International Conference on Cloud Computing (CLOUD 2016), IEEE, 2016, pp. 100–107.
[13] O. Marcu, A. Costan, G. Antoniu, M.S. Pérez-Hernández, R. Tudoran, S. Bortoli, B. Nicolae, Towards a unified storage and ingestion architecture for stream processing, in: 2017 IEEE International Conference on Big Data (BigData 2017), IEEE, 2017, pp. 2402–2407.
[14] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas, Apache flink™: stream and batch processing in a single engine, IEEE Data Eng. Bull. 38 (4) (2015) 28–38.
[15] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J.M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, D.V. Ryaboy, Storm@twitter, in: International Conference on Management of Data (SIGMOD 2014), ACM, 2014, pp. 147–156.
[16] V. Cardellini, F. Lo Presti, M. Nardelli, G.R. Russo, Runtime adaptation of data stream processing systems: the state of the art, ACM Comput. Surv. 54 (11s) (2022) 237:1–237:36.
[17] Y. Simmhan, Big data and fog computing, in: Encyclopedia of Big Data Technologies, Springer, 2019.
[18] M. Jansen, A. Al-Dulaimy, A.V. Papadopoulos, A. Trivedi, A. Iosup, The SPEC-RG reference architecture for the compute continuum, in: 23rd IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid 2023), IEEE, 2023, pp. 469–484.
[19] M. Satyanarayanan, The emergence of edge computing, Computer 50 (1) (2017) 30–39.
[20] D.T. Hoang, C. Lee, D. Niyato, P. Wang, A survey of mobile cloud computing: architecture, applications, and approaches, Wirel. Commun. Mob. Comput. 13 (18) (2013) 1587–1611.
[21] G.T. Lakshmanan, Y. Li, R.E. Strom, Placement strategies for internet-scale data stream systems, IEEE Internet Comput. 12 (6) (2008) 50–60.
[22] C. Thoma, A. Labrinidis, A.J. Lee, Automated operator placement in distributed data stream management systems subject to user constraints, in: Proceedings of the 30th International Conference on Data Engineering Workshops, ICDE 2014, Chicago, IL, USA, March 31 - April 4, 2014, IEEE Computer Society, 2014, pp. 310–316.
[23] L. Prosperi, A. Costan, P. Silva, G. Antoniu, Planner: cost-efficient execution plans placement for uniform stream analytics on edge and cloud, in: 2nd IEEE/ACM Workflows in Support of Large-Scale Science, IEEE, 2018, pp. 42–51.
[24] R. Ecker, V. Karagiannis, M. Sober, E. Ebrahimi, S. Schulte, Latency-aware placement of stream processing operators, in: International Workshop on Scalable Compute Continuum (WSCC 2023), Springer, 2023, pp. 1–12.
[25] A. Harwood, M.R. Read, G.N. Amarasinghe, Dragon: a lightweight, high performance distributed stream processing engine, in: 40th IEEE International Conference on Distributed Computing Systems, (ICDCS 2020), IEEE, 2020, pp. 1344–1351.

[26] L. Aniello, R. Baldoni, L. Querzoni, Adaptive online scheduling in storm, in: The 7th ACM International Conference on Distributed Event-Based Systems, DEBS '13, Arlington, TX, USA, June 29 - July 03, 2013, ACM, 2013, pp. 207–218.
[27] J. Xu, Z. Chen, J. Tang, S. Su, T-Storm: Traffic-Aware Online Scheduling in Storm, IEEE 34th International Conference on Distributed Computing Systems, ICDCS 2014, Madrid, Spain, June 30 - July 3, 2014, IEEE Computer Society, 2014, pp. 535–544.
[28] B. Peng, M. Hosseini, Z. Hong, R. Farivar, R.H. Campbell, R-storm: resource-aware scheduling in storm, in: 16th Annual Middleware Conference, ACM, 2015, pp. 149–161.
[29] A. Muhammad, M. Aleem, M.A. Islam, Top-storm: a topology-based resource-aware scheduler for stream processing engine, Clust. Comput. 24 (1) (2021) 417–431.
[30] T. Qi, M. Rodriguez, A traffic and resource aware online storm scheduler, in: ACSW '21: 2021 Australasian Computer Science Week Multiconference, Dunedin, New Zealand, 1-5 February, 2021, ACM, 2021, pp. 8:1–8:10.
[31] V. Cardellini, M. Nardelli, D. Luzi, Elastic stateful stream processing in storm, in: 14th International Conference on High Performance Computing & Simulation, HPCS 2016, Innsbruck, Austria, July 18-22, 2016, IEEE, 2016, pp. 583–590.
[32] X. Liu, R. Buyya, Resource management and scheduling in distributed stream processing systems: a taxonomy, review, and future directions, ACM Comput. Surv. 53 (3) (2020) 1–41.
[33] T. De Matteis, G. Mencagli, Proactive elasticity and energy awareness in data stream processing, J. Syst. Softw. 127 (2017) 302–319.
[34] V. Karagiannis, S. Schulte, edgerouting: using compute nodes in proximity to route iot data, IEEE Access 9 (2021) 105841–105858.
[35] T. Onishi, J. Michaelis, Y. Kanemasa, Recovery-conscious adaptive watermark generation for time-order event stream processing, in: 2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI), IEEE, 2020, pp. 66–78.
[36] O. Qayyum, W. Yu, Toward replicated and asynchronous data streams for edge-cloud applications, in: Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing, 2022, pp. 339–346.
[37] P. Eades, A heuristic for graph drawing, Congr. Numer. 42 (1984) 149–160.
[38] S.G. Kobourov, Spring embedders and force directed graph drawing algorithms, CoRR, arXiv:1201.3011, 2012.
[39] M. Szymaniak, D.L. Presotto, G. Pierre, M. van Steen, Practical large-scale latency estimation, Comput. Netw. 52 (7) (2008) 1343–1364.
[40] Apache Software Foundation, Apache storm documentation: Faq, https://storm.apache.org/releases/2.4.0/FAQ.html, 2022. (Accessed 31 March 2022).
[41] C. Tsai, J.J.P.C. Rodrigues, Metaheuristic scheduling for cloud: a survey, IEEE Syst. J. 8 (1) (2014) 279–291.
[42] M. Dorigo, V. Maniezzo, A. Colorni, Ant system: optimization by a colony of cooperating agents, IEEE Trans. Syst. Man Cybern. B 26 (1) (1996) 29–41.
[43] T. Stützle, H.H. Hoos, MAX-MIN ant system, Future Gener. Comput. Syst. 16 (8) (2000) 889–914.
[44] Redis Labs, Redis, https://redis.io/, 2021. (Accessed 31 March 2022).
[45] Apache Software Foundation, Apache storm documentation: Storm ui rest api https://storm.apache.org/releases/2.4.0/STORM-UI-REST-API.html, 2022. (Accessed 31 March 2022).
[46] Apache Software Foundation, Apache storm documentation: Resource aware scheduler https://storm.apache.org/releases/2.4.0/Resource_Aware_Scheduler_overview.html#Enhancements-on-original-DefaultResourceAwareStrategy, 2022. (Accessed 31 March 2022).
[47] P. Eades, X. Lin, W.F. Smyth, A fast and effective heuristic for the feedback arc set problem, Inf. Process. Lett. 47 (6) (1993) 319–323.
[48] A.B. Kahn, Topological sorting of large networks, Commun. ACM 5 (11) (1962) 558–562.
[49] R.E. Tarjan, Depth-first search and linear graph algorithms, SIAM J. Comput. 1 (2) (1972) 146–160.
[50] X. Wei, Y. Zhuang, H. Li, Z. Liu, Reliable stream data processing for elastic distributed stream processing systems, Clust. Comput. 23 (2020) 555–574.
[51] A. Jlassi, C. Tedeschi, Merge, split, and cluster: dynamic deployment of stream processing applications, in: 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID), IEEE, 2020, pp. 71–80.
[52] Apache Software Foundation, Apache storm documentation: Scheduler https://storm.apache.org/releases/2.4.0/Storm-Scheduler.html, 2022. (Accessed 31 March 2022).
[53] B. Charyyev, E. Arslan, M.H. Gunes, Latency comparison of cloud datacenters and edge servers, in: IEEE Global Communications Conference, GLOBECOM 2020, Virtual Event, Taiwan, December 7–11, 2020, IEEE, 2020, pp. 1–6.
[54] Apache Software Foundation, Quagga routing suite, https://www.quagga.net/, 2018. (Accessed 31 March 2022).
[55] S. Rizou, F. Dürr, K. Rothermel, Solving the multi-operator placement problem in large-scale operator networks, in: Proceedings of the 19th International Conference on Computer Communications and Networks, IEEE ICCCN 2010, Zürich, Switzerland, August 2-5, 2010, IEEE, 2010, pp. 1–6.
[56] A.D.S. Veith, M.D. de Assunção, L. Lefèvre, Latency-aware placement of data stream analytics on edge computing, in: 16th International Conference on Service-Oriented Computing, ICSOC 2018, Hangzhou, China, November 12-15, 2018, in: Lecture Notes in Computer Science, vol. 11236, Springer, 2018, pp. 215–229.
[57] B. Ottenwälder, B. Koldehofe, K. Rothermel, U. Ramachandran, Migcep: operator migration for mobility driven distributed complex event processing, in: The 7th ACM International Conference on Distributed Event-Based Systems, DEBS '13, Arlington, TX, USA - June 29 - July 03, 2013, ACM, 2013, pp. 183–194.
[58] T. Lambert, D. Guyon, S. Ibrahim, Rethinking operators placement of stream data application in the edge, in: The 29th ACM International Conference on Information and Knowledge Management, CIKM 2020, Virtual Event, Ireland, October 19-23, 2020, ACM, 2020, pp. 2101–2104.
[59] R. Zhu, B. Liu, D. Niu, Z. Li, H.V. Zhao, Network latency estimation for personal devices: a matrix completion approach, IEEE/ACM Trans. Netw. 25 (2) (2017) 724–737.
[60] R. Tripathi, K. Rajawat, Dynamic network latency prediction with adaptive matrix completion, in: 12th International Conference on Signal Processing and Communications (SPCOM), Bangalore, India, July 16–19, 2018, IEEE, 2018, pp. 407–411.
[61] Y. Fu, Y. Wang, Hyperspring: accurate and stable latency estimation in the hyperbolic space, in: 15th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2009, Shenzhen, China, December 8-11, 2009, IEEE Computer Society, 2009, pp. 864–869.

Raphael Ecker graduated with a master's degree in Software Engineering & Internet Computing at the Vienna University of Technology in 2023. He is currently working on Industry 4.0 as a developer of distributed systems in the field of high-performance automation.

Vasileios Karagiannis is a Scientist at the Austrian Institute of Technology where he is part of the Competence Unit for Cooperative Digital Technologies. He holds a PhD in Computer Science from the Vienna University of Technology and an MSc in Hardware and Software Integrated Systems from the University of Patras. With a keen interest in the digital transformation of services and societies, Vasileios focuses his research on Edge Computing, Data Sovereignty, and the Internet of Things. His work has resulted in the publication of numerous scientific articles and patents, highlighting his contributions to the advancement of these fields.

Michael Sober received his master's degree in Software Engineering & Internet Computing from TU Wien in 2020. He is a research assistant and Ph.D. student at the Institute for Data Engineering at TU Hamburg and working on blockchain interoperability solutions in the Christian Doppler Laboratory Blockchain Technologies for the Internet of Things (CDL-BOT).

Professor Stefan Schulte heads the Institute for Data Engineering at Hamburg University of Technology, Germany, and leads the Christian Doppler Laboratory Blockchain Technologies for the Internet of Things (CDL-BOT). His research interests include the areas of data stream processing, the Internet of Things, and distributed ledger technologies. Findings from his research have been published in more than 100 refereed scholarly publications.