Meerkat: Scalable, Network-Aware Failure Recovery for the Internet of Things

Anastasiia Kozar
BIFOLD, TU Berlin
anastasiia.kozar@tu-berlin.de

Ankit Chaudhary
BIFOLD, TU Berlin
ankit.chaudhary@tu-berlin.de

Steffen Zeuch
BIFOLD, TU Berlin
steffen.zeuch@tu-berlin.de

Volker Markl
BIFOLD, TU Berlin, DFKI
volker.markl@tu-berlin.de

ABSTRACT

The Internet of Things (IoT) demands real-time, low-latency processing of data generated by thousands of heterogeneous, resource-constrained devices. In such dynamic environments, ensuring fault tolerance becomes critical, especially for safety-sensitive applications like disaster management or patient monitoring. However, existing centralized fault tolerance solutions face serious scalability challenges across large, hierarchically connected IoT topologies.

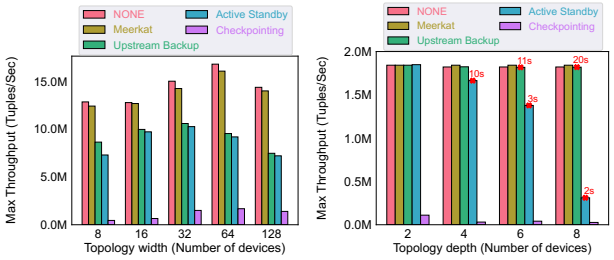
In this paper, we present Meerkat, a network-aware fault-tolerance protocol explicitly designed for IoT environments. Meerkat achieves zero-downtime recovery via redundant operator execution on disjoint paths and efficient duplicate detection. It also includes dynamic load balancing that adapts operator placement to device volatility, ensuring fair resource use. Compared to state-of-the-art techniques, Meerkat sustains up to 70× higher throughput with only 28% network overhead. These results highlight Meerkat's ability to deliver efficient fault tolerance with minimal overhead at IoT scale.

PVLDB Reference Format:
Anastasiia Kozar, Ankit Chaudhary,
Steffen Zeuch, and Volker Markl. Meerkat: Scalable, Network-Aware Failure Recovery for the Internet of Things. PVLDB, 19(3): 306-319, 2025.
doi:10.14778/3778092.3778094

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://anonymous.4open.science/r/meerkat/README.md.

1 INTRODUCTION

Large-scale sensor network deployments, such as the Internet of Things (IoT), enable low-latency, real-time data processing across heterogeneous devices operating often in constrained and volatile environments [59, 65, 78]. These systems follow a hierarchical architecture in which resource-limited devices (e.g., sensors or gateways) process data locally and forward results to cloud data centers [14, 78]. As deployments might scale to thousands or even millions of nodes, maintaining fault tolerance becomes essential for availability and correctness in such volatile environments [7, 25, 26]. This is particularly critical in domains such as intensive care monitoring [40, 60], disaster response [30], and autonomous vehicles [66], where failures can have serious consequences.



(a) Wide topology (b) Deep topology

Figure 1: Scalability in wide and deep topologies.

Ensuring reliable stream processing in edge-cloud environments poses unique challenges [14, 47, 59]. Unlike cloud-based systems that rely on replayable logs (e.g., Kafka [64]), high-speed interconnects, and persistent storage (e.g., HDFS [67]), edge deployments operate under stricter constraints. Devices are often mobile, battery-powered, sparsely connected, and lack shared infrastructure [29, 58, 78]. These constraints break the assumptions needed to apply conventional fault tolerance techniques in edge infrastructures.

To illustrate these limitations, we revisit three common fault tolerance techniques: Upstream Backup (UB) [42], Active Standby (AS) [38], and Checkpointing (CH) [19]. Figure 1 presents throughput on a Raspberry Pi cluster under increasing *width* (depth 1) and *depth* (at width 1 or 2 for protocols requiring parallel paths) of the infrastructure, comparing UB ▬, AS ▬, and CH ▬ to a no-fault-tolerance baseline (NONE) ▬. If the system stalls, we mark it with ✗ and report the failure time. All three rely on re-playable sources and shared storage, which edge systems often lack.

UB requires each device to buffer output until the sink acknowledges it, since disconnected devices cannot reliably retransmit. While this prevents data loss, it creates scalability bottlenecks: deeper topologies delay acknowledgments and increase buffer pressure, and wider topologies force the coordinator to track more paths [47]. As Figure 1 shows, UB throughput drops by 48% at width 128 and stalls by depth 6. AS shares the same design, failing by depth 4 with a similar 50% drop at width 128. CH encounters comparable issues.

Systems like Flink rely on shared storage and upstream logs for recovery, but these assumptions break in edge-cloud environments. Devices must retain in-flight data until sink confirmation, and local

Proceedings of the VLDB Endowment, Vol. 19, No. 3 ISSN 2150-8097.
doi:10.14778/3778092.3778094

storage is unsafe without replication. Consequently, CH combines UB-style buffering with centralized snapshots, further increasing memory and communication overhead. As shown in Figure 1, CH throughput falls from 110k at depth 2 to 26k at depth 8, and reaches only 1.4M at width 128.

In general, while centralized designs simplify recovery, they introduce bottlenecks, reduce fault isolation, and limit autonomy. As shown in Figure 1, a single coordinator becomes a scalability barrier in resource-constrained environments. In contrast, fully decentralized systems, such as peer-to-peer engines [12, 13] or WSN protocols [51, 75], offer autonomy but lack strong guarantees. Neither extreme is well-suited to modern edge-cloud deployments.

In this paper, we explore a hybrid design that combines local decision-making with global coordination. We introduce Meerkat, the first hybrid fault tolerance protocol for IoT stream processing that provides an exactly-once* guarantee (assuming non-failing sources). Meerkat overcomes scalability bottlenecks by decentralizing runtime decisions and avoiding trimming-based coordination, allowing the system to remain responsive even if the coordinator becomes unavailable. As shown in Figure 1, Meerkat ▬ sustains near–baseline throughput even at large scale and across diverse topologies. Meerkat achieves this through redundant execution along parallel paths with duplicate suppression and a lightweight recovery process that ensures continuous operation.

Meerkat addresses three key challenges in IoT stream processing. First, it supports hierarchical topologies that break assumptions of traditional all-to-all fault tolerance, preserving neighborhood-level routing and enabling local recovery via dynamic device discovery. Second, it adapts to mobility and volatility by redistributing load among neighbors based on runtime conditions. Third, it handles heterogeneous device capabilities using a unified load metric reflecting memory, CPU, and network usage for fair task placement. By offloading part of the fault tolerance logic to workers, Meerkat reduces coordinator pressure and improves responsiveness, combining centralized replication for initial placement with decentralized load redistribution during execution.

We evaluate Meerkat in NebulaStream (NES), an IoT SPE leveraging hardware-tailored code compilation and a highly dynamic execution model [78]. NES with Meerkat achieves up to 70x higher throughput than existing solutions with only 28% added network usage. It enables seamless task migration under failures and load spikes and preserves availability in constrained, volatile environments.

In summary, this paper makes the following contributions:

- We introduce Meerkat, a hybrid fault tolerance protocol for massively distributed IoT stream processing that addresses heterogeneity, volatility, and hierarchical topologies (Section 3).
- We present Meerkat's hybrid architecture, which combines centralized operator replication with decentralized load balancing based on local resource conditions (Sections 4 and 5).
- We demonstrate that Meerkat achieves zero-downtime recovery with minimal overhead across a wide set of workloads (Section 6).

2 BACKGROUND

In the following section, we introduce concepts of an IoT-based SPE (Sec. 2.1), existing load balancing strategies (Sec. 2.3), and recovery types (Sec. 2.2).

2.1 IoT-based Stream Processing Engines

A SPE handles finite sequences of data streams ($s \in S$), where each stream consists of infinite relational tuples ($t \in T$). Each tuple $t = (\sigma, \pi)$ includes a timestamp ($\sigma \in \mathbb{N}^+$) and a payload (π). Depending on the system and its sources, these timestamps may be assigned either by an external physical clock or by an internal logical clock. Tuples can arrive out of order relative to their timestamps (i.e., $\sigma_i < \sigma_j$ is not guaranteed for $i < j$), for instance when network latency or device clocks skew their arrival times.

Most SPEs, including those tailored for IoT scenarios, represent computations as a directed acyclic graph (DAG) $G = (V, E)$, where vertices V are operators (e.g., map, filter, join), and edges E denote data dependencies among these operators. A vertex with no incoming edges is a source, while a vertex with no outgoing edges is a sink. Data flows from sources to sinks in a tree-like (or sometimes multi-level) topology [14, 78].

IoT SPEs typically follow a coordinator-worker model [9, 19, 78], where the central coordinator handles query compilation, optimization, and operator placement [59, 78]. It parses queries into DAGs, decomposes them into subplans, and assigns them to workers. Workers either compile subplans locally (e.g., using hardware-specific methods [39]) or deploy them directly, establishing communication channels to forward results toward the sinks. This architecture enables centralized planning and fault tolerance, while distributing computation to the edge to minimize latency and bandwidth.

2.2 Recovery Types

Distributed systems, including IoT deployments, may experience disruptions caused by component crashes, message omissions, transient software bugs, security breaches, malicious behavior, timing violations, or environmental conditions [37]. In stream processing, three broad recovery paradigms often appear:

Precise Recovery [42] ensures that output under failure is identical to a system with no failures. While this is the strongest guarantee, it can be costly in terms of time and memory.

Gap Recovery permits partial information loss to accelerate recovery time. By accepting a data "gap", the system can return to normal operations more quickly but at the cost of incomplete results.

Rollback Recovery does not lose data but may allow duplicates in the output. Rather than requiring perfectly consistent states, the system "rolls back" operators to a recent checkpoint or stored state and then replays any missing input. This approach balances correctness and resource overhead, making it well-suited to distributed, resource-limited IoT scenarios.

In this paper, we concentrate on rollback recovery because it avoids data loss while requiring less global synchronization than precise recovery, an important advantage in large-scale, hierarchical edge-cloud environments.

2.3 Load Balancing

Task delegation, also known as load balancing, distributes work across nodes to avoid overload, improve performance, and ensure efficient use of resources. This is especially important in large or heterogeneous IoT deployments. We briefly outline several established strategies:

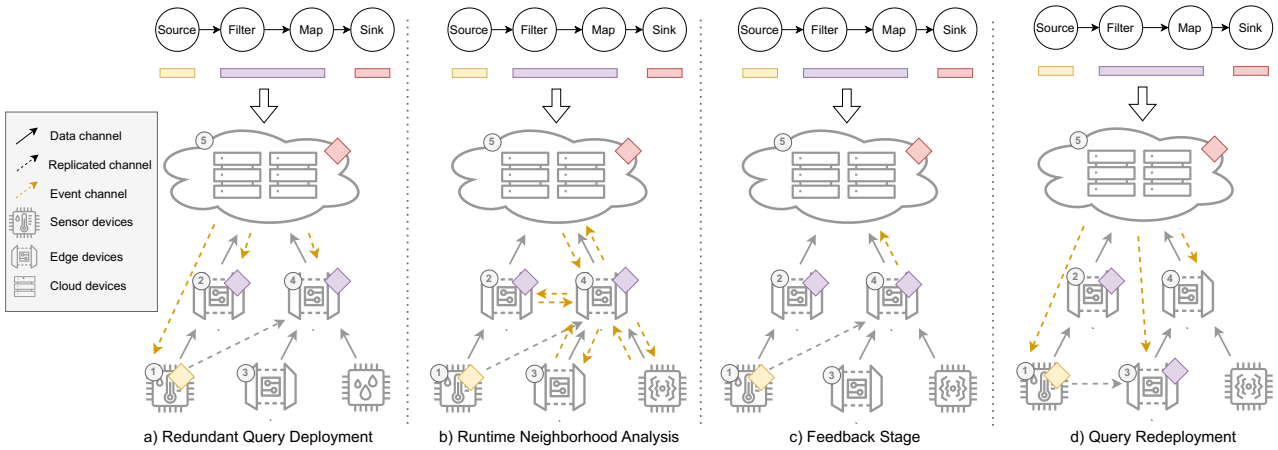


Figure 2: Query deployment with Meerkat.

Randomized. Tasks are assigned to randomly selected neighbors [6, 17, 57]. This method is simple and requires no global state, but it may lead to uneven distribution.

Round-Robin. Tasks are forwarded cyclically among peers [8, 46, 56, 73]. This guarantees fairness in count but ignores variations in resource availability.

Resource-Based. Offloading decisions depend on current resource levels, such as CPU, memory, or bandwidth [18, 44], allowing better matching of tasks to node capacity.

Demand-Based. Here, task movement is guided by workload: nodes with lower load may pull tasks from overloaded peers [15, 16].

Location-Aware. These strategies prefer geographically or topologically closer nodes [11, 76], reducing communication delays and improving throughput.

Hybrid. Many systems combine multiple strategies, using, for instance, resource-based load balancing in normal conditions but switching to a demand-based method during hotspots [31, 43, 44, 63].

3 MEERKAT

In this section, we introduce Meerkat, a hybrid fault tolerance protocol designed for the unique challenges of IoT-based stream processing. Meerkat combines centralized deployment via a Global Replication Manager (GRM) with decentralized runtime adaptation through a Local Load Balancing (LLB) protocol. It extends the standard coordinator–worker model used in cloud-based SPEs with local mechanisms that let workers optimize execution based on their own resource constraints. We begin by motivating the need for Meerkat (Sec. 3.1), then present its layered architecture (Sec. 3.2), and finally describe its operational phases for scalable recovery and load regulation (Sec. 3.3).

3.1 Motivation

IoT-based stream processing engines (SPEs) operate under fundamentally different constraints than traditional cloud systems. Edge devices are resource-limited and volatile, topologies are tree-shaped and heterogeneous [14, 77], and shared infrastructure like durable brokers or distributed file systems is absent. These conditions make fault tolerance protocols such as UB, CH, and AS ineffective without major adaptation. We identify four core limitations.

P1. Lack of replayable sources. In contrast to cloud systems, which rely on Kafka-like brokers to replay data and discard local buffers, IoT settings offer no such guarantees. Once emitted, data cannot be recovered unless explicitly retained. To preserve correctness, protocols must buffer in-flight data along the full path to the sink, which alone can confirm safe delivery. This forces centralized trimming based on global progress, burdening the coordinator with continuous tracking and inflating the number of trimming confirmations as topologies grow.

P2. Higher storage requirements. Fault tolerance protocols depend on persistent storage for intermediate state. Cloud systems provide this via high-throughput, reliable shared storage such as HDFS [67]. In IoT settings, this model fails: devices are bandwidth-constrained, intermittently connected [59, 77], and lack shared infrastructure [14]. Remote writes are impractical, while local storage is unreliable unless explicitly replicated. Since most protocols lack built-in replication [19, 42], failures can result in data loss. Both remote and local storage thus require costly adaptations in cloud-edge environments.

P3. Unsupported topology assumptions. Many cloud protocols [19, 41, 52] assume hot-standby nodes can be launched on demand, typically by spawning virtual machines. In IoT deployments, however, device failures require physical replacement [14, 59], and idle capacity is rare [34, 53]. As a result, re-routing or replication must operate within a fixed hardware set, without assuming elastic capacity.

P4. Heterogeneity and resource constraints. IoT deployments span from low-power sensors to high-end servers [59, 78], with varying CPU, memory, and network bandwidth. However, existing protocols assume homogeneous clusters [19, 35, 41] and lack fine-grained modeling of per-device constraints. Furthermore, intermittent connectivity and resource heterogeneity [14, 59] complicate the use of heavyweight mechanisms like checkpointing or replication, which rely on reliable links and stable capacity.

These limitations show that assumptions of state-of-the-art approaches, such as all-to-all connectivity, homogeneous resources, and shared infrastructure do not hold in edge-cloud environments.

To overcome these challenges and enable fault-tolerance in cloud-edge environments, we propose Meerkat. It modifies the query plan

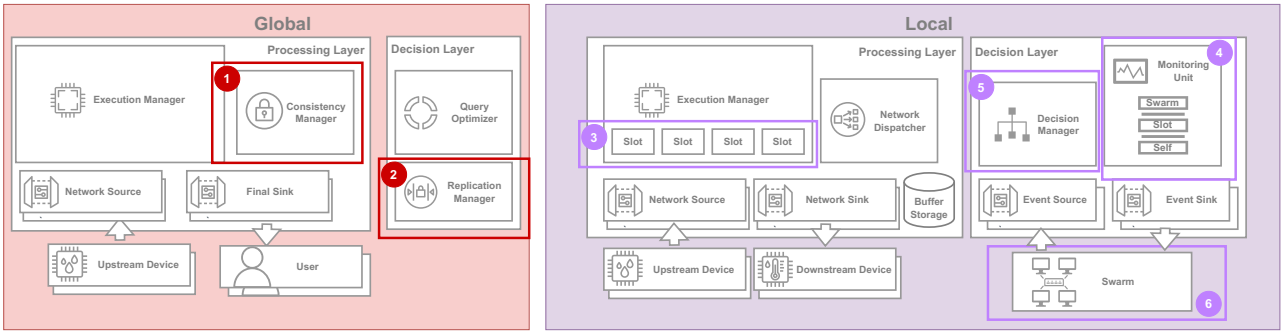


Figure 3: Architecture of the Meerkat protocol.

to introduce redundant paths and parallel operators without requiring data retention (P2). Figure 2(a) shows how Meerkat augments a logical DAG G with additional operators. For example, instead of deploying operators only on N1, N2, and N5, Meerkat inserts operators on N4 and routes data along a second path from N1 to N4. This approach eliminates the need to buffer tuples for retransmission, reducing memory usage and removing trimming coordination. As a result, workers can make progress independently, even if the coordinator is temporarily unavailable. Duplicate tuples are removed at the sink using lightweight deduplication (Sec. 4.3), ensuring correctness without extra storage overhead.

Once deployed, Meerkat's decentralized load balancing (DLB) enables each node to monitor its own and neighbors' load. In Figure 2(b–d), N4 detects overload (P4), notifies the CRM, and offloads part of its work to N3 by rerouting data from N1. This adaptive model avoids reliance on hot-standby nodes or overprovisioned resources (P3). The coordinator is informed of any updates to maintain global consistency.

These mechanisms form the foundation of Meerkat's hybrid design. We now describe the architecture that enables them, separating global and local responsibilities.

3.2 Architecture

We now present the architecture of Meerkat and describe its main components, which integrate global topology knowledge with local resource awareness (Fig. 3). To support this, each device is structured into a Processing Layer, responsible for ingesting and forwarding data, and a Decision Layer, which manages monitoring, adaptation, and coordination tasks. This separation supports scalable execution and enables local control while maintaining global consistency.

Global. The global *Processing Layer* contains the *Final Sink*, where query results are collected, and the *Consistency Manager* (1), which suppresses duplicate outputs and propagates per-origin sequence numbers to ensure recovery safety. Both components operate at the global level because they require a consistent view of system-wide execution to coordinate correctness across replicas and ensure deterministic output.

In the global *Decision Layer*, the *Replication Manager* (2) is responsible for (i) placing replicas during initial query deployment and (ii) verifying node candidates during runtime offload requests. It maintains a consistent view of the system state and enforces placement validity checks before migrations proceed.

Local. On each worker, the *Processing Layer* ingests data through *Network Sources*, routes it to the *Execution Manager*, and forwards outputs via *Network Sinks*. Each task is encapsulated in a platform-independent *Slot* (3), a logical unit sized proportionally to the device's capacity. This abstraction allows Meerkat to match task granularity to device resources, improving placement flexibility on heterogeneous hardware (P4). On systems that lack native task isolation, Meerkat emulates execution boundaries at the level of decomposed subqueries rather than individual slots, using buffered state and metadata to preserve processing consistency.

The *Decision Layer* includes the *Monitoring Unit* (4), which collects local and neighbor metrics to detect imbalance and trigger adaptation. This unit includes the *Status Monitor* (Swarm neighbors), the *Slot Monitor* (per-slot usage), and the *Self Monitor* (device-wide capacity). These feed into the *Decision Manager* (5), which reacts to imbalance by throttling input or initiating migration. To execute such actions, it interacts with the node's *Swarm* (6), a dynamically discovered set of nearby peers used for routing and offload. Together, these local components enable each worker to autonomously detect overloads and take corrective action, thus avoiding a centralized bottleneck (P1, P3). This layered design supports both scalable execution and adaptive runtime behavior.

3.3 Meerkat Phases

Building on the layered architecture, Meerkat's runtime behavior is structured into five coordinated phases. Each phase contributes to system stability by supporting decentralized adaptation, topology-aware offloading, and safe recovery.

1) Swarm Formation and Device Registration. When a device joins the system, it first registers with the central coordinator to retrieve its immediate neighborhood and configuration metadata. This includes the current slot size and a BFS-based list of nearby peers. The device then performs a handshake with these neighbors, announcing its presence and declaring its slot capacity—derived from its resource profile relative to the weakest node. This lightweight peer discovery process, inspired by P2P overlays [62], ensures accurate slot accounting and allows newly joined devices to contribute to load balancing. The discovered peers are added to the device's Swarm (6). Once devices are registered and their neighborhood is formed, each node begins local monitoring to detect emerging hotspots and load imbalances.

2) Local Monitoring and Overload Detection. Each device partitions its compute into logical slots, each hosting a single task. Devices with more resources provide more slots. Internally, three monitors collect metrics: one for the device's own limits (Self Monitor), one per slot (Slot Monitor), one from the neighbors (Status Monitor), all part of the Monitoring Unit 4 . This monitoring pipeline enables decentralized overload detection. When the usage exceeds threshold levels, either per slot or device-wide, a candidate offload target is selected from the Swarm. By structuring monitoring at multiple levels, Meerkat enables early detection of both localized and global imbalance. This allows rapid response before system-wide bottlenecks emerge. Detected overloads then trigger reconfiguration.

3) Offload Proposal and Candidate Selection. The local device submits the candidate offload target to the global controller. The decision logic is handled by the local Decision Manager 5 , which checks load thresholds and generates the request. At the coordinator, the Replication Manager 2 validates whether this reassignment preserves global constraints such as connectivity, past placement state, and redundancy. If necessary, the Replication Manager may augment connectivity (e.g., by link spawning). If the candidate is infeasible, an alternative is requested. This phase preserves system safety while enabling devices to propose adaptations autonomously. It eliminates the need for fine-grained, centralized control while retaining correctness.

4) Global Plan Update and State Migration. Once approved, the coordinator updates the global placement metadata and notifies the involved devices. The source worker packages the operator's state and buffered tuples, and transfers it to the target slot, a unit of execution defined in the local Processing Layer (Slot 3). The new device then resumes execution, allowing recovery to complete without input replay or checkpoint restoration, and supporting failure-free migration under overload.

5) Continuous Decentralized Regulation. Even after a successful offload, the Monitoring Unit 4 continues collecting metrics. The Decision Manager 5 uses this input to trigger further adaptations if needed. This control loop enables ongoing tuning as workloads shift or new hotspots arise. Coordinator involvement is kept minimal, as validation is only required when global constraints are impacted.

As a result, each node self-regulates itself with minimal coordination overhead (as shown in Sec. 6.2.2) $\leq 0.8\%$ CPU and $\leq$ 2KB/s control traffic per node, using localized metrics and neighborhood insight to maintain system balance. This decentralized scheme complements the global controller and enables Meerkat to scale across large, volatile environments.

4 GLOBAL REPLICATION

Meerkat's global controller is responsible for ensuring system-wide fault tolerance through intelligent replica placement. Unlike traditional systems that either pause execution during failures or rely on costly backups, Meerkat proactively replicates operators along disjoint paths in the network. These global decisions are made during query deployment and adjusted at runtime to account for device heterogeneity and network volatility.

The goal of global replication is to preserve end-to-end availability and throughput in the face of failures without overprovisioning. To this end, Meerkat strategically places redundant operators across the resource topology to create fault-isolated execution paths. These placements must obey constraints on resource capacity, path disjointness, and network connectivity to ensure both resilience and efficiency.

We formalize this as the Replicated Operator Placement Problem (ROPP), a constrained variant of the classic operator placement problem. ROPP captures the global replica placement challenge as an optimization task that balances communication cost, fault isolation, and device constraints. We present this problem next (Section 4.1), followed by our deployment-time and runtime placement strategy (Section 4.2), and finally show how correctness is preserved during replicated execution (Section 4.3).

4.1 Placement Problem

In IoT-based SPEs, efficient operator placement is crucial due to device limitations, network constraints, and real-time latency demands [24, 27, 28, 47, 74]. We define the *Replicated Operator Placement Problem (ROPP)* as a specialized extension of the commonly studied operator placement problem [22]. The operator placement problem generally deals with assigning operators in a dataflow (or query) DAG to a set of resources (e.g., edge or cloud nodes) to minimize a cost function (e.g., communication overhead or processing latency) subject to resource constraints.

As in the general operator placement formulation, ROPP considers a global resource topology, a dataflow DAG representing the query, and an optimization objective such as minimizing communication overhead or avoiding resource overutilization [2, 22]. However, ROPP extends the classic model by incorporating fault tolerance constraints. Specifically, it assumes that the placement of primary operators is already determined, for example by a greedy strategy such as Bottom Up planning [61], and focuses on selecting valid locations for their replicas. To ensure resilience, each replica must be placed along a data path that remains physically disjoint from the path of its corresponding primary. This disjointness is crucial to prevent a single failure from simultaneously affecting both instances of an operator [42, 49].

Formally, let $G_{\text{app}} = (V_{\text{app}}, E_{\text{app}})$ denote the query plan DAG (covering both primary and potential replicated operators) and $G_{\text{res}} = (V_{\text{res}}, E_{\text{res}})$ be the resource DAG (representing IoT devices, their capacities, and network links). Given a set of operator replicas $V_{\text{replica}} \subset V_{\text{app}}$ and an existing placement for $V_{\text{app}} \setminus V_{\text{replica}}$, ROPP seeks a mapping:

$$x = \langle x_{i,u} \rangle, \quad i \in V_{\text{replica}}, u \in V_{\text{res}} \tag{1}$$

that assigns each replica i to exactly one valid node u, respecting the following constraints.

C1. Path Disjointness. Each replica operator must be placed so that its end-to-end path $\pi_i = (v_0, e_1, v_1, \ldots, e_k, v_k)$ to the sink shares no edge with the path $\pi_{i'}$ of the corresponding primary operator i', i.e.,

$$\pi_i \cap \pi_{i'} = \emptyset, \quad \forall i \in V_{\text{replica}} \tag{2}$$

C2. Resource Capacity. Device resource limits must not be exceeded. Let r_i denote the resource demand of operator i and R_u the capacity of node u:

$$\sum_{i \in V_{\text{app}}} x_{i,u} \cdot r_i \leq R_u, \quad \forall u \in V_{\text{res}} \tag{3}$$

C3. Topology-Aware Resilience. Each replica must be placed so that a valid source–sink path exists through the hosting node, even under a dynamic hierarchical topology. If fewer than two disjoint paths are available at placement time, the system may attempt to spawn an ephemeral link to create one. Let $\pi(u)$ denote the set of edges on the sink-bound path through node u, and let $\rho(i)$ and $\rho(i')$ be the replica and its primary, respectively. Let $\eta(u) = 1$ if node u supports ephemeral link creation:

$$x_{i,u} = 1 \Rightarrow \left[\left(\pi(u) \cap \pi(u') = \emptyset \right) \vee \eta(u) = 1 \right], \quad \forall i \in V_{\text{replica}}, u' = \text{primary}(i) \tag{4}$$

This ensures that the selected placement can maintain fault isolation via disjoint paths (C1) even under changing topologies, by adapting path structure if necessary.

C4. Unique Assignment. Every replica must be assigned to one and only one device:

$$\sum_{u \in V_{\text{res}}^i} x_{i,u} = 1, \quad \forall i \in V_{\text{replica}} \tag{5}$$

This constraint ensures that replicas r are placed on separate devices to preserve fault isolation, as co-locating them would undermine resilience in volatile cloud-edge environments. While our current model assumes $r = 1$, Meerkat can be extended to higher replication factors $r > 1$, e.g., based on the number of available disjoint source–sink paths, to improve availability at the cost of increased duplicate detection and coordination overhead.

C5. Binary Decision Variables. Each placement decision is binary:

$$x_{i,u} \in \{0,1\}, \quad \forall i \in V_{\text{app}}, u \in V_{\text{res}}^i \tag{6}$$

We define a cost function:

$$F(x) = w_N \cdot N(x) + w_P \cdot P(x) \tag{7}$$

where $N(x)$ is the total communication overhead induced by the operator placement, $P(x)$ captures penalties for resource overutilization, and w_N, w_P tune their relative weights.

Exact solutions to ROPP are computationally intractable at scale. Classical methods like Integer Linear Programming (ILP) [2] are impractical in large, dynamic IoT networks due to their global state assumptions. Heuristics like Genetic Algorithms or Simulated Annealing [32, 54, 55, 70] often require domain-specific tuning and cannot easily enforce disjoint path constraints. We therefore adopt a two-stage solution strategy in Meerkat. In the first stage, a lightweight Breadth-First Search (BFS)-based algorithm assigns replicas to ensure disjoint paths whenever feasible. In the second stage, a local refinement heuristic adapts placements to respond to capacity shifts or transient failures without requiring full re-optimization. Our hybrid strategy achieves only minor placement overhead compared to heuristic approaches, while significantly outperforming cost-based placement strategies (see Section 6.4.2).

4.2 Replicated Operator Placement and Offload

In an IoT-based SPE, efficiently placing replicated operators across a large-scale and dynamic network is crucial for enhancing fault tolerance without overloading the system. For this purpose Meerkat uses GRM that performs a two-stage procedure: (i) one-shot initial placement at deployment time and (ii) task offload at runtime whenever a device becomes overloaded.

Algorithm 1 Replica placement (initial deployment)

```
1:  function ReplicaPlacement(G_res, G_app)
2:      levels ← ComputeBFSLevels(G_res)
3:      for each query q in G_app do
4:          paths ← findAllPaths(q.leaf, q.sink, G_res)
5:          if |paths| < 2 then
6:              paths ← trySpawnLink(paths, levels)
7:              if |paths| < 2 then
8:                  warn: newNodeRequired()
9:                  continue
10:         (p_1, p_2) ← pickTwo(paths)
11:         placePrimary(p_1, q); placeReplica(p_2, q)
12:     return success
```

Initial Placement: Establishing a starting point that considers IoT network constraints is a crucial step, as it affects the search space traversal and optimization. Therefore, our GRM uses the placement calculated by the Bottom-Up strategy as the initial state. The Bottom-Up strategy is a common, heuristics-based approach which greedily pushes operators towards the data sources and thus minimizes inter-device traffic in tree-shaped topologies [61]. The heuristic executes in $O(|E_{\text{res}}|)$ time, so the initial placement is computed in linear time with respect to the physical links.

The GRM then guarantees at least two disjoint source–sink routes for every query. Algorithm 1 (Line 2–12) shows the BFS-based procedure. Lines 4-9 enumerate all existing paths between the leaf and the sink. If fewer than two exist, Line 6 attempts to trySpawnLink, i.e., attach the leaf to a node on another BFS layer. If that fails, Line 8 warns the administrator that an additional device must be added. Finally, the two routes are selected (Lines 10-11) and the primary and replica operators are placed accordingly. Because the algorithm touches every vertex and edge at most once, it runs in $O(|V_{\text{res}}| + |E_{\text{res}}|)$.

The Bottom Up start state narrows the search space, while the BFS refinement helps avoid poor local choices by systematically exploring alternative routes level by level. While this heuristic does not guarantee globally optimal placements in all topologies, it often finds short disjoint paths early and balances efficiency with placement quality in practice.

Task Offload. After deployment, load fluctuations or incorrect initial assumptions may still cause individual devices to become overloaded. In such cases, the affected worker requests an offload from the coordinator, proposing a candidate node to take over part of its subquery. Algorithm 2 outlines this process.

First, isReachableFromPinnedNodes ensures that the proposed candidate node lies on a valid route from the pinned source to the pinned sink. If it is unreachable, the offload fails immediately (Lines 2–3). Next, on Lines 4–6 isDisjoint checks whether migrating the sub-query to candidate would overlap with the existing replica path. If so, the coordinator attempts to spawn a new link at the same or higher BFS level, mirroring the logic used during initial placement. If this still fails to preserve two distinct routes, the offload is rejected (Lines 7–8). The GRM then demands the worker to send an alternative candidate for the offload.

Each reachability and disjointness test is at most $O(|E_{\text{res}}|)$, and the actual migration is dominated by state transfer time, which we evaluate in Section 6.

Algorithm 2 Candidate verification and task offload

```
1: function OffloadTask(subQuery, cand)
2:     if not isReachableFromPinnedNodes(cand, subQuery) then
3:         return fail( candidate unreachable )
4:     if not isDisjoint(cand,subQuery) then
5:         if not trySpawnLink(cand) then
6:             return fail( no disjoint route )
7:     stopSubQuery(subQuery.originWorker)
8:     redeploySubQuery(cand,subQuery)
9:     return success
```

Once a replica is successfully placed, both the primary and redundant operators process data concurrently. This naturally results in duplicated outputs at the sink. To ensure correct query semantics, Meerkat integrates a lightweight detection mechanism that filters out such duplicates, as detailed later in Section 4.3.

4.3 Correctness in Replicated Execution

Meerkat ensures exactly-once* semantics in case of non-failing data-producing sources by suppressing duplicate outputs from replicated operators and enabling safe recovery without replay. This requires filtering redundant outputs at the sink and aligning the recovered state with global progress.

Each tuple τ carries an origin ID id_O, a monotonically increasing sequence number seq_O, and an event-time timestamp w_O. Operators in Meerkat process data in an event-driven fashion: input updates state, and output is triggered only once watermarks advance far enough to close a window or join. Each operator maintains state per origin, updated strictly in seq_O order.

To restore logical order, Meerkat applies in-order processing only at stateful operators and at the final sink. Each operator employs a *multi-origin watermark processor* that buffers updates per id_O in a priority queue and applies them only when $\text{seq}_O = \text{last seen} + 1$. This blocks progress on gaps and ensures consistent per-origin state. Aggregations trigger only when all inputs have passed a watermark. Join output is produced only after both sides are watermark-complete. This guarantees that no partial or invalid results are emitted.

The sink suppresses duplicates by tracking the highest seq_O seen per origin. Incoming tuples with lower or equal seq_O are discarded, eliminating redundant outputs from replicas.

When a primary fails at s_f, a shadow replica continues to s_r. A new instance resumes using the shadow's state, aligning to s_r and w_r piggybacked from the sink. It sets $\text{seq}_O = s_r + 1$, updates its watermark to w_r, clears buffered tuples, and suppresses the first output to avoid duplicating partial aggregates. It then proceeds with fresh input, mirroring the replica's behavior.

For example, if the primary failed after tuple 5 and the replica reached tuple 11, the sink provides $s_r = 11$ and $w_r = 8$. The recovering operator skips to $\text{seq}_O = 12$, adopts watermark 8, and processes only new tuples.

This strategy covers both early and late failure cases. If the failure occurs before output, the residual state is bypassed. If it occurs after, duplicate output is prevented by emission suppression.

Proof Sketch. Let $S(t)$ be the operator state before processing τ_t. Since state evolves deterministically from ordered input and watermark progression, output is uniquely defined.

A recovering operator starts from $S(s_f)$, resets $\text{seq}_O = s_r + 1$, $w_O = w_r$, clears inherited buffers, and suppresses its first output.

This ensures: (1) already-processed data is skipped, (2) no result is emitted until progress advances, and (3) state/output remains consistent with the replica. Thus, Meerkat guarantees exactly-once* semantics across failure scenarios.

5 LOCAL LOAD BALANCING

The highly dynamic nature of IoT deployments introduces transient failures, heterogeneous device capabilities, and fluctuating latencies that can overload some devices while leaving others idle. Meerkat addresses these challenges with LLB protocol that continuously adapts task distribution to runtime conditions. It comprises three mechanisms: threshold-aware redistribution to handle local overloads (Sec. 5.1), hierarchical knowledge propagation to coordinate across volatile topologies (Sec. 5.2), and replication-aware slot migration to maintain correctness during reconfiguration (Sec. 5.3). While these mechanisms govern load balancing decisions and coordination, Meerkat delegates actual deployment and migration of tasks to the underlying SPE.

5.1 Threshold-Aware Load Redistribution

To handle load volatility in IoT environments, Meerkat employs a threshold-based redistribution model that dynamically identifies overloaded nodes and selects a candidate for task migration. Inspired by techniques used in structured peer-to-peer systems [62] and slot-based schedulers like Flink [20], Meerkat subdivides each device into logical slots. Each slot corresponds to a processing task and is monitored independently.

We define the normalized load of a device L_i as the average of its slot loads:

$$L_i = \frac{1}{|S_i|} \sum_{s \in S_i} \text{load}(s), \tag{8}$$

where S_i is the set of slots on device i, and $\text{load}(s)$ denotes the aggregated resource pressure observed on slot s.

Meerkat's load function is configurable and designed to support heterogeneous resource indicators such as CPU utilization, memory usage, I/O pressure, or energy consumption. These metrics are first normalized to a common scale (e.g., relative to hardware limits or platform-specific baselines) and then combined into a weighted sum:

$$\text{load}(s) = \sum_j w_j \cdot m_j(s),$$

where $m_j(s)$ is the normalized value of metric j for slot s, and w_j is its corresponding weight. The weights can be set statically based on domain knowledge or adjusted dynamically depending on system priorities.

In our current implementation, which is tailored to resource-constrained edge deployments, we focus on memory consumption and runtime latency. Specifically, we track the available buffer capacity and latency as primary indicators of overload. This setup allows Meerkat to detect both memory saturation and performance degradation. The resulting load score L_i is then compared to a predefined

Algorithm 3 Slot migration protocol executed on each worker

```
1:  function SlotMovement(slot)
2:      if slot.isHeavy() then
3:          topC ← {}
4:          for each node ∈ swarm do
5:              if node.isLight() then
6:                  topC.add(node)
7:          artifacts ← bufferStorage.getArtifacts(slot)
8:          for each node ∈ topC do
9:              if node.isActive then
10:                 notifyReplicationManager(slot,node)
11:                 transfer(artifacts,node)
12:                 return true
13:     return false
```

threshold to determine whether the node is considered overloaded. In particular, a device is considered *heavy* if $L_i > T$, where T is a threshold derived from the device's hardware profile or configured empirically. When a node exceeds this threshold, the system executes the following redistribution protocol:

- **Primary strategy:** Migrate the *lightest* task t such that $L_i - \text{load}(t) < T$.
- **Fallback strategy:** If no such t exists, migrate the *heaviest* task to reduce peak load.
- **Safety check:** Ensure target node ℓ satisfies $L_\ell + \text{load}(t) \leq T$.

This conservative transfer scheme minimizes unnecessary migrations and avoids ping-pong effects, which are known to destabilize load balancing in edge environments [10]. By redistributing tasks that relieve hotspots while not overwhelming offload candidates, Meerkat achieves a stable system state under continuous workload shifts.

5.2 Hierarchical Knowledge Propagation

Effective task redistribution in IoT deployments requires both fast reactions to overloads and proactive anticipation of future imbalances. Meerkat achieves this by propagating load and status information through a hierarchical mechanism that combines reactive local communication with predictive global updates.

At the core of this design is the concept of a Swarm (see Sec. 3.3), a dynamically maintained local neighborhood that gives each node partial visibility into the system. This structure allows Meerkat to make localized decisions even in large-scale deployments, where full-system coordination introduces bottlenecks.. Each node builds its Swarm based on its topology position and available metadata from the coordinator. The Swarm may include 1-hop parents, children, and siblings in the dataflow graph. A larger Swarm increases the chances of finding viable offload targets but also raises the maintenance overhead due to more frequent status exchanges. The size and update frequency of the Swarm are therefore configurable parameters, enabling system designers to trade off between responsiveness and communication effort.

Direct Propagation (Local Feedback): Swarm neighbors exchange local load metrics using low-latency links, enabling fast, decentralized responses to emerging imbalances. This peer-to-peer communication forms the first tier of Meerkat's knowledge propagation.

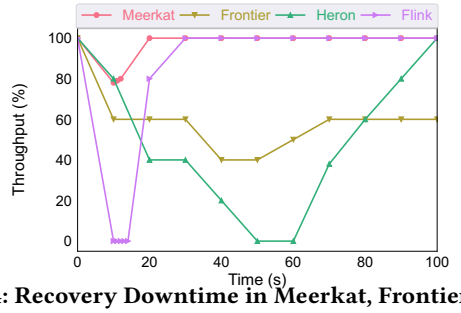


Figure 4: Recovery Downtime in Meerkat, Frontier, Heron, and Flink

Upon detecting a threshold violation in its local load metrics:

(1) A node issues a direct feedback message to its upstream producers, requesting a reduction of input rate or a temporary suspension of data.
(2) Simultaneously, it contacts a set of Swarm neighbors to explore immediate offload options, such as rerouting part of the task or replicating the execution locally.
(3) The Decision Manager on the overloaded node collects the status of contacted neighbors and selects a suitable mitigation strategy, i.e., triggering an offload request to the coordinator.

This proximity-based mechanism acts as a fast-response strategy that contains localized overloads before they propagate further. By relying on direct neighbor knowledge, it reduces coordination overhead and enables swift mitigation.

Indirect Propagation (Predictive Coordination): For network-wide optimization, Meerkat maintains a coarser, multi-hop view of resource status. While direct propagation focuses on real-time feedback from immediate Swarm neighbors, indirect propagation provides a multi-hop, delayed view of the wider system state.

In this mode, each device periodically aggregates and forwards compact summaries of its load metrics to a configurable subset of more distant peers. If no immediate offload target is available, this mechanism also allows overloaded nodes to issue feedback messages to more distant upstream devices, requesting a reduction in output rate (so-called backpressure handling). This serves as an additional mitigation strategy when local options are exhausted.

Unlike direct propagation, which enables immediate mitigation of localized issues, this indirect channel is less reactive (i.e., not event-driven but periodic) and trades responsiveness for global foresight. It helps prevent cascading overload chains that arise when local decisions are made without broader context. By combining both strategies, Meerkat balances short-term responsiveness with long-term planning.

5.3 Slot Migration

To dynamically redistribute workload under overload conditions, Meerkat drives the migration process by selecting which slot to offload and identifying suitable target devices based on current load and topology. This migration logic is tightly coupled with the GRM to ensure that every offload decision preserves consistency and respects global replication constraints.

Algorithm 3 outlines the migration procedure. The worker first detects whether a slot has become overloaded (Line 2). Upon overload,

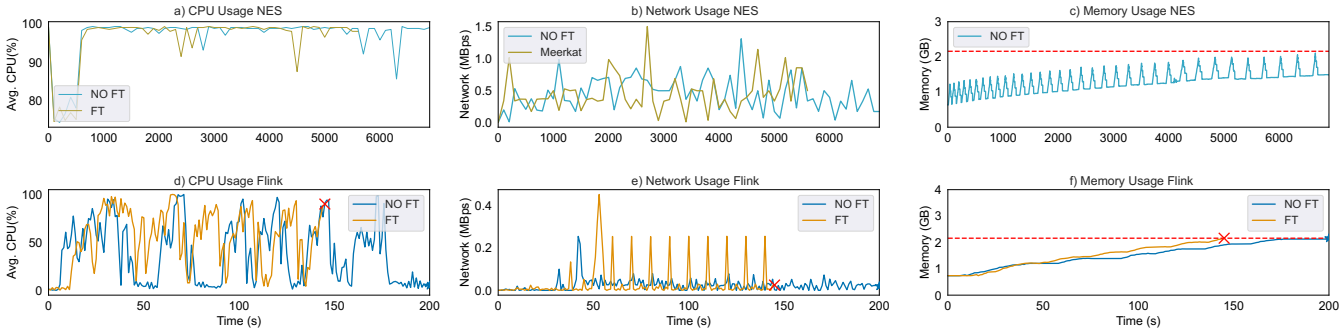


Figure 5: CPU, network, and memory utilization during stateful queries submission in NES and Flink.

it searches its Swarm neighborhood for lightly loaded and reachable peers (Lines 3–6). Candidate devices are ranked by load and topological proximity, favoring targets that reduce migration delay and inter-device communication. While Meerkat uses this simple distance-aware heuristic, other strategies such as load prediction, history-based stability, or centralized optimization have been explored in prior work on stateful operator migration [21, 41, 50, 71]. Meerkat's design trades generality for speed and deployability in resource-constrained IoT environments.

Once candidates are selected, the worker collects the runtime artifacts of the slot (Line 7). These include buffered input, watermark and sequence tracking metadata, and the internal operator state. This information forms the slot's complete execution context and is required to resume processing without data loss or semantic violations. This information is passed to the SPE, which handles the transfer and ensures correct operator restoration on the destination device.

The worker then iterates through the ranked candidate list (Line 8). For each active target node, the Decision Manager first contacts the GRM to verify that migrating the slot to this node preserves replication invariants (Line 10). If the GRM approves, the migration proceeds (Line 11). The protocol then returns success (Line 12).

If the target is inactive or the GRM rejects the transfer, the worker continues scanning the remaining candidates. The migration fails gracefully only after all candidates have been tried (Line 13). Meerkat does not support dynamic repartitioning of oversized tasks. When no offload target can absorb a heavy task, it issues backpressure signals to upstream operators. Future extensions could integrate partition-aware approaches to enable task splitting or multi-node replication.

6 EVALUATION

In this section, we experimentally evaluate Meerkat in NES. In Section 6.1, we introduce our experimental setup. After that, we present three sets of experiments. First, we evaluate the resource efficiency of Meerkat in NES and compare it to the other approaches in state-of-the-art SPEs (Sec. 6.2). Second, we analyze the impact of Meerkat with different hyperparameters on the system's throughput and latency (Sec. 6.3). Third, we explore the decision time of Meerkat on large-scale topologies (Sec. 6.4).

6.1 Experimental Setup

We run our experiments on four types of hardware. *Type A:* a Linux server with an AMD EPYC 7742 2.25GHz CPU (64 physical cores) and 1TB of main memory. *Type B:* a hierarchical cluster of eight Linux servers with 2 Intel Xeon Silver 4216 2.20 GHz CPU (32 physical cores), 500 GB of main memory, and 100 Gbit Infiniband connection. *Type C:* 256 Rasberry Pis united into a Kubernetes cluster. Each node is equipped with two virtual CPUs (vCPUs) at 1.2 GHz, 2 GB RAM, and 10 GB disk space. *Type D:* Raspberry Pi 4 Model B with 1.5 MHz Quad-Core, 2GB RAM.

6.1.1 Workloads. To evaluate performance across diverse operator demands, we select five NEXMark queries [72]: NxQ_0, NxQ_2, NxQ_4, NxQ_6, and NxQ_8. These span stateless, stateful, and join-based operators with varying resource profiles [19, 39, 47, 50, 53]. Aggregation windows are limited to 4s due to device memory constraints. NxQ_0 is a stateless pass-through query that we use as a throughput baseline. NxQ_2 is a CPU-bound filter on auction IDs. NxQ_4 performs windowed aggregation with moderate memory use. NxQ_6 extends this with overlapping windows and time tracking. NxQ_8 is a count-based join between person and auction streams and is the most memory-intensive. Together, these queries model common streaming patterns and allow us to assess Meerkat under varied compute and memory loads.

6.2 System Comparison

In this section, we compare Meerkat's performance against other state-of-the-art fault tolerance (FT) solutions in environments with constrained resources. The baseline systems employ different fault tolerance mechanisms. Flink [19] uses coordinated checkpointing with periodic snapshotting and recovery via input replay. Heron [48] follows a passive standby model, where failed tasks are relaunched on idle containers. Frontier [59] uses active replication, executing multiple copies of each operator and switching to a replica upon failure. Our experiments focus on four aspects: recovery downtime (Sec. 6.2.1), resource utilization (Sec. 6.2.2), throughput (Sec. 6.2.3) and scalability overhead (Sec. 6.2.4).

6.2.1 Recovery Downtime. This experiment measures how Meerkat, Frontier [59], Heron [48], and Flink [19] recover from a single worker failure. We deploy the stateless query NxQ_0 on three Type D devices, using two as worker nodes and one as a master node. At time 0, we fail one worker and observe the systems' relative throughput (%) over time, capturing how each fault tolerance approach responds to the disruption.

Results. Figure 4 presents the throughput over time for Meerkat (▬), Frontier (▬), Heron (▬), and Flink (▬) following the failure at $t = 0$. Meerkat shows only a brief 22% throughput drop for approximately 3 sec, then returns to 100%. In contrast, Flink pauses the entire job for 5 s, dropping throughput to 0, while it restarts the failed

worker. Heron exhibits similar behavior with longer downtime (10 s), resulting in a complete throughput stall for that interval.

Frontier, which also deploys parallel processing similar to Meerkat, never fully stalls at 0%. But its throughput hovers between 60% and 40% for about 30 s before stabilizing around 60%. Because Frontier lacks a built-in mechanism to restart the failed worker, it cannot recover to full capacity, persisting at reduced throughput. In contrast, Meerkat both avoids a total throughput drop and rapidly scales back to 100% after its quick recovery phase. Overall, these results underscore Meerkat's ability to handle worker failures with zero downtime and maintain high throughput in resource-constrained IoT deployments.

6.2.2 Resource Utilization. In this experiment we explore how fault tolerance influences CPU, memory, and network usage in a resource constrained setting. The experiment deploys the stateful query NxQ_4 multiple times on three Type D devices, each providing 3.5 GB of RAM. One device functions as a Coordinator (or Master), and two others host a Worker. All three orginize a sequential topology. We analyze NES (with and without Meerkat) and Flink (with and without its default FT), observing how many queries can be launched before hitting resource exhaustion. Each ★ marks a successful query deployment (NES and Flink without FT), while ✗ indicates a failed deployment due to resource exhaustion (Flink with FT).

Results. Figure 5 compares CPU, network, and memory usage over time. For CPU usage, NES can deploy 102 queries successfully without FT. With Meerkat the optimizer continues to admit work: at $t \approx 450$ s the worker that hosts the aggregation starts offloading the next three queries to a neighboring device, after which we stop submitting further load. In both NES runs no failure occurs. By contrast, Flink manages 14 queries without FT and fails on the 11-th query when FT is enabled.

Figure 5a shows that NES with Meerkat keeps CPU utilization at around 26–32%. Enabling FT in Flink increases CPU pressure from 40% to 55% on average, with utilization reaching 95% just before the system crashes (Figure 5d). Although additional monitoring, deduplication, and heartbeat messages might have increased NES's CPU load, the CPU overhead remains negligible due to the small Swarm size and the streamlined implementation of deduplication (Section 4.3).

Regarding network utilization, Flink requires about 25% more bandwidth under FT than without FT, mainly because it transfers large checkpoint snapshots to the coordinator (Figure 5e). NES, on the other hand, shows only a $\approx 28\%$ increase with Meerkat, driven by periodic status updates rather than bulk checkpoints. The spike around $t \approx 450$ s in Figure 5b corresponds to Meerkat's load balancing protocol probing peers as saturation approaches and the consequent state offload.

Memory usage follows the same trend. NES consumes just $\approx 6\%$ more memory with Meerkat's FT enabled, compared to 16% for Flink. As shown in Figure 5c, Meerkat's proactive offloading keeps the device below the 3.5 GB limit. Starting from $t \approx 530$ s we observe three small drops, which indicate the offload of the partially constructed state of newly arrived queries. Meerkat in time notices the potential device overload and proactively moves the task to one of its neighbors. Enabling FT in Flink increases the size of in-flight checkpoint data, pushing memory usage beyond the 3.5 GB limit and triggering failure (Figure 5f). Overall, this experiment highlights

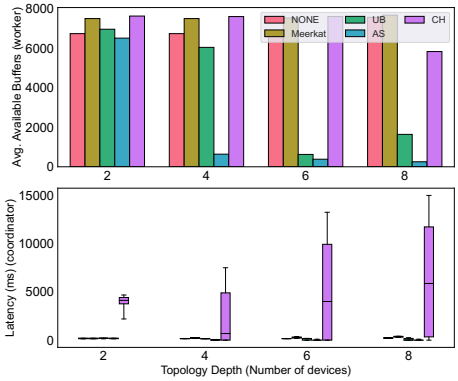


Figure 6: Average number of available buffers at the source worker and latency at the coordinator with growing depth.

that determining the right time of task offloading is crucial to keep the system performance and avoid underprovisioning.

6.2.3 Throughput Comparison. We finally compare overall throughput among Flink [20], Heron [48], Frontier (with $r = 2$) [59], and NES (with Meerkat) on Type C hardware using the CPU-bound NxQ_2 workload. The topology has three layers: a single source layer, an intermediate processing layer, and a sink layer. Meerkat's monitoring interval is set to 1 s, while Flink and Heron rely on passive standby for fault tolerance, and Frontier uses active standby.

Results. Our results reveal that NES with Meerkat processes around 1.84 million tuples per second. In contrast, the other systems reach significantly lower throughput: Frontier peaks near 26 k tuples/s, Flink around 30 k, and Heron near 34 k. Part of NES's speedup stems from its code-generated operator pipelines [78], which efficiently exploit modern hardware and minimize runtime overhead. In addition, Meerkat's lightweight decentralized FT approach, based on short-range monitoring and deduplication, maintains high throughput despite failures or load imbalances. In contrast, JVM-based Flink, Heron, and Frontier, running on Raspberry Pis with minimal memory, struggle with garbage collection overheads and the monolithic checkpoint or replication protocols. As a result, these systems cannot match NES's throughput in constrained edge scenarios.

Our results confirm that a hybrid fault tolerance strategy, combined with efficient operator pipelines, is well-suited for resource-limited IoT deployments.

6.2.4 Scalability Overhead. This experiment examines the throughput degradation observed in Figure 1 by reporting the average number of available buffers at the source worker (furthest from the coordinator) and the median latency at the coordinator for each protocol and depth. Results are shown in Figure 6.

Results. UB and AS show a steep decline in average buffer availability as depth increases. UB drops from 6956 at depth 2 to 1639 at depth 8, with a low of 625 at depth 6. AS falls more sharply, from 6506 to 253, already reaching 638 at depth 4. This confirms that as acknowledgment delays grow with depth, output buffers accumulate throughout the path. In contrast, NONE and Meerkat remain stable across all depths, with buffer availability increasing slightly from 6733 and 7495 at depth 2 to 7556 and 7668 at depth 8, indicating no backpressure buildup.

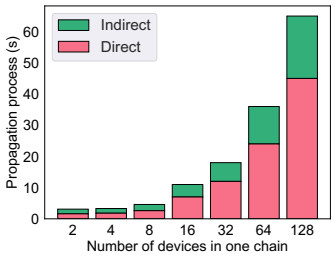


Figure 7: Indirect vs. Direct knowledge propagation.

Coordinator latency for CH rises from 4116 ms at depth 2 to 5861 ms at depth 8, reflecting the increasing cost of serializing and transmitting snapshots in the absence of shared storage. NONE, UB, and AS remain under 350 ms at all depths. Meerkat also stays consistently low, rising moderately from 190.5 ms to 346.0 ms. These results confirm that UB and AS are limited by memory, CH by centralized coordination delays, while Meerkat stays stable.

6.3 Meerkat Configuration Impact

In the following, we study the impact of Direct vs. Indirect knowledge propagation on propagation time (Sec. 6.3.1) and topology size on the time distribution during task movement (Sec. 6.3.2).

6.3.1 Knowledge Propagation. In this experiment, we measure the duration to propagate load-related information across a varying numbers of devices arranged in a sequential (chained) topology. We consider chain lengths ranging from 2 to 128 nodes, all of Type D hardware. Meerkat provides two strategies for knowledge propagation: direct and indirect approaches (see Section 5.2). We measure the total time needed for a load alert originating at the first node in the chain to reach the final node, thus capturing worst-case propagation latency.

Results. Figure 7 illustrates how propagation delay scales with the number of hops in a linear chain for both direct and indirect communication modes. Both approaches show approximately linear growth, reflecting how each additional hop contributes incrementally to the total propagation time. Across all chain sizes, direct propagation is consistently faster. For example, at 16 hops, it completes in approximately 4000 milliseconds, compared to 7000 milliseconds for indirect propagation. At 128 hops, direct updates require about 20 seconds, while indirect updates take around 45 seconds.

These results highlight the latency advantage of direct communication when disseminating status updates across the network. In contrast, the indirect mode, although slower, remains useful for maintaining broader system knowledge and informing long term planning. Accordingly, Meerkat adopts a combined strategy. It prioritizes direct updates for responsive local decisions while using indirect propagation to support coordinated adjustments across the wider network.

6.3.2 Task Movement. In this experiment, we evaluated the performance overheads of different system components when handling varying numbers of device requests. We specifically measure the time taken for four key operations: Neighbor Selection, Candidate Verification, Query Reconfiguration, and Task Movement. We use a diamond-shaped topology (two paths connected by one source node as leaf and one coordinator node as root) with 4, 8, 32, 64, and 128 devices. We run this experiment on the Type A hardware with the CPU-bound NxQ_2 workload.

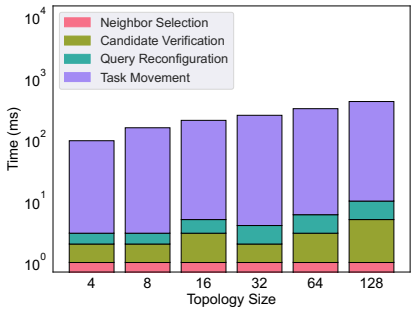


Figure 8: Task movement latency with varying topology size.

Results. Figure 8 presents the stacked bar chart illustrating the contributions of each component to the task movement latency under different request loads. Notably, the task movement component dominates the latency profile. For example, at four requests, task movement accounts for 6567 ms, whereas at 128 requests, it rises significantly to 12937 ms. In contrast, the neighbor selection time remains constant at 1 ms across all cases, and the candidate verification time shows only modest variation. The query reconfiguration time increases progressively from 2 ms at four requests to 24 ms at 128 requests, highlighting a scaling overhead in decision-making as the load increases.

These findings underscore that while the core decision and verification processes incur minimal delays, the task movement process is the primary contributor to overall latency, particularly as the number of device requests grows. This insight is critical for guiding further optimizations, particularly in scenarios where rapid reconfiguration is necessary under heavy load conditions.

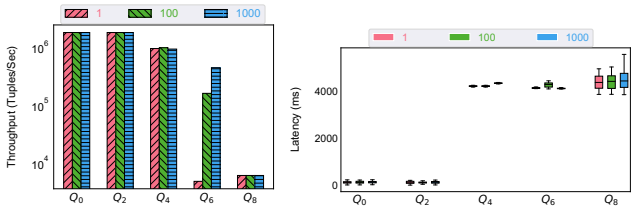
6.4 Overhead

In this set of experiments, we estimate the overhead of Meerkat in NES. Specifically, we show how different monitoring frequency influences various workloads (Sec. 6.4.1) and compare Meerkat's placement time to other placement strategies (Sec. 6.4.2).

6.4.1 Various Workloads. In this experiment, we assess the impact of monitoring frequencies 1, 100, and 1000 ms on five NEXMark queries: NxQ_0, NxQ_2, NxQ_4, NxQ_6, and NxQ_8. NES is deployed on a diamond-shaped topology with six workers (four intermediate, one source, one Coordinator), where the Coordinator runs on Type A hardware and the Workers on Type B. We measure throughput and latency at the final sink.

Results. Figures 9(a) and 9(b) show that NxQ_0, NxQ_2, and NxQ_4 achieve stable throughput and latency across all monitoring frequencies. NxQ_4, which performs aggregation, shows 47% lower throughput than NxQ_0 due to added processing. NxQ_6, a sliding window aggregation, is more sensitive: its throughput increases from 4.8k to 446k tuples/sec as monitoring frequency decreases. This reflects the cost of maintaining overlapping windows under frequent monitoring. NxQ_8, a count-based join, maintains stable throughput but shows greater variance in latency.

These results indicate that Meerkat supports diverse workloads efficiently, though some operators benefit from tuning the monitoring frequency.



(a) Throughput (b) Latency

Figure 9: Varying queries with different monitoring frequencies.

6.4.2 Placement Time. To assess the placement time of Meerkat, we conducted an experiment using data from the OpenCellid database [4]. Specifically, we extracted information about LTE, GSM, and UMTS towers located in Berlin, where LTE has 16147, GSM 29757, and UMTS 43602 towers as an example of a running IoT-based SPE. Based on each tower type, we construct a global worker topology within NES and measure Meerkat, heuristic Bottom-Up (it pushes computation as much to the edges as possible), and cost-based ILP operator placement strategies.

Results. The ILP approach, which attempts to optimize multiple constraints simultaneously, times out under the tested workloads. In practice, we observe that even for the smaller LTE topology. In contrast, the Bottom-Up algorithm and the Meerkat approach significantly reduce decision time. Bottom-Up places and configures operators in roughly 241 seconds for LTE, 500 seconds for GSM, and 1861 seconds for UMTS. In contrast, Meerkat requires about 267 seconds for LTE, 600 seconds for GSM, and 2034 seconds for UMTS. By focusing on a smaller set of objectives than ILP, these heuristic algorithms streamline the placement process and avoid exponential blowups in decision time. Overall, Bottom-Up and Meerkat deliver near-optimal operator placement while performing orders of magnitude faster than the exhaustive ILP approach.

7 RELATED WORK

Fault tolerance has been studied extensively in centralized engines, sensor networks, edge platforms, and decentralized systems. However, most existing solutions assume resource-rich environments or static topologies, limiting their applicability to dynamic and heterogeneous IoT deployments.

Cloud-based SPEs. Engines like Aurora [5], TelegraphCQ [23], Flink [19], and Heron [48] rely on checkpointing or active standby under assumptions of reliable connectivity and infrastructure. Later systems such as Chi [52], Rhino [53], and Megaphone [41] introduce incremental migration and lightweight reconfiguration, but still depend on centralized coordination. More recent approaches like Clonos [69] and CheckMate [68] explore decentralized checkpointing to reduce coordination overhead, yet still assume stable nodes and reliable storage. In contrast, Meerkat targets volatile networks and constrained devices. While retaining a central coordinator for placement, it decentralizes failure handling through local propagation and on-device failover, enabling progress under disconnections and heterogeneous conditions.

Wireless Sensor Networks. Systems like TinyDB [51] and Cougar [75] optimize energy use via in-network aggregation but support only static topologies and basic queries. They are not suited for stateful or dynamic workloads. Meerkat supports expressive, reconfigurable execution across volatile nodes while maintaining exactly-once guarantees.

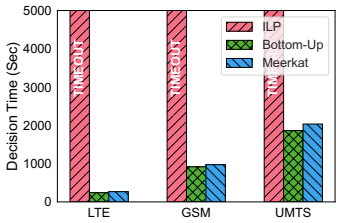


Figure 10: Decision time for the Berlin cell tower dataset.

Peer-to-peer SPEs. Structured overlays [62] and swarm-based clustering [36] offer decentralized routing and load balancing, tolerating churn for stateless tasks. However, they lack coordinated state handling and recovery support. Meerkat builds on these ideas, extending them to stateful workloads through coordinated migration and deduplication.

Edge-based SPEs. Platforms like Frontier [59], Azure IoT Edge [3], and AWS Greengrass [1] support edge execution but rely on cloud controllers and do not ensure continuous recovery. SEEP [34] and SDG [33] enable operator migration but assume uniform resources. Dhalion [35] and DS2 [45] monitor load but offer limited failure resilience. Meerkat integrates lessons from these systems into a hybrid model with fault isolation, zero-downtime failover, and decentralized scaling for multi-layered IoT networks.

8 CONCLUSION

This paper introduces Meerkat, a fault tolerance protocol designed for large-scale distributed sensor data processing that encapsulates sensor, edge, and cloud environments, such as the IoT. Meerkat focuses on sustaining high throughput and minimizing downtime by combining partial decentralization with real-time reconfiguration of operator replicas. Meerkat is based on three key ideas: first, a unified load metric allows it to adapt fault tolerance replicas according to each node's capacity, which is critical for environments that combine low-power sensors with high-end servers. Second, it incorporates a hybrid replication strategy, enabling multiple parallel data paths while avoiding the coordination pitfalls of centralized fault tolerance. Third, it continuously updates resource awareness at runtime, quickly reassigning tasks when devices exceed their resource capacity.

Our evaluation confirms that existing approaches frequently yield data loss, excessive coordination overhead, and poor scalability when confronted with the latency variability of an IoT environment. By contrast, Meerkat's lightweight load balancing and dynamically placed fault tolerance replicas achieve zero-downtime and up to an order of magnitude higher throughput than state-of-the-art approaches. This positions Meerkat as a foundational solution for high-performance, failure-resilient streaming in heterogeneous IoT environments.

ACKNOWLEDGMENT

This work was funded by the German Federal Ministry for Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (ref. BIFOLD24B).

REFERENCES

[1] 2007. Amazon AWS Greengrass. Accessed May 2023: https://aws.amazon.com/greengrass/.

[2] 2016. Optimal operator placement for distributed stream processing applications. *DEBS 2016 - Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems*, 69–80. https://doi.org/10.1145/2933267.2933312

[3] 2017. Microsoft Azure IoT Edge. Accessed Jul 2023: https://azure.microsoft.com/en-us/services/iot-edge/.

[4] 2022. OpenCellid. Accessed Apr 2023: https://www.opencellid.org/.

[5] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. 2003. Aurora: A Data Stream Management System. 666.

[6] Micah Adler, Soumen Chakrabarti, Michael Mitzenmacher, and Lars Rasmussen. 1995. Parallel randomized load balancing. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*. 238–247.

[7] Chaudhary Ankit, Lang Felix, Ferents Danila, Schubert L. Nils, Pandey Varun, Karimov Jeyhun, Zeuch Steffen, Beedkar Kaustubh, and Markl Volker. 2025. Incremental Stream Query Deployment under Continuous Infrastructure Changes in the Cloud-Edge Continuum. *Proc. VLDB Endow.* 19, 2 (2025), 210–223. https://doi.org/10.14778/3773749.3773759

[8] Jonatha Anselmi. 2019. Combining size-based load balancing with round-robin for scalable low latency. *IEEE Transactions on Parallel and Distributed Systems* 31, 4 (2019), 886–896.

[9] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 601–613. https://doi.org/10.1145/3183713.3190664

[10] Cosmin Avasalcai, Christos Tsigkanos, and Schahram Dustdar. 2022. Resource Management for Latency-Sensitive IoT Applications With Satisfiability. *IEEE Transactions on Services Computing* 15, 5 (2022), 2982–2993. https://doi.org/10.1109/TSC.2021.3074188

[11] Quentin Baert, Anne Cécile Caron, Maxime Morge, Jean-Christophe Routier, and Kostas Stathis. 2019. A location-aware strategy for agents negotiating load-balancing. In *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 668–675.

[12] Magdalena Balazinska, Hari Balakrishnan, Samuel R. Madden, and Michael Stonebraker. 2008. Fault-Tolerance in the Borealis Distributed Stream Processing System. *ACM Trans. Database Syst.* 33, 1 (2008).

[13] Magdalena Balazinska, Hari Balakrishnan, and Michael Stonebraker. 2004. Load management and high availability in the Medusa distributed stream processing system. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (Paris, France) *(SIGMOD '04)*. Association for Computing Machinery, New York, NY, USA, 929–930. https://doi.org/10.1145/1007568.1007701

[14] Christian Berger, Philipp Eichhammer, Hans P. Reiser, Jörg Domaschka, Franz J. Hauck, and Gerhard Habiger. 2022. A Survey on Resilience in the IoT: Taxonomy, Classification, and Discussion of Resilience Mechanisms. *Comput. Surveys* 54 (9 2022). Issue 7. https://doi.org/10.1145/3462513

[15] Azer Bestavros. 1995. Demand-based document dissemination to reduce traffic and balance load in distributed information systems. In *Proceedings. Seventh IEEE Symposium on Parallel and Distributed Processing*. IEEE, 338–345.

[16] Azer Bestavros. 1997. WWW traffic reduction and load balancing through server-based caching. *IEEE concurrency* 5, 1 (1997), 56–67.

[17] Maury Bramson, Yi Lu, and Balaji Prabhakar. 2010. Randomized load balancing with general service time distributions. *ACM SIGMETRICS performance evaluation review* 38, 1 (2010), 275–286.

[18] Kristian Paul Bubendorfer. 1996. Resource based policies for load distribution. (1996).

[19] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *Proc. VLDB Endow.* 10, 12 (aug 2017), 1718–1729. https://doi.org/10.14778/3137765.3137777

[20] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38 (2015), 28–38.

[21] Paris Carbone, Jonas Traub, Asterios Katsifodimos, Seif Haridi, and Volker Markl. 2016. Cutty: Aggregate Sharing for User-Defined Windows *(CIKM '16)*. Association for Computing Machinery, New York, NY, USA, 1201–1210. https://doi.org/10.1145/2983323.2983807

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

[23] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. 2003. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Conference on Innovative Data Systems Research*. https://api.semanticscholar.org/CorpusID:339517

[24] Xenofon Chatziliadis, Eleni Tzirita Zacharatou, Alphan Eracar, Steffen Zeuch, and Volker Markl. 2024. Efficient Placement of Decomposable Aggregation Functions for Stream Processing over Large Geo-Distributed Topologies. *Proc. VLDB Endow.* 17, 6 (Feb. 2024), 1501–1514. https://doi.org/10.14778/3648160.3648186

[25] Ankit Chaudhary, Kaustubh Beedkar, Jeyhun Karimov, Felix Lang, Steffen Zeuch, and Volker Markl. 2025. Incremental Stream Query Placement in Massively Distributed and Volatile Infrastructures. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 376–390.

[26] Ankit Chaudhary, Steffen Zeuch, and Volker Markl. 2020. Governor: Operator Placement for a Unified Fog-Cloud Environment. In *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, Angela Bonifati, Yongluan Zhou, Marcos Antonio Vaz Salles, Alexander Böhm, Dan Olteanu, George H. L. Fletcher, Arijit Khan, and Bin Yang (Eds.). OpenProceedings.org, 631–634. https://doi.org/10.5441/002/EDBT.2020.81

[27] Ankit Chaudhary, Steffen Zeuch, Volker Markl, and Jeyhun Karimov. 2023. Incremental Stream Query Merging.. In *EDBT*. 604–617.

[28] Ankit Chaudhary, Ninghong Zhu, Laura Mons, Steffen Zeuch, Varun Pandey, and Volker Markl. 2025. Incremental Stream Query Merging In Action. In *Datenbanksysteme für Business, Technologie und Web (BTW 2025)*. Gesellschaft für Informatik, Bonn, 907–915. https://doi.org/10.18420/BTW2025-58

[29] Mung Chiang and Tao Zhang. 2016. Fog and IoT: An overview of research opportunities. *IEEE Internet of Things Journal* 3, 6 (2016), 854–864.

[30] Robertas Damaševičius, Nebojsa Bacanin, and Sanjay Misra. 2023. From Sensors to Safety: Internet of Emergency Services (IoES) for Emergency Response and Disaster Management. *Journal of Sensor and Actuator Networks* 12, 3 (2023). https://doi.org/10.3390/jsan12030041

[31] Hui Deng, Xiaolong Huang, Kai Zhang, Zhisheng Niu, and Masahiro Ojima. 2003. A hybrid load balance mechanism for distributed home agents in mobile IPv6. In *14th IEEE Proceedings on Personal, Indoor and Mobile Radio Communications, 2003. PIMRC 2003.*, Vol. 3. IEEE, 2842–2846.

[32] Michael Emmerich and André Deutz. 2018. A tutorial on multiobjective optimization: fundamentals and evolutionary methods. *Natural Computing* 17 (09 2018). https://doi.org/10.1007/s11047-018-9685-y

[33] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter R. Pietzuch. 2014. Making State Explicit for Imperative Big Data Processing. In *USENIX Annual Technical Conference*. https://api.semanticscholar.org/CorpusID:12956245

[34] Raul Castro Fernandez, Matthias Weidlich, Peter Pietzuch, and Avigdor Gal. 2014. Scalable stateful stream processing for smart grids. *DEBS 2014 - Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, 276–281. https://doi.org/10.1145/2611286.2611326

[35] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. 2017. Dhalion: self-regulating stream processing in heron. *Proc. VLDB Endow.* 10, 12 (Aug. 2017), 1825–1836. https://doi.org/10.14778/3137765.3137786

[36] Gianluigi Folino, Agostino Forestiero, and Giandomenico Spezzano. 2006. Swarm-Based Distributed Clustering in Peer-to-Peer Systems. In *Artificial Evolution*, El-Ghazali Talbi, Pierre Liardet, Pierre Collet, Evelyne Lutton, and Marc Schoenauer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 37–48.

[37] Sukumar Ghosh. 2014. *Distributed Systems: An Algorithmic Approach, Second Edition*.

[38] Jim Gray. 1986. Why Do Computers Stop and What Can Be Done About It?. In *Symposium on Reliability in Distributed Software and Database Systems*.

[39] Philipp M. Grulich, Breß Sebastian, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl, and Volker Markl. 2020. Grizzly: Efficient Stream Processing Through Adaptive Query Compilation. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2487–2503. https://doi.org/10.1145/3318464.3389739

[40] Divya Gupta, Shalli Rani, and Syed Hassan Ahmed Shah. 2022. *ICN-Fog Computing for IoT-Based Healthcare*. 19–37. https://doi.org/10.1002/9781119816829.ch2

[41] Moritz Hoffmann, Andrea Lattuada, and Frank McSherry. 2019. Megaphone: latency-conscious state migration for distributed streaming dataflows. *Proc. VLDB Endow.* 12, 9 (May 2019), 1002–1015. https://doi.org/10.14778/3329772.3329777

[42] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. 2005. High-availability algorithms for distributed stream processing. In *21st International Conference on Data Engineering (ICDE'05)*. 779–790.

[43] Anurag Jain and Rajneesh Kumar. 2017. Hybrid load balancing approach for cloud environment. *International Journal of Communication Networks and Distributed Systems* 18, 3-4 (2017), 264–286.

[44] Yichuan Jiang. 2015. A survey of task allocation and load balancing in distributed systems. *IEEE Transactions on Parallel and Distributed Systems* 27, 2 (2015), 585–599.

[45] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three Steps is All You Need: Fast, Accurate, Automatic Scaling Decisions for Distributed Streaming Dataflows. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*

(Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 783–798.

[46] Sukhveer Kaur, Krishan Kumar, Japinder Singh, and Navtej Singh Ghumman. 2015. Round-robin based load balancing in Software Defined Networking. In *2015 2nd international conference on computing for sustainable global development (INDIACom)*. IEEE, 2136–2139.

[47] Anastasiia Kozar, Bonaventura Del Monte, Steffen Zeuch, and Volker Markl. 2024. Fault Tolerance Placement in the Internet of Things. *Proc. ACM Manag. Data* 2, 3, Article 138 (May 2024), 29 pages. https://doi.org/10.1145/3654941

[48] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 239–250. https://doi.org/10.1145/2723372.2742788

[49] Geetika T. Lakshmanan, Ying Li, and Rob Strom. 2010. Placement of replicated tasks for distributed stream processing systems. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems* (Cambridge, United Kingdom) *(DEBS '10)*. Association for Computing Machinery, New York, NY, USA, 128–139. https://doi.org/10.1145/1827418.1827450

[50] Wei Lin, Haochuan Fan, Zhengping Qian, Junwei Xu, Sen Yang, Jingren Zhou, and Lidong Zhou. 2016. StreamScope: Continuous Reliable Distributed Processing of Big Data Streams. In *NSDI*.

[51] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. 2005. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Trans. Database Syst.* (mar 2005), 122–173.

[52] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanan Muthukrishnan, Vamsi Kuppa, Sudheer Dhulipalla, and Sriram Rao. 2018. Chi: a scalable and programmable control plane for distributed stream processing systems. *Proc. VLDB Endow.* 11, 10 (June 2018), 1303–1316. https://doi.org/10.14778/3231751.3231765

[53] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanan Muthukrishnan, Vamsi Kuppa, Sudheer Dhulipalla, and Sriram Rao. 2018. Chi: a scalable and programmable control plane for distributed stream processing systems. *Proc. VLDB Endow.* 11, 10 (June 2018), 1303–1316. https://doi.org/10.14778/3231751.3231765

[54] R. Marler and Jasbir Arora. 2004. Survey of Multi-Objective Optimization Methods for Engineering. *Structural and Multidisciplinary Optimization* 26 (04 2004), 369–395. https://doi.org/10.1007/s00158-003-0368-6

[55] A. Messac, A. Ismail-Yahaya, and C.A. Mattson. 2003. The normalized normal constraint method for generating the Pareto frontier. *Structural and Multidisciplinary Optimization* 25 (07 2003), 86–98. https://doi.org/10.1007/s00158-002-0276-1

[56] Michael Mitzenmacher. 1997. On the analysis of randomized load balancing schemes. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*. 292–301.

[57] Michael Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems* 12, 10 (2001), 1094–1104.

[58] Rashid A Mohammed et al. 2019. A survey of IoT management protocols and frameworks. *Computer Networks* 148 (2019), 132–150.

[59] Dan O'Keeffe, Theodoros Salonidis, and Peter Pietzuch. 2018. Frontier: resilient edge processing for the internet of things. *Proc. VLDB Endow.* 11, 10 (June 2018), 1178–1191. https://doi.org/10.14778/3231751.3231767

[60] Rahul Krishnan Pathinarupothi, Dipu T. Sathyapalan, Merlin Moni, K A Unnikrishna Menon, and Maneesha Vinodini Ramesh. 2021. REWOC: Remote Early Warning of Out-of-ICU Crashes in COVID Care Areas using IoT Device. In *2021 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. 2010–2013. https://doi.org/10.1109/BIBM52615.2021.9669464

[61] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. 2006. Network-Aware Operator Placement for Stream-Processing Systems. In *22nd International Conference on Data Engineering (ICDE'06)*. 49–49. https://doi.org/10.1109/ICDE.2006.105

[62] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica. 2003. Load Balancing in Structured P2P Systems. In *Peer-to-Peer Systems II*, M. Frans Kaashoek and Ion Stoica (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 68–79.

[63] Neeraj Rathore. 2018. Performance of hybrid load balancing algorithm in distributed web server system. *Wireless Personal Communications* 101 (2018), 1233–1246.

[64] Matthias J. Sax, Guozhang Wang, Matthias Weidlich, and Johann-Christoph Freytag. 2018. Streams and Tables: Two Sides of the Same Coin.

[65] Zhitao Shen, Vikram Kumaran, Michael J. Franklin, Sailesh Krishnamurthy, Amit Bhat, Madhu Kumar, Robert Lerche, and Kim Macpherson. 2015. CSA: Streaming Engine for Internet of Things. *IEEE Data Eng. Bull.* 38, 4 (2015), 39–50. http://sites.computer.org/debull/A15dec/p39.pdf

[66] Rathin Chandra Shit and Suraj Sharma. 2018. Localization for Autonomous Vehicle: Analysis of Importance of IoT Network Localization for Autonomous Vehicle Applications. In *2018 International Conference on Applied Electromagnetics, Signal Processing and Communication (AESPC)*, Vol. 1. 1–6. https://doi.org/10.1109/AESPC44649.2018.9033329

[67] Konstantinos Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–10. https://doi.org/10.1109/MSST.2010.5496972

[68] Giorgos Siachamis, Konstantinos Psarakis, Marios Fragkoulis, Arie van Deursen, Paris Carbone, and Asterios Katsifodimos. 2024. CheckMate: Evaluating Checkpointing Protocols for Streaming Dataflows. In *Proceedings of the 2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 4030–4043. https://doi.org/10.1109/ICDE60468.2024.00415

[69] Pedro F. Silvestre, Marios Fragkoulis, Diomidis Spinellis, and Asterios Katsifodimos. 2021. Clonos: Consistent Causal Recovery for Highly-Available Streaming Dataflows. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*. ACM, 1637–1650. https://doi.org/10.1145/3448016.3452834

[70] Fei Song, Khaled Zaouk, Chenghao Lyu, Arnab Sinha, Qi Fan, Yanlei Diao, and Prashant Shenoy. 2021. Spark-based Cloud Data Analytics using Multi-Objective Optimization. In *ICDE - 37th IEEE International Conference on Data Engineering*. Chania, Greece. https://hal.inria.fr/hal-02549758

[71] Li Su and Yongluan Zhou. 2015. Tolerating Correlated Failures in Massively Parallel Stream Processing Engines. *ICDE*.

[72] Peter A. Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2002. NEXMark – A Benchmark for Queries over Data Streams DRAFT. https://api.semanticscholar.org/CorpusID:18302897

[73] Weikun Wang and Giuliano Casale. 2014. Evaluating weighted round robin load balancing for cloud web services. In *2014 16th international symposium on symbolic and numeric algorithms for scientific computing*. IEEE, 393–400.

[74] Benjamin Warnke, Stefan Fischer, and Sven Groppe. 2023. Distributed SPARQL queries in collaboration with the routing protocol. In *Proceedings of the 27th International Database Engineered Applications Symposium* (Heraklion, Crete, Greece) *(IDEAS '23)*. Association for Computing Machinery, New York, NY, USA, 99–106. https://doi.org/10.1145/3589462.3589497

[75] Yong Yao and Johannes Gehrke. 2002. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.* 31, 3 (Sept. 2002), 9–18. https://doi.org/10.1145/601858.601861

[76] Boyang Yu and Jianping Pan. 2015. Location-aware associated data placement for geo-distributed data-intensive applications. In *2015 IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 603–611.

[77] Sebastian Zeuch, Malte Appelt, Martin Feldmann, Tilmann Rabl, and Volker Markl. 2020. Decentralized Stream Processing for IoT: Remember Your Edge!. In *CIDR 2020, Conference on Innovative Data Systems Research*. www.cidrdb.org. https://www.cidrdb.org/cidr2020/papers/p7-zeuch-cidr20.pdf

[78] Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp M Grulich, Sebastian Breß, Jonas Traub, and Volker Markl. 2019. The nebulastream platform: Data and application management for the internet of things. *arXiv preprint arXiv:1910.07867* (2019).