# Elastic Edge-Based Stream Processing Over Apache Storm

by

Mitra SHAHABADI

THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
IN PARTIAL FULFILLMENT FOR THE DEGREE OF
MASTER'S WITH THESIS IN SOFTWARE ENGINEERING
M.A.Sc.

MONTREAL, "DECEMBER 8, 2025"

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

**BOARD OF EXAMINERS**

THIS THESIS HAS BEEN EVALUATED

BY THE FOLLOWING BOARD OF EXAMINERS

Dr. Julien Gascon-Samson, Thesis supervisor
Department of Software Engineering and IT, École de technologie supérieure

Dr. Patrick Cardinal, Chair, Board of Examiners
Department of Software Engineering and IT, École de technologie supérieure

Dr. Marcos Dias De Assunção, Member of the Jury
Department of Software Engineering and IT, École de technologie supérieure

THIS THESIS WAS PRESENTED AND DEFENDED

IN THE PRESENCE OF A BOARD OF EXAMINERS AND THE PUBLIC

ON "NOVEMBER 25, 2025"

AT ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

## ACKNOWLEDGEMENTS

This thesis marks the end of a long and difficult journey. When I began my master's studies, I thought it would simply be about studying, doing research, and coding. I could not have been more wrong. It tested me in ways I never expected and pressured me to go far beyond what I had imagined. It challenged my emotional strength, my ability to solve problems, to adapt quickly to a new country, to work while studying, to manage with limited resources, and to think over and over again about every possible scenario. Throughout this process, I struggled with anxiety and severe depression, often feeling lost and uncertain about when this path would finally end. There were moments when I could not bear the judgment of others or find the words to answer their repeated questions about when I would graduate. No one could see the battles I was quietly fighting.

Despite all of this, I learned that life goes on, and the only person I truly have is myself. I am proud to reach this point.

I would like to express my deepest gratitude to my supervisor, Professor Julien Gascon-Samson, for his exceptional support, kindness, and understanding. He is perhaps the nicest person I have ever met. He guided me with patience and care, never making me feel pressured or stressed. Although it is hard to describe in words the gratitude I feel toward him, I am truly thankful for his support.

My heartfelt thanks go to my parents and my sister Mina, whose constant love and understanding gave me the strength to continue. Mina put me before herself at a time when I needed it the most and was the first person to push me forward when I could not keep going. I wish I could hug my parents now, after 4 years, and thank them for their endless love and support.

To my partner, Soroush, thank you for your love, patience, and unconditional care. You gave me hope when I had lost it and never let me judge myself for falling behind in life. Your presence made even the hardest days feel lighter.

I am grateful to my friends Hoda, Sajjad, Amna, Pranav, and Daniel for being there for me with kindness, laughter. Hoda and Sajjad, thank you for a long-lasting friendship that always made me feel I have true friends, even though we are far away from each other. I will never forget that phone call with you two, asking a question that I had no answer to. It ended up changing my life. Amna, thank you for greeting everyone in the lab in every language they spoke, for your sarcasm, and for your amazing ability to roast people in Persian slang. Pranav, I am still not sure whether you are a robot or a human, but you are remarkably capable, and you made me realize that I love playing badminton, right after hitting me on the forehead with the shuttlecock by accident. Daniel, you showed me that friendship without expecting anything in return still exists. Thank you for not letting us freeze during the coldest winter days.

You all made me feel that I was never alone in this journey.

Finally, I would like to express my sincere appreciation to Dr. Forogh Tahersoltani. Your wisdom and compassion opened my eyes and helped me feel alive.

**Traitement de flux élastique basé sur la périphérie avec Apache Storm**

Mitra SHAHABADI

## RÉSUMÉ

Apache Storm est un cadre de traitement de flux distribué conçu pour gérer des flux de données continus, mais son élasticité demeure limitée lorsque les composants sont déployés sur des nœuds hétérogènes et que les débits de données entrants varient au fil du temps. Les travaux existants sur Apache Storm ne traitent pas de manière adéquate la façon dont l'élasticité peut être atteinte sans intervention manuelle ni redémarrage du système. En conséquence, des goulots d'étranglement liés à l'utilisation du processeur et à la bande passante persistent souvent, augmentant ainsi la latence de bout en bout.

Cette thèse introduit une couche supplémentaire au-dessus d'Apache Storm afin d'améliorer son élasticité dans des conditions distribuées et dynamiques. La solution proposée aborde à la fois la prévention et la résolution des goulots d'étranglement liés à la bande passante entrante, à la bande passante sortante et aux ressources CPU, dans le but de minimiser la latence globale de traitement.

L'élasticité a été obtenue grâce à deux mécanismes complémentaires. Le premier, l'adaptation globale, surveille les performances du système pour décider quand et où créer de nouvelles répliques et quand supprimer en toute sécurité les répliques sous-utilisées sans nuire au débit. Le second, l'adaptation locale, est mis en œuvre au niveau des opérateurs sous la forme d'une stratégie de répartition de charge sensible aux ressources, équilibrant les données entre les répliques en aval. Ensemble, ces mécanismes permettent d'éviter les goulots d'étranglement autant que possible et de les résoudre efficacement lorsqu'ils se produisent.

Le système a été implémenté sur Docker pour un déploiement sur des nœuds hétérogènes. Les résultats expérimentaux montrent que le système peut créer et supprimer des répliques à la volée, sans interrompre l'exécution, tout en détectant et en atténuant avec succès les goulots d'étranglement liés au processeur et à la bande passante. Ces mécanismes réduisent considérablement la latence moyenne de bout en bout et améliorent l'utilisation globale des ressources par rapport aux configurations de référence. De plus, les résultats ont montré que la solution proposée peut évoluer de manière fiable dans le cadre expérimental.

Cette recherche propose une approche pratique pour renforcer l'élasticité des systèmes de traitement de flux distribués, assurant des performances robustes dans des environnements dynamiques et hétérogènes.

**Mots-clés:** Apache Storm, Élasticité, Traitement de flux distribué, Répartition de charge sensible aux ressources, Gestion des répliques, Docker

**Elastic Edge-Based Stream Processing Over Apache Storm**

Mitra SHAHABADI

## ABSTRACT

Apache Storm is a distributed stream processing framework designed to handle continuous data flows, yet its elasticity remains limited when components are deployed across heterogeneous nodes and incoming data rates vary over time. Existing work on Apache Storm does not adequately address how elasticity can be achieved without manual intervention or system restarts. As a result, performance bottlenecks in CPU utilization and bandwidth often persist, increasing end-to-end latency.

This thesis introduces an additional layer on top of Apache Storm to enhance its elasticity under distributed and dynamic conditions. The proposed solution addresses both the avoidance and resolution of bottlenecks in inbound bandwidth, outbound bandwidth, and CPU resources, aiming to minimize overall processing latency.

Elasticity was achieved through two complementary mechanisms. The first, global adaptation, monitors system performance to decide when and where to create new replicas and when to safely remove underutilized replicas without harming throughput. The second, local adaptation, is implemented at the operator level as a resource-aware load distribution strategy, balancing data across downstream replicas. Together, these mechanisms helped ensure that bottlenecks are avoided as much as possible and effectively resolved when they occur.

The system was implemented on Docker for deployment across heterogeneous nodes. Experimental results show that the system can create and remove replicas on the fly, without interrupting execution, while successfully detecting and mitigating CPU and bandwidth bottlenecks. These mechanisms significantly reduce average end-to-end latency and improve overall resource utilization compared to baseline configurations. Moreover, the findings showed that the proposed solution can scale reliably within the experimental setup.

This research provides a practical approach to enhancing elasticity in distributed stream processing systems, enabling robust performance in dynamic and heterogeneous environments.

**Keywords:** Apache Storm, Elasticity, Distributed Stream Processing, Resource-Aware Load Balancing, Replica Management, Docker

## TABLE OF CONTENTS

Page

## LIST OF TABLES

Page

## LIST OF FIGURES

## LIST OF ALGORITHMS

## LIST OF ABBREVIATIONS

| | |
|---|---|
| ETS | École de Technologie Supérieure |
| DAG | Directed Acyclic Graphs |
| SAW | Simple Additive Weighting |

# INTRODUCTION

Stream processing is a computational model designed to handle continuous data streams in real time, so that the system can react to events as they occur. This capability is increasingly important in modern applications such as the Internet of Things (IoT), industrial monitoring, and other data-intensive domains that require low-latency responses. Relying solely on centralized cloud resources for processing data streams can introduce significant delays, since raw data must travel long distances before being processed. To reduce latency and improve responsiveness, computation is being moved closer to where data is generated by using edge computing. Processing data at the edge allows systems to react quickly and meet the strict needs of real-time applications.

Apache Storm is one of the most widely used frameworks for distributed stream processing. However, its elasticity, which is defined as the ability to adapt dynamically to varying workloads and heterogeneous environments, remains limited.

In practice, the replication settings are defined before execution and remain fixed throughout the run. As a result, when workloads increase, bottlenecks emerge in terms of CPU or bandwidth because the nodes hosting operators may lack sufficient resources. On the other hand, when workloads decrease, replicas continue running unnecessarily, leading to wasted resources. Existing solutions either rely on manual intervention, which disrupts execution, or consider only limited performance metrics, making it difficult to achieve both low latency and efficient resource utilization.

The primary objective of this thesis is to design and implement a layer on top of Apache Storm that provides dynamic elasticity in distributed, heterogeneous environments. More specifically, the research aims to:

1. Develop mechanisms to prevent, detect, and address bottlenecks related to CPU, input bandwidth, and output bandwidth;

2. Propose strategies to automatically create replicas when resources are saturated, and remove replicas when workloads decrease on the fly without stopping the execution or manual intervention;
3. Design a resource-aware distribution strategy to efficiently balance load among replicas, ensuring that available resources are utilized effectively.

To achieve these objectives, this work relies on two mechanisms: global adaptation and local adaptation. Global adaptation mechanism monitors the system as a whole, deciding when new replicas should be created, where they should be deployed, and when underutilized replicas can be safely removed. Local adaptation mechanism operates within each operator, applying a resource-aware load distribution strategy to spread data among downstream replicas. Together, these mechanisms are designed to avoid and resolve bottlenecks, reduce latency, and make the system more responsive.

The key contributions of this thesis are as follows:

- Elasticity Layer Design: This work introduces an additional layer on top of Apache Storm that allows the system to automatically adjust to changing workloads in heterogeneous environments;
- Global Adaptation Strategy: A system-level strategy was introduced to monitor performance, detect bottlenecks, and dynamically add or remove replicas based on resource conditions;
- Local Adaptation Strategy: An operator-level load distribution method was proposed to route data intelligently among downstream replicas according to available resources;
- Real Deployment and Evaluation: The solution was implemented and tested in a real Docker-based deployment across heterogeneous nodes, making the evaluation as close to real-world conditions as possible;

- Evidence of Scalability: Experimental results confirmed that the approach not only reduces latency and improves resource utilization but also scales effectively as workload and system size increase.

The remainder of this thesis is organized as follows. Chapter 1 begins with the fundamental concepts of distributed stream processing, then reviews related work on elasticity in stream processing systems. The chapter concludes by identifying the research gap that motivates this study. Chapter 2 presents the system architecture, describes the algorithms for global and local adaptation, and explains all aspects related to system design. Chapter 3 provides the implementation details, including tools and technologies used. Chapter 4 introduces the experimental setup, explains the evaluation methodology, and discusses the results. Finally, Chapter 5 concludes the thesis by summarizing the contributions and suggesting directions for future research.

# CHAPTER 1

# LITERATURE REVIEW AND BACKGROUND

## 1.1 Stream Processing

In today's fast-paced digital systems, processing data quickly is not only helpful, but often necessary. For certain types of applications, particularly those that operate in real-time or rely on rapid responses, even brief delays can cause the data to lose its usefulness. If the response is too slow, the data may lose its relevance, and in certain situations, the lack of timely action can result in significant issues (Shi, Cao, Zhang, Li & Xu (2016), Abbas, Zhang, Taherkordi & Skeie (2017), Satyanarayanan (2017)). We can better understand this by looking at a few areas where fast data processing plays a critical role. For instance, self-driving cars must continuously process data from cameras, sensors, and GPS systems to make split-second decisions that ensure safety (Liu *et al.* (2019)). In medical monitoring systems, wearable devices or ICU monitors must instantly detect abnormal physiological signals to alert healthcare providers and prevent serious complications (Paganelli *et al.* (2022)). Similarly, in automated industrial environments, sensor data is used to monitor equipment health and enable real-time safety control, where any delay could lead to accidents, equipment failure, or production downtime (Sharma (2024)).

Stream processing refers to the real-time or near-real-time handling of continuously arriving data, where the system processes elements incrementally as they arrive. Stream processing systems work with potentially unbounded data streams and must operate under tight constraints of memory, latency, and correctness. These systems are designed to deliver timely and correct results even when data arrives out of order or with delays (Fragkoulis, Carbone, Kalavri & Katsifodimos (2024)).

The architecture of the stream processing system is illustrated in Figure 1.1, which depicts how continuous data streams are ingested, processed in real time, and produce various output components such as analytics, databases, and insight generation.

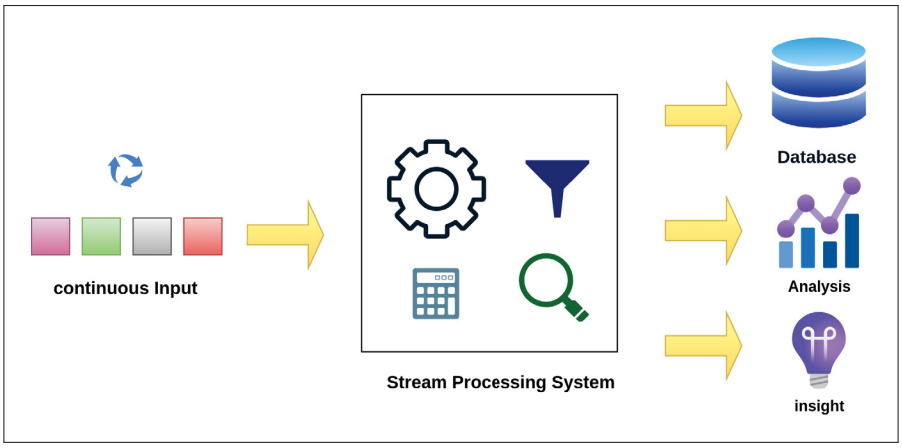



Figure 1.1 Architecture of a stream processing system

Stream processing differs from traditional batch processing in several key ways. Batch processing refers to the traditional method of processing data in large chunks collected over a period of time, such as hours or days. The system waits for all the data to arrive before performing any computation. This approach is suitable for handling large volumes of data. However, due to its high latency, batch processing is not ideal for time-sensitive applications, as insights are only available once the entire batch has been processed (Carbone, Fragkoulis, Kalavri & Katsifodimos (2020)). In contrast, stream processing handles data continuously as it arrives, enabling immediate analysis and timely responses, which support use cases such as real-time monitoring, anomaly detection, and live decision-making.

Modern stream processing systems are commonly structured around the dataflow model, where computations are represented as Directed Acyclic Graphs (DAGs). In this model, each node, also known as an operator, performs a specific task. The edges between these operators define the direction of data movement, indicating how the output of one operator becomes the input to the next (Akidau *et al.* (2015)). This design enables the system to process data in a clear, organized, and highly parallel manner.

Figure 1.2 shows the general structure of such systems, represented as a Directed Acyclic Graph (DAG) with four nodes, each corresponding to an operator. The arrows illustrate the flow of data between operators, where the output of one becomes the input of the next. For example, Op2

receives the output produced by Op1, and this same pattern continues through the subsequent operators in the sequence.

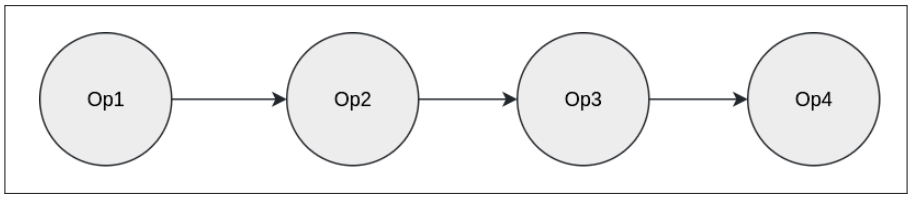



Figure 1.2 DAG representing a stream processing pipeline

Operators can be stateless, processing each incoming element independently, or stateful, maintaining information over time depending on the needs of the computation (Han *et al.* (2025)).

Most stream processing systems are not designed to run on a single machine. Instead, they typically operate in a distributed environment, where multiple interconnected computing devices, collectively referred to as a cluster, work together to achieve a common goal.

In the context of distributed stream processing, deployment refers to the process of assigning each operator in the Directed Acyclic Graph (DAG) to specific nodes within the cluster. The placement of these operators directly affects system performance, particularly with respect to latency. Therefore, it is essential to examine the underlying computing resources that support the execution of stream processing systems.

## 1.2 Stream Processing Deployment

Over the years, various deployment paradigms have emerged for stream processing systems. In this section, we focus on three of the most significant paradigms and examine their respective advantages and disadvantages.

### 1.2.1 Traditional Clusters

Traditional computing clusters were the primary infrastructure used to provide the computing resources required for distributed systems. These clusters consisted of a fixed set of locally maintained servers, which organizations used to host and manage their hardware and software internally. This setup offered a stable and controlled environment for running any system. While this method served organizations well initially, it eventually began to show major weaknesses, as it could not keep up with the rapid pace of change. These limitations encouraged businesses to seek more efficient solutions, as the traditional model was no longer adequate. The main problems with traditional computing include:

- Flexibility Limitations: In traditional computing, scaling resources such as storage or processing power requires purchasing and installing new hardware. This process can be time-consuming and costly, making it difficult to quickly adapt to sudden increases or decreases in demand (Nagy (2022));
- Resource Utilization and Cost Inefficiency: Locally managed infrastructure tends to suffer from inefficient use of resources. During periods of low activity, much of the computing capacity remains idle, while sudden increases in demand can overwhelm available resources, resulting in poor overall efficiency (Nagy (2022)). Additionally, the requirement for significant upfront spending on equipment, combined with continuous maintenance and operational costs, leads to a high total cost (Marston, Li, Bandyopadhyay, Zhang & Ghalsasi (2011), Armbrust *et al.* (2010)).

To address these challenges, newer computing paradigms have emerged, as discussed in the next part.

### 1.2.2 Cloud-Based Platforms

The increasing demand for computing solutions that are adaptable to changes has exposed the limitations of traditional centralized computing models. In response, cloud computing has emerged as a flexible and cost-effective paradigm capable of addressing these requirements.

A widely recognized definition by the National Institute of Standards and Technology (NIST) explains cloud computing as a framework that provides users with easy and on-demand access to a shared set of configurable computing resources such as servers, storage, networks, and applications, delivered over the internet. These resources can be quickly allocated or released with minimal involvement from either the user or the service provider (Cloud (2011)).

Another key advantage of cloud computing is its cost-effectiveness, as it replaces the need for large upfront capital investments with a usage-based pricing. By converting infrastructure costs into operational expenses, organizations can access computing resources on demand and pay only for what they consume. This approach not only reduces overall costs but also supports extensibility, making cloud computing well-suited for applications that demand adaptability and responsiveness (Bokhari, Makki & Tamandani (2018)).

While cloud computing offers many benefits, it also poses significant challenges for stream processing, especially in scenarios involving time-sensitive and data-intensive workloads. One major issue is latency. Since cloud servers are often located far from where the data is generated, sending information back and forth can introduce delays, making it unsuitable for stream processing. Another challenge is bandwidth usage. Constantly transferring large amounts of data to distant servers can quickly use up network capacity, leading to slowdowns and higher costs. These issues make it harder for cloud computing alone to support fast, responsive services (Alharthi, Alshamsi, Alseiari & Alwarafy (2024)), required by applications such as stream processing

In recent years, the growing complexity of intelligent systems and the rise of latency-sensitive applications have exposed the limitations of centralized computing models. As industries move toward automation, real-time analytics, and hyper-connected devices, relying solely on distant cloud servers has proven inadequate. Delays caused by long-distance data transmission, coupled with increasing network congestion and bandwidth costs, have made it clear that not all processing can happen remotely.

### 1.2.3 Edge Computing Environments

Edge computing emerged as a practical solution to the limitations of cloud computing, particularly for real-time, latency-sensitive applications that cannot afford delays. By bringing computation and data processing closer to the physical location of devices or users, edge computing helps reduce the delays caused by sending data back and forth to distant cloud servers. This is especially important in scenarios such as stream processing, where even a slight delay cannot be tolerated. In addition, edge computing helps reduce bandwidth usage by limiting how much data needs to be transmitted to the cloud. It also enhances system reliability and data privacy by keeping more data local. Moreover, operations can continue smoothly even when cloud connectivity is unstable or temporarily unavailable (Sharma, Tomar & Hazra (2024)).

In this context, an edge node refers to any computing device capable of processing data on its own. When interconnected, these nodes form a distributed cluster that can execute tasks close to the data source. Edge nodes can be anything from routers to smartphones, smart cameras, or even sensors with built-in computing power.

By handling tasks locally, edge computing helps reduce delays, saves bandwidth, and allows systems to respond more quickly. These advantages make edge computing an ideal solution for stream processing (Chiang & Zhang (2016)). However, deploying such systems in edge environments presents significant challenges. Unlike centralized cloud infrastructures that can allocate large pools of resources on demand, edge environments are inherently limited in computational power, memory, and network bandwidth. Edge nodes also tend to be highly heterogeneous, meaning they differ widely in their hardware capabilities and network conditions. This heterogeneity introduces additional complexity when designing and managing stream processing applications, especially with regard to resource allocation and system adaptation (de Assuncao, da Silva Veith & Buyya (2018)).

Now that we have covered infrastructure options for deploying stream processing systems, we turn to the inherent challenges of stream processing that persist regardless of where the pipeline runs

## 1.3 Stream Processing Challenges

While stream processing offers powerful capabilities for handling data in real time, it also brings a unique set of challenges. Stream processing adds considerable complexity to system design, as it must handle data continuously and without interruption while still producing consistent results (Stonebraker, Çetintemel & Zdonik (2005)). In addition, maintaining low-latency performance in practice can be far more difficult than in theory as workloads in real-world systems might not remain constant (Fragkoulis *et al.* (2024)). Data rates can spike suddenly due to events such as sensor bursts, a sharp rise in user activity, or unexpected traffic patterns. When such spikes occur, one or more operators in the processing pipeline may struggle to keep pace with the speed at which data arrives. This often happens when they lack sufficient computational resources, such as CPU, memory, or network bandwidth, to process incoming data in real time. This situation is known as a bottleneck, where overloaded operators cause incoming data to queue until resources become available. Even a single bottleneck in the processing pipeline can slow the stages that follow and can have a cascading effect on the entire system. The result is not only higher latency but also the potential loss of critical information.

Addressing these challenges requires resource management strategies that can adapt to changing workloads, which are reflected in the concepts of scalability and elasticity. Scalability describes a system's capacity to accommodate increasing workloads by expanding available resources (Henning & Hasselbring (2021)). This can involve vertical scaling, which means upgrading to more powerful hardware, or horizontal scaling, which involves adding more nodes or devices to the system (Grolinger, Higashino, Tiwari & Capretz (2013)).

One way to achieve horizontal scaling in stream processing is through replication, which means creating multiple identical copies of an operator and running them in parallel, with each processing a portion of the incoming data at the same time. These identical copies are referred to as replicas, and the capability of an operator to execute multiple replicas concurrently is called its parallelism. The exact number of replicas assigned is known as the parallelism degree. Increasing the parallelism degree distributes the workload across more replicas, which

can improve system performance and reduce the likelihood of any single replica becoming a performance bottleneck (Röger & Mayer (2019)).

However, elasticity focuses on the system's ability to automatically adjust its resource usage in response to workload fluctuations. An elastic system can provision additional resources when demand rises and release them when demand decreases, ideally in real-time. This dynamic adaptability helps maintain performance and resource efficiency, especially in environments where workloads are unpredictable or bursty (Borkowski, Hochreiner & Schulte (2019)).

Scalability and elasticity are critical for the stability and efficiency of stream processing systems, especially in environments where data rates can change rapidly. Without these capabilities, a system may fail to respond properly to increases in workload, which can lead to serious performance issues. Scalability helps address this by allowing the system to grow in capacity, either by strengthening existing resources or adding new ones, so it can keep up with rising demands (Borkowski *et al.* (2019)). On the other hand, elasticity ensures that resources are adjusted dynamically, scaling up when demand increases and scaling down when it drops. This is important not only for performance but also for efficient resource utilization. Without elasticity, resources might remain excessively allocated during low-load periods, resulting in waste, or insufficiently allocated during peak times, which can lead to degraded performance (Lorido-Botran, Miguel-Alonso & Lozano (2014)).

The next section evaluates major stream-processing frameworks to see how well they handle these challenges.

## 1.4 Stream Processing Frameworks

Over the past decade, several stream processing frameworks have been developed to meet the growing demand for real-time data analysis. These frameworks differ in design goals, processing models, and performance characteristics. The most well-known platforms include Apache Storm, Apache Flink, Apache Spark Streaming, Apache Samza. While these frameworks offer diverse features tailored to different use cases, they also vary significantly in terms of performance,

resource management, and adaptability (Kumar & Ismail (2022)). The following sections provide a brief overview of each framework, outlining their architecture, key concepts, and core operational model.

### 1.4.1 Apache Flink

Apache Flink is an open-source, distributed stream-processing engine designed for high-throughput, low-latency computation over both unbounded data streams and bounded datasets (Apa (2025a)). At its core, Flink turns an application into a parallel dataflow graph, where each node represents an operator and each edge represents the flow of events between them. This graph executes across a cluster of machines in a coordinated way, and Flink provides a unified programming model so the same runtime can handle real-time streams and bounded batch datasets without needing two separate systems.

Flink jobs are executed by two main components: the JobManager and one or more TaskManagers. The JobManager acts as the coordinator. It receives the program, transforms it into an optimized execution graph, schedules tasks across the cluster, and manages recovery. TaskManagers are the worker processes that actually run the operators. Each TaskManager contains a number of task slots, which represent units of computational resources. When the JobManager schedules tasks, it assigns operators' subtasks to these slots. This design gives Flink a clear separation between coordination and execution, enabling efficient parallel processing.

A fundamental characteristic of Flink is its emphasis on stateful stream processing. Unlike simple streaming engines that treat each incoming event independently, Flink provides operators with the ability to maintain state that evolves as new records arrive.

To ensure fault tolerance, Flink uses a checkpointing mechanism based on asynchronous snapshotting. Periodically, Flink injects checkpoint barriers into the streams; these barriers flow alongside the data and mark consistent points in the input streams. When all operators have observed a given barrier, the system captures the state of the entire job, allowing it to recover from failures without losing progress or producing inconsistent results.

Flink does allow applications to change their level of parallelism at runtime, but it does this in a controlled way rather than through instant, on-the-fly scaling. The basic idea is that Flink can "pause" a running job, capture its entire state, and then restart it with a different number of operator instances. This process is called rescaling. Because Flink stores operator state in a structured, partitioned form, it knows how to split or merge that state when increasing or decreasing parallelism, so the job continues as if nothing happened.

When a job is rescaled, Flink takes a savepoint, which is essentially a complete snapshot of every operator's internal state and the exact position in each input stream. The job is then stopped, the state is redistributed across the new set of parallel tasks, and the job resumes execution from the moment the snapshot was taken. This means that scaling up or down is safe and deterministic, but it does introduce a brief interruption because the job needs to be restarted. For many applications, this pause lasts only a few seconds and is acceptable, but it's important to note that it isn't perfectly seamless.

More recently, Flink introduced a reactive mode, which makes elasticity feel more automatic. In this mode, Flink monitors the available TaskManager resources. When new TaskManagers join the cluster, Flink increases parallelism to take advantage of them; when resources shrink, it reduces parallelism accordingly. Even in reactive mode, though, reconfiguration still goes through the savepoint-and-restart process.

In practical use, this form of elasticity is quite helpful. Applications can scale up during heavy workload periods, scale down to save resources when traffic drops, or adjust to changing cluster capacity without rewriting the application. The limitation is that Flink's elasticity is more similar to dynamic reconfiguration with a short interruption rather than continuous, zero-downtime scaling. It provides strong adaptability, but it's built around controlled restarts and state redistribution, not instantaneous operator replication.

### 1.4.2 Apache Samza

Apache Samza is a distributed stream-processing framework originally developed at LinkedIn and later open-sourced through the Apache Foundation (Apa (2025b)). Like other modern streaming engines, Samza is designed to process data continuously as it arrives, but what makes Samza distinct is how tightly it integrates with Apache Kafka, which acts as a high-throughput, durable event-log and messaging system. Kafka provides partitioned input streams, guarantees ordering at the partition level, and stores data long enough for Samza tasks to read at their own pace.

Samza delegates cluster management to Apache YARN or Kubernetes. Instead of building its own resource manager, Samza focuses on the execution and state aspects of streaming while relying on an external system for scheduling and resource allocation.

Samza's architecture is built around the idea of Samza jobs, which are executed as containers managed by YARN or Kubernetes. Each job is composed of tasks, and each task processes a partition of the input stream. Tasks run inside Samza containers, and the resource manager is responsible for placing those containers on nodes across the cluster.

A key feature of Samza is its built-in support for local state. Tasks can maintain their own state using embedded key–value stores, often RocksDB, which are stored on local disk. To remain fault-tolerant, Samza checkpoints state to durable storage and uses Kafka changelogs to track updates. If a task fails or is moved to another machine, the system can rebuild its state by replaying the changelog. This combination of local RocksDB storage with Kafka-backed changelogs gives Samza both speed and reliability.

After describing the core architecture, we can look at how Samza approaches elasticity. Samza's elasticity model is closely tied to the underlying resource manager (YARN or Kubernetes). To scale a job, Samza increases or decreases the number of containers, which effectively changes the number of tasks and therefore the level of parallelism. Because Samza maps tasks directly to Kafka partitions, scaling out typically requires increasing Kafka partitions ahead of time. When

scaling occurs, Samza redistributes partitions across the available containers, and tasks begin processing their new partitions from the last committed offset.

Samza does not support seamless, on-the-fly operator replication. Elasticity in Samza is more of a controlled redeployment: the job is updated with a new container count, resources are allocated by YARN or Kubernetes, and tasks are redistributed. State recovery relies on replaying changelogs from Kafka, which ensures correctness but introduces some restart overhead. In practice, this approach allows Samza to scale up during high traffic and scale down during quieter periods, but it is not designed for immediate adjustments to rapid fluctuations in resource availability or bottleneck conditions.

### 1.4.3 Apache Storm

Among these, Apache Storm stands out as one of the earliest and most widely adopted frameworks for low-latency stream processing. Its simplicity and modular architecture make it a compelling choice for a range of real-time analytics tasks, which is why this work focuses specifically on enhancing Storm's capabilities.

Apache Storm is a powerful framework designed for processing continuous streams of data in real-time. It provides a robust platform for building stream processing applications that can handle large volumes of data with low latency. To better understand how Storm achieves this, the following sections first introduce its core concepts and then describe its architecture and execution model.

#### 1.4.3.1 Apache Storm Core Concepts

This subsection details the essential building blocks of Apache Storm and their role in enabling real-time computation:

- **Tuple:** A tuple is the basic unit of data in Apache Storm. It's an ordered list of values, similar to a row in a database, that moves through the system (Toshniwal *et al.* (2014)). As it flows through the system, different components process and transform it;

- **Spout:** A spout in Apache Storm is the component that injects data into the system. It connects to external sources such as message queues, databases, or APIs, and emits this data as tuples to be processed in real time (Requeno *et al.* (2019));
- **Bolt:** In Apache Storm, a bolt is the core processing unit that receives streams of tuples, performs computations or transformations on them, and emits new tuples as output. This is where most of the data processing and business logic takes place within a Storm application (Requeno *et al.* (2019));
- **Sink:** In the context of Apache Storm, a sink refers to the final component in a stream processing topology that receives processed data and writes it to an external system, such as a file system, database, or another persistent storage. The sink is the endpoint of the data flow within the system and does not emit further tuples;
- **Topology:** Topology refers to a network of spouts and bolts connected together. It follows a Directed Acyclic Graph (DAG) structure, where the links between components indicate how data moves from one processing element to the next (Cao, Wu, Bao, Hou & Shen (2020)). The following figures illustrate examples of linear and non-linear topologies, highlighting different ways in which spouts and bolts can be connected to form data processing workflows in Apache Storm.

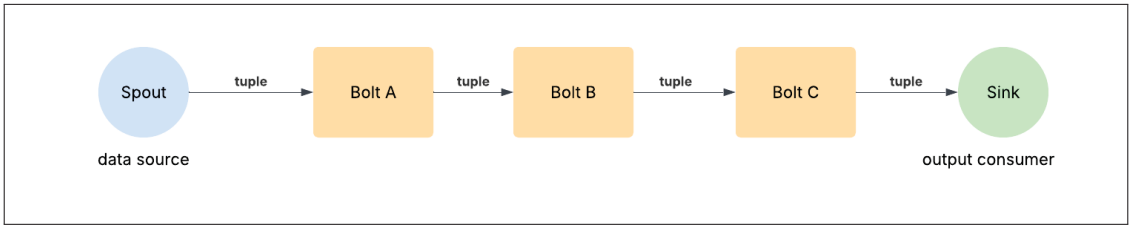



Figure 1.3 Linear topology example in Apache Storm

As shown in Figure 1.3, the linear topology consists of five distinct operators arranged in a sequential processing chain. It begins with a spout that handles data ingestion, followed by three bolts, which are responsible for executing the core processing logic, and concludes with a sink that stores or forwards the final output. In this configuration, tuples flow through the topology in a fixed order, with each bolt executing its operation before passing the processed

data to the next component. In contrast, Figure 1.4 illustrates a non-linear topology that supports more complex data flow patterns;

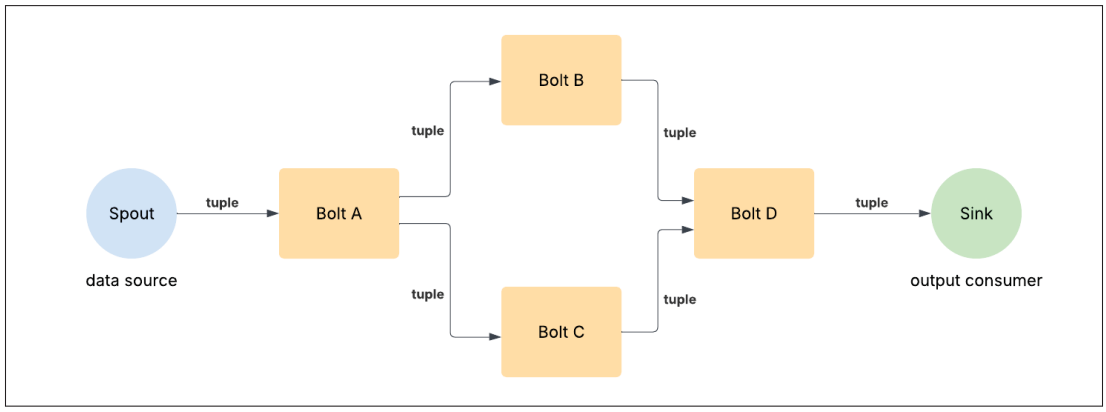



Figure 1.4 Non-Linear topology example in Apache Storm

- **Component:** In Storm, a component is either a spout or a bolt, or a sink. A topology is made up of multiple components connected together. As an example, in Figures 1.3 and 1.4, the topologies consist of 5 and 6 components, respectively.
- **Upstream Component:** In Apache Storm, a component is considered upstream of another if it sends tuples to that component. The term refers to the position of a component earlier in the data flow within the topology's DAG. For example, in Figure 1.4, Bolt A is the upstream component of Bolt B and Bolt C because it sends tuples to both. Similarly, Bolt B and Bolt C are upstream components of Bolt D.
- **Downstream Component:** A component is considered downstream of another if it receives tuples from that component, which means it appears later in the data flow path. In Figure 1.4, Bolt B and Bolt C are downstream of Bolt A, while Bolt D is downstream of both Bolt B and Bolt C. The Sink is downstream of Bolt D, as it receives tuples from it.
- **Task:** A task is one running copy of a spout or bolt. It does the actual data processing, such as reading data or transforming it (Toshniwal *et al.* (2014)).
- **Executor:** An executor is a thread responsible for running one or more tasks of the same type. This means all tasks within the same executor perform the same logic, but on different data (Toshniwal *et al.* (2014)).

- **Worker:** A worker is a separate Java Virtual Machine (JVM) process that runs a set of executors for the topology. Tasks run inside executors, executors run inside workers, and workers run on the machines in the Storm cluster (Toshniwal *et al.* (2014)). Figure 1.5 illustrates the relationship between the host machine, workers, executors, and tasks in Apache Storm.

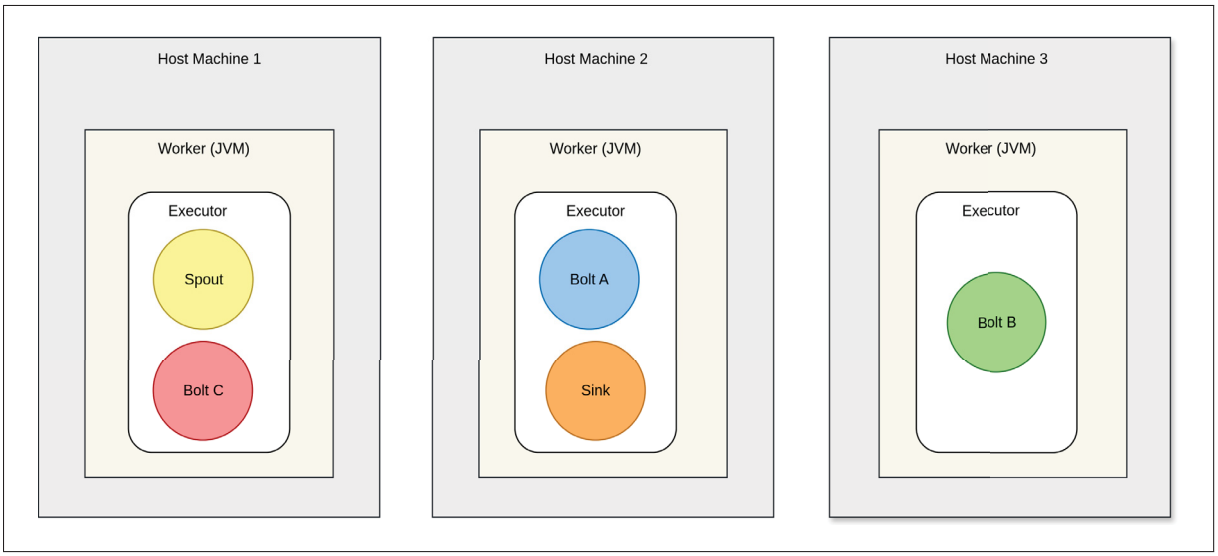



Figure 1.5 Relationship between workers, executors, and tasks in Apache Storm

- **Scheduling:** In Apache Storm, scheduling refers to the process of assigning tasks to available worker processes across the cluster. Each task is executed within an executor, which runs inside a worker slot (a JVM process) on a specific cluster node. The scheduler determines this mapping, which directly influences how computational resources are utilized and how balanced the workload is across the system.

  By default, Storm uses a round-robin scheduling strategy that evenly distributes executors among all available worker slots (Peng, Hosseini, Hong, Farivar & Campbell (2015)). This approach helps prevent overloading a single node. Scheduling decisions occur when a topology is initially launched and whenever a rebalance operation is triggered. Storm also lets users create custom scheduling policies that consider factors such as CPU load, memory usage, and network bandwidth.

#### 1.4.3.2 Apache Storm Architecture

Apache Storm is designed for distributed execution, running streaming topologies across a cluster. Among the ways to organize computation in clusters, Storm adopts a master–worker design, in which the master node is responsible for overall coordination, scheduling, and resource management, while the worker nodes are responsible for executing the actual processing (Zhao *et al.* (2025)).

To support this distributed setup, the Storm architecture relies on the following key components, which form the backbone of a Storm cluster:

- **Nimbus:** Nimbus serves as the master node in Apache Storm and acts as the central coordinator for the entire cluster. It is responsible for distributing application code, assigning tasks to worker nodes, and monitoring the overall health of running topologies. Nimbus also handles scheduling and resource allocation, deciding which worker should execute each component of a topology (Jayalakshmi (2021)). It maintains the cluster state and communicates with the nodes responsible for executing tasks to ensure that tasks are correctly executed. If Nimbus fails, the cluster continues to run existing topologies, but no new topologies can be submitted until Nimbus is restored.
- **Supervisor:** Each worker node hosts a Supervisor process that applies Nimbus's scheduling decisions on the local machine. It accepts task assignments, starts the required processes, monitors their health, and restarts them on failure. It also reports local status back to the cluster. In short, the Supervisor is the per-node manager that ensures assigned topology components run correctly on that machine (Toshniwal *et al.* (2014)).
- **Zookeeper:** Storm relies on ZooKeeper as its centralized coordination and configuration layer. It stores the critical information about the state of the cluster, such as task assignments and worker heartbeats, and keeps Nimbus and Supervisors in sync. Nimbus publishes assignments and cluster state to ZooKeeper, then supervisors read their instructions and write back status updates. Because this state lives in ZooKeeper, the system can recover from failures. Even if Nimbus goes down, a new instance can read the saved state and continue (Toshniwal *et al.* (2014); Hunt, Konar, Junqueira & Reed (2010)).

Figure 1.6 presents a high-level overview of the Apache Storm architecture.

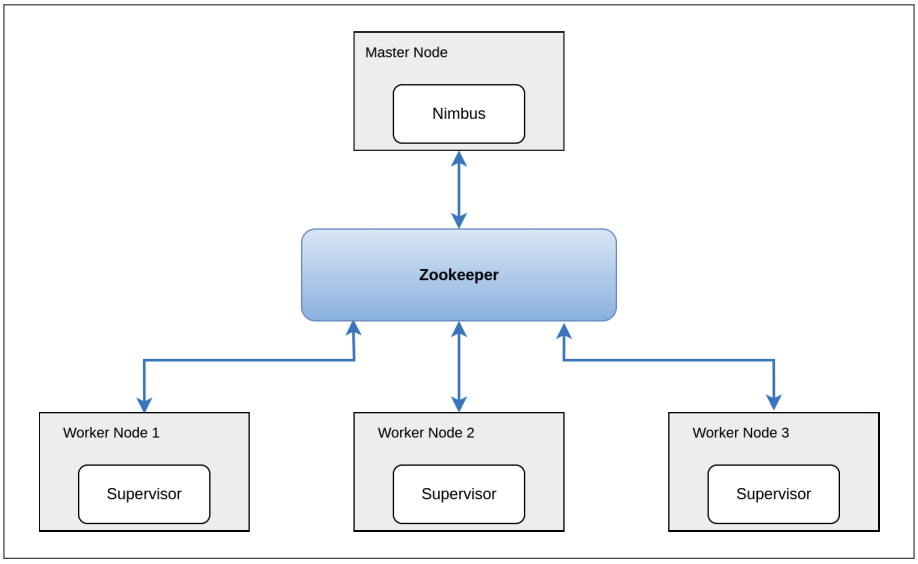



Figure 1.6 Apache Storm Architecture

After understanding the basic structure and operational flow of Apache Storm, it is essential to examine the mechanisms it employs to achieve scalability and elasticity.

#### 1.4.3.3 Scalability and Elasticity in Apache Storm

Storm achieves scalability primarily through its distributed architecture, which enables horizontal scaling by distributing computation across multiple worker nodes in a cluster. Scalability is further supported by multi-level parallelism controls that allow fine-grained tuning at several levels. At the highest level, the number of workers per node determines how many JVM processes run on each machine, with each worker having its own configurable resources, such as memory allocation. Within each worker, the number of executors per worker, which is a configurable parameter, specifies how many threads are dedicated to executing tasks. Finally, the number of tasks per executor defines how many task instances each thread will manage, with each task being an identical copy of the same code that processes different portions of the incoming data (Toshniwal *et al.* (2014)). While the number of workers and executors can be adjusted at runtime using manual commands, the number of tasks is set before topology deployment and

remains fixed for the lifetime of that topology (Nasiri, Nasehi, Divband & Goudarzi (2023)). By adjusting these parameters, Storm can manage resource usage to match workload and hardware capacity.

Another key aspect of scalability in Apache Storm lies in its stream grouping strategies, which define how data tuples are routed from one component to another. The choice of grouping directly affects load distribution and overall processing efficiency. Storm offers several built-in grouping options, each designed for specific requirements. The descriptions below outline how each grouping works and when it is typically applied:

- **Shuffle Grouping:** Shuffle grouping distributes tuples to downstream tasks in a round-robin fashion (Apache Software Foundation (2025)). This ensures that the workload is balanced across all tasks, preventing any single task from being overloaded when data arrives at varying rates. It is particularly suitable for stateless processing, where each tuple can be handled independently without requiring information from earlier tuples.
- **Fields Grouping:** Fields grouping ensures that tuples sharing the same value in a specified field (or set of fields) are always routed to the same downstream task (Apache Software Foundation (2025)). This guarantees that related data is consistently processed by the same task, enabling stateful operations, where computations depend on maintaining and updating information over time. Typical applications include counting events per user, aggregating sales by store, and tracking sensor measurements by device ID.
- **All Grouping:** All grouping broadcasts every tuple to all downstream tasks, ensuring that each task receives the same data simultaneously (Apache Software Foundation (2025)). This approach is particularly useful for distributing control information or shared updates across the topology.
- **Global Grouping:** Global grouping routes all tuples from the source component to one specific downstream task. This means that instead of distributing the load across multiple tasks, all processing is directed to a single execution point (Apache Software Foundation (2025)). It is typically used in situations where all partial results from upstream tasks must be

combined in one place before producing the final output. This centralization ensures that the final aggregation or decision-making step has complete visibility of the entire data stream.

- **Direct Grouping:** Direct grouping is a special routing strategy in Apache Storm where the sending task explicitly determines which specific downstream task will receive each tuple. Unlike other strategies, where Storm's built-in routing logic decides the destination, direct grouping grants complete control to the upstream task over tuple delivery (Apache Software Foundation (2025)). This fine-grained control enables customized load balancing or application-specific routing rules that cannot be achieved through predefined grouping strategies. Common applications include implementing priority-based processing or dynamically adjusting routing decisions based on performance metrics.

While scalability ensures that Storm can handle increasing workloads by adding resources, elasticity determines how effectively it can adjust those resources when workloads fluctuate. This adaptability is crucial, as it can prevent both bottlenecks and waste.

In Apache Storm, once a topology is submitted, the number of tasks remains fixed throughout its execution. These tasks are created according to the initial parallelism settings, and the topology continues to operate with the same number of replicas unless it is stopped and redeployed with new settings. This static allocation prevents the system from adjusting the number of tasks dynamically to respond to workload changes. Stopping and redeploying a topology is interfering because it interrupts processing and can result in delays or temporary data loss, which is undesirable for real-time stream processing.

To address this limitation, Apache Storm includes a built-in mechanism that allows parallelism settings to be modified while the topology is still running. This technique is triggered through a specific system command, enabling changes to parameters such as the number of workers and executors without stopping execution. Once the command is executed, Storm redistributes tasks across the available workers, assigning them to newly allocated resources when scaling up or to existing resources when scaling down (Puttaswamy (2018)).

Storm also includes buffer management through internal send and receive queues between components. Every time a spout or bolt emits a tuple, it is placed in an output queue, which serves as temporary storage before the tuple is delivered to the receiving component's input queue. These queues act as shock absorbers for the data stream. When the incoming rate briefly exceeds the processing rate of the downstream component, tuples accumulate in the input queue instead of being dropped immediately. This buffering allows the system to continue processing without data loss during short-lived bursts in workload. Once the downstream component catches up, the queued tuples are processed in order, smoothing out workload variations and preventing sudden drops in performance.

Another way Storm provides elasticity is through its backpressure mechanism. Backpressure is a built-in flow control feature that ensures downstream bolts are not overwhelmed by high incoming data rates. When a bolt's receive queue becomes full, it signals upstream components to slow down tuple emission. This signal propagates through the processing chain until it reaches the spouts, which temporarily reduce or pause tuple generation. When sufficient capacity becomes available again, the slowdown request is withdrawn, and normal data flow resumes. This self-regulation mechanism prevents congestion, avoids potential data loss, and allows the system to maintain stability.

Despite these capabilities, Storm's flexibility to adapt at runtime is limited. For instance, automatic addition or removal of cluster nodes is not supported. Instead, any adjustments during execution are restricted to redistributing existing tasks among workers and executors within the resources that have already been allocated to the cluster. Furthermore, the number of tasks per component remains fixed during a run, even after rebalancing. As a result, this static allocation can lead to inefficiencies. During periods of low demand, resources may remain underutilized, whereas in high-load intervals, the fixed task count may cause a bottleneck. In addition, there are no built-in workload prediction mechanisms or automatic scaling triggers, which means Storm requires external tools to achieve true elasticity.

## 1.5 Related Work

Over time, a wide range of approaches have been used to bring elasticity to stream processing systems. These approaches differ based on the goals they aim to achieve, such as reducing latency or improving resource usage, and the stream processing frameworks they rely on, each offering its own features and trade-offs. Several studies have explored these ideas in practice, focusing on different environments. We start by looking at the work done on stream processing systems in the cloud.

### 1.5.1 Elastic Stream Processing in Cloud

Among the various elasticity strategies developed for cloud stream processing systems, reactive approaches have been the most widely adopted. Reactive elasticity refers to scaling actions that occur in direct response to current system conditions, rather than based on predictions of future demand. In this approach, the system continuously monitors performance metrics such as CPU usage, memory consumption, or tuple arrival rate. When any of these metrics exceed predefined thresholds, additional resources are allocated; when they drop below the thresholds, resources can be reduced. This method is widely used in stream processing frameworks because it is simple to implement and can quickly resolve performance bottlenecks as they occur (Röger & Mayer (2019)).

Several systems have implemented reactive elasticity principles to address specific performance challenges in Apache Storm. Sgp-Stream is developed on top of Apache Storm to enhance the performance of distributed stream processing in cloud environments by managing scheduling, grouping, and parallelism. Sgp-Stream handles these three aspects in a coordinated way to avoid the performance problems that can happen when they are optimized separately. It employs adaptive techniques such as runtime-aware stream grouping, elastic scheduling, and optimization-based scaling to handle fluctuating workloads. Experiments conducted on Alibaba Cloud demonstrate that Sgp-Stream delivers lower latency, higher throughput, and improved resource utilization compared to existing solutions (Sun, Chen, Gao & Buyya (2024)).

FUGU is another system built around reactive elasticity. (Heinze, Jerzak, Hackenbroich & Fetzer (2014)). It is a cloud-based elastic scaling system designed for distributed stream processing, with a strong focus on reducing latency violations during scaling actions. The system follows a reactive strategy: it continuously monitors CPU utilization and initiates scale-out or scale-in decisions when predefined thresholds are crossed. A central contribution of FUGU is its latency cost estimation model, which predicts the temporary increase in end-to-end latency that occurs when operators are moved across hosts. This allows the system to categorize scaling actions as either mandatory, when they are required to avoid overload, or optional, when they aim to optimize cost. Optional actions are postponed whenever the predicted latency increase risks violating service-level agreements.

FUGU provides elasticity at two layers. First, it supports VM-level horizontal scaling by adding or removing hosts to adapt to workload fluctuations in pay-per-use cloud environments. Second, it offers operator-level placement elasticity by migrating operators between existing hosts for better load distribution. However, the system does not support parallelizing an operator into multiple processing instances. Each operator is a single, non-replicated entity that can only be relocated, which limits the granularity of elasticity compared to frameworks such as Apache Storm that allow dynamic scaling at the executor or task level. Operator placement is managed centrally using a FirstFit bin-packing heuristic that considers CPU load, available network bandwidth, memory, and the operator's state size. Movement cost is estimated using several factors, including state-transfer time, queue accumulation during operator pauses, and the number of upstream or downstream queries affected in the processing graph.

While reactive mechanisms offer simplicity and responsiveness, they do not anticipate future load fluctuations. As a result, they might be prone to delayed responses to sudden workload spikes, which later leads to temporary performance degradation. This limitation has motivated the development of predictive elasticity mechanisms. These mechanisms aim to forecast future workload patterns and scale resources proactively before performance issues occur (Hanif, Lee & Helal (2020)). For example, PASCAL presents a proactive auto-scaling architecture for distributed services in cloud environments that predicts future workload patterns using

machine learning, rather than reacting to system changes after they occur. Its key innovation lies in its ability to estimate system performance even when the workload is unevenly distributed across nodes, and to schedule scaling actions in advance by considering the time required to add or remove resources. The authors implemented PASCAL for both Apache Storm (stream processing) and Apache Cassandra (distributed datastore), showing that proactive scaling can lead to significant resource savings while maintaining performance comparable to over-provisioned systems and clearly outperforming reactive approaches (Lombardi *et al.* (2019)).

Predictive methods work well in many cases, but they can have trouble when workloads change suddenly in ways that were not expected. To address this, many systems integrate predictive and reactive techniques. PA-SPS introduces an adaptive and predictive solution for achieving elasticity in cloud-based stream processing systems. It is implemented as an extension of Apache Storm, with modifications to the original framework that enable predictive auto-scaling, including a pool of pre-allocated replicas and a load-aware grouping mechanism. Instead of reacting to current load, the system anticipates changes by forecasting future input rates and dynamically adjusts the number of operator replicas accordingly. Its key features include the ability to activate or deactivate replicas at runtime without downtime, and intelligent event distribution based on replica utilization. PA-SPS uses machine learning to predict incoming workload and follows a control loop that monitors the system, analyzes the situation, makes decisions, and applies changes automatically. Experiments conducted on Google Cloud Platform show that the system reduces resource usage effectively while matching the performance of over-provisioned setups and outperforming other adaptive stream processing systems (Wladdimiro, Arantes, Sens & Hidalgo (2024)).

Another contribution worth noting is Es-Stream, which is a cloud-based elastic scaling framework built on Apache Storm that addresses the challenge of processing fluctuating data streams with stateful operators in distributed computing clusters. The system extends Storm's architecture by introducing four key modules called Monitor, Topology Analysis, Resource Analysis, and State Notification. It dynamically scales along two dimensions, which are operator parallelism and resource allocation, combining both proactive workload forecasting and reactive adjustments

to balance latency and resource usage. When stateful operators need to scale, Es-Stream employs a low-overhead state management mechanism in which upstream operators back up downstream states and cache data tuples at dynamic intervals, avoiding costly full system state resets that are required in traditional Storm deployments. The framework also analyzes workload patterns to make intelligent scaling decisions across the computing cluster. Experimental results show that Es-Stream significantly outperforms existing Storm-based solutions in both latency reduction and recovery time while effectively handling variable workloads across distributed cloud infrastructure (Wu, Sun, Gao, Li & Buyya (2024)).

In the same direction, a recent study presents a platform-independent framework for adaptive load balancing in distributed stream processing systems, compatible with tools such as Apache Storm, Apache Flink, and Apache Spark Streaming. It integrates AI-based workload prediction with dynamic resource management, combining techniques such as moving operators between nodes and feedback-driven scaling to reallocate resources before bottlenecks occur. The framework works in both cloud deployments and serverless environments, where the cloud provider automatically manages and scales computing resources based on demand. It also applies advanced machine learning methods to detect workload patterns that traditional techniques miss. Experiments on multiple cloud platforms show improved processing speed, better resource utilization, and reduced costs compared to conventional reactive approaches (Mukkath (2025)).

### 1.5.2 Elasticity in Edge-based Stream Processing

Due to the inherent geographical distance between centralized cloud data centers and data sources, cloud computing alone may not always be the best choice for stream processing systems where latency is critical. Transferring large volumes of streaming data to the cloud for processing can introduce network delays and bandwidth bottlenecks, compromising the latency requirements of real-time systems. To overcome these challenges, extensive research has examined the integration of edge computing into stream processing architectures. One example of such work is Amnis, a stream processing framework built on Apache Storm for edge computing environments that focuses on static, deployment-time optimization rather than runtime elasticity.

Its optimization process centers on determining the most efficient initial placement of operators across the cluster, aiming to reduce data movement, balance workload, and improve scheduling efficiency. While this approach delivers strong performance improvements, it functions primarily as an advanced initial placement optimizer. Once the configuration is deployed, it remains fixed during execution, and the system does not adapt dynamically to workload changes. This limitation makes Amnis well suited for stable environments but less effective in scenarios where workloads fluctuate significantly (Xu, Palanisamy, Wang, Ludwig & Gopisetty (2022)).

A related effort proposes a heterogeneity-aware scheduling algorithm to address the poor performance of Storm's default Round-Robin scheduler by combining a CPU utilization prediction model, built from profiling data, with a progressive scheduling approach. The method determines the optimal number of task instances for each topology component and assigns them to machines based on their processing capabilities. While this leads to better resource utilization and higher throughput, the number of task instances is fixed once execution begins. Like Amnis, this means any change requires stopping and redeploying the topology, preventing true runtime elasticity (Nasiri *et al.* (2023)).

While these deployment-time approaches improve efficiency, other works attempt to handle runtime behavior directly. EdgeWise is a lightweight stream processing engine designed for the realities of resource-constrained edge devices (Fu, Ghaffar, Davis & Lee (2019)). Its core idea is to keep the system responsive by detecting congestion early and prioritizing operations whose queues begin to build up. Instead of relying on large thread pools or heavy profiling, it uses a small, fixed worker pool and a dynamic, queue-based scheduler that automatically adjusts to workload changes.

Evaluated on Raspberry Pi devices, EdgeWise delivers much higher throughput while maintaining low latency, showing that efficient scheduling alone can meaningfully improve performance on limited hardware. Still, its adaptability is mostly local to a single node. The system focuses primarily on CPU behavior and does not fully incorporate network conditions, bandwidth

constraints, or intelligent cross-node placement, limiting its applicability in larger heterogeneous edge deployments.

Another system that targets edge-based stream processing is VideoPipe, which provides a system for building video stream processing pipelines across heterogeneous edge devices in home environments. It follows a modular architecture, using lightweight modules for application logic and stateless services for heavy computational tasks. To reduce latency, modules are placed on the same device as the services they require, and the system employs a queue-less design with backpressure to avoid intermediate buffering delays. VideoPipe can detect bottlenecks and demonstrates service reusability for horizontal scaling. However, scaling is entirely manual, and there are no automated mechanisms for adapting to workload changes, limiting its applicability for fully elastic deployments (Salehe, Hu, Mortazavi, Mohomed & Capes (2019)).

Cumulus is more focused on runtime elasticity within Apache Storm by automatically adding or removing virtual machines from the cluster based on real-time performance metrics. Using a Monitor–Analyze–Plan–Execute (MAPE) control loop, it tracks CPU usage, tuple processing rates, and queue sizes, scaling up when bottlenecks are detected and scaling down when resources are underutilized. While effective, Cumulus assumes homogeneous nodes and does not incorporate network bandwidth or latency into its scaling decisions. As a result, it may fail to optimize performance in heterogeneous edge environments (Van Der Veen, Van Der Waaij, Lazovik, Wijbrandi & Meijer (2015)).

Another line of work explores ways to coordinate multiple edge devices in a simple, decentralized manner. DART offers a more adaptive architecture for edge environments through a decentralized design that avoids centralized bottlenecks by using a peer-to-peer overlay network. Built from scratch on top of Apache Flume and Pastry, DART supports automatic operator placement and chaining, meaning operators are linked to run together in the same process to avoid extra communication delays, as well as scaling across edge nodes. While DART can dynamically adjust workloads and achieve large-scale deployments, its model may not integrate effectively with Apache Storm, where the number of tasks is fixed unless the topology is redeployed. This

constraint limits the direct applicability of DART's methods to systems with Storm's execution model (Liu, Da Silva & Hu (2021)).

Another example of such a solution is ECStream Processing, which introduces a decentralized edge computing architecture that distributes stream processing tasks across clusters of heterogeneous edge devices (Dautov & Distefano (2020)). Its goal is to keep computation close to the data source so time-critical IoT applications can respond quickly, without relying on the cloud. The system builds on an extended version of Apache NiFi, using containerization, ZooKeeper coordination, and overlay networking to enable dynamic node discovery and seamless cluster reconfiguration. Device profiles, expressed in JSON, capture hardware and network characteristics so the system can adapt to heterogeneity and select appropriate nodes at runtime.

To balance the workload, the architecture uses resource-aware selection policies that consider CPU capacity, bandwidth, and latency before assigning tasks. A combination of greedy placement, custom job prioritization, and pipeline-level parallelism ensures that weaker devices are not overloaded and throughput remains consistent. The design demonstrates significant performance gains on a real-time license plate recognition pipeline deployed on devices like Raspberry Pi boards and smartphones. Although the approach is most suitable for relatively small clusters due to coordination overhead, its ideas could inspire adaptations in other stream processing platforms, even though substantial changes would be required to translate them into Storm's operator-centric execution model.

Among modern approaches, some systems apply machine learning and reinforcement learning to automate elasticity decisions. For instance, Dias de Assunção presents a forward-looking vision for elastic and sustainable distributed stream processing on heterogeneous edge infrastructures (Dias de Assuncao (2021)). The study addresses the NP-hard challenge of operator placement by integrating queueing-theory performance modeling with deep reinforcement learning techniques. A key contribution is its explicit focus on heterogeneity, covering a wide spectrum of resources from micro data centers to constrained devices like Raspberry Pis. The framework considers multiple objectives, including end-to-end latency, cost, energy efficiency, and resource utilization,

and it supports runtime elasticity through dynamic scale-in and scale-out decisions guided by RL-based reconfiguration policies that also account for network constraints and inter-node latency.

By using a custom simulator, the authors explore how such techniques could support intelligent placement, dynamic reconfiguration, and fault tolerance in realistic edge environments. While the paper positions itself as part of an ongoing research agenda rather than a fully deployed system, it provides a solid conceptual foundation and identifies clear pathways for advancing DSP elasticity on real heterogeneous edge platforms.

Within this family of edge elasticity approaches, MBLinUCB offers an adaptive elasticity mechanism that uses reinforcement learning to automatically adjust the parallelism of streaming operators at runtime on heterogeneous edge environments (Xu & Palanisamy (2021)). Instead of relying on manual tuning, it learns how to scale operators up or down based on observed performance and improves its learning efficiency by using a simple queuing model for pre-training. The system is integrated with Apache Storm and shows fast convergence and strong results in both simulated and real testbed experiments.

Despite its effectiveness, MBLinUCB has two notable limitations for edge environments. It does not monitor real resource conditions such as CPU load or network bandwidth, relying only on parallelism as a proxy for resource usage. It also assumes that operator placement remains fixed and provides no mechanism for dynamic migration or load balancing across nodes. Approaches that incorporate real resource metrics and support adaptive placement decisions offer a more complete solution for managing elasticity in heterogeneous and resource-constrained edge settings.

### 1.5.3 Elastic Stream Processing in Cloud-Edge Environment

The third group of studies combines the strengths of both cloud and edge environments, aiming to coordinate elasticity across a geographically dispersed hierarchy where powerful cloud nodes and resource-limited edge devices work together. EDRP proposes an optimization-driven way to

decide where operators should run and when they should be replicated in large, geographically distributed stream processing systems (Cardellini, Lo Presti, Nardelli & Russo Russo (2018)). It treats placement and scaling as a single coordinated problem and uses a control loop to continuously reassess the system and apply changes such as migrations or scale-out/scale-in actions. The framework is built with cloud–edge deployments in mind and explicitly accounts for real-world heterogeneity, including differences in CPU power, bandwidth between sites, and the latency of wide-area networks. A prototype integrated into Apache Storm demonstrates the approach in both simulations and smaller real deployments.

One of the central ideas in EDRP is to recognize that reconfiguration itself has a cost. Because operators must pause briefly during migration or scaling, these actions can temporarily slow the system down. By modeling this disruption, EDRP becomes more selective and avoids unnecessary changes. At the same time, the reliance on pause-and-resume migration points to a broader challenge in cloud–edge stream processing: building elasticity mechanisms that can adjust the system in real time without interrupting ongoing processing.

## 1.6 Research Gap

So far, existing systems either do not support heterogeneous environments or fail to integrate multiple key performance metrics, including CPU usage, inbound and outbound bandwidth, and node-to-node latency, into scaling and load balancing decisions in Apache Storm. None of the prior solutions offers operator-level elasticity that works on the fly and without downtime. This gap motivates the need for a unified, multi-metric elasticity mechanism for distributed stream processing on the edge. To address this gap, this work presents an approach that enhances elasticity and load balancing in Apache Storm for distributed stream processing over edge environments. The aim is to lower end-to-end latency and maintain the performance required for real-time stream processing with zero downtime.

# CHAPTER 2

# ARCHITECTURE AND SYSTEM DESIGN

This chapter introduces the system model developed in this research. First, Section 2.1 describes the problem in detail. Then, in Section 2.2, we explain the assumptions made in this research. Section 2.3 then explains the architecture of the proposed model and its main components. The remaining sections of this chapter provide a detailed explanation of the model, including the metrics and algorithms used for performance evaluation and system adaptation.

## 2.1 Research Problem

With the rapid growth of big data and the Internet of Things, stream processing has become one of the major bases for real-time data analysis and decision-making. Apache Storm is a highly deployed, open-source, unified, distributed stream computation system that has been widely used due to its great ability in handling massive volume data streams in real time. Yet, it encounters issues related to elasticity and load balancing in resource-limited environments. The following section outlines these challenges.

### 2.1.1 Apache Storm Limitations

In using Apache Storm in edge computing, a number of challenges come into view, especially for adaptation dynamically to a changeable condition. These challenges become more serious when dealing with the heterogeneous nature and resource limitation of the edge devices. There are several inherent inefficiencies in Apache Storm that limit its scalability in such environments:

- Static Resource Allocation and Limited Resource Awareness: Apache Storm uses a fixed resource allocation strategy that cannot adapt to dynamic workloads. Because of this, over-allocation or underutilization of resources takes place, leading to either network congestion in one case or wasted processing power in another, depending on fluctuating data input rates.
- Lack of elasticity: Storm is not inherently elastic. It cannot scale out/in by itself according to the changes in workload, especially in a distributed edge environment. Although it is

manually configurable, that too is not smooth and effective while dealing with decentralized resources.

- Limited Awareness of Node Heterogeneity: Apache Storm assumes that all nodes in a cluster have similar processing and storage capacities. However, in an edge computing setting, these vary from high-powered servers to low-resource IoT devices. This heterogeneity has a number of consequences: resources are imbalanced, and inefficiencies because Storm cannot optimize the task allocation in accordance with particular node capabilities. It can even lead to overloads and high latency or data loss.
- Load Balancing Inefficiently: Storm's defaults of round-robin or random task assignment usually lead to a very poor balance of loads, wherein a few nodes bear all the loads while the rest remain underutilized, thus aggravating performance in edge environments.

These limitations highlight that Apache Storm is not well-suited for deployment in heterogeneous edge environments where workloads fluctuate and resources are constrained. As a result, the system often suffers from congestion, increased latency during periods of high data input, inefficient load distribution across nodes, and poor resource utilization when workloads decrease. This remains the case unless manual intervention is applied.

### 2.1.2 Research Objectives

The objective of this research is to enhance the scalability and elasticity of Apache Storm in heterogeneous edge environments. Specifically, the proposed solution aims to prevent and resolve bandwidth and CPU congestion, which in turn reduces latency. In addition, the research focuses on achieving efficient load balancing across heterogeneous nodes by leveraging real-time metrics such as CPU usage, bandwidth availability, and workload conditions, ensuring that resources are utilized effectively.

To achieve these objectives, the thesis addresses the following research questions:

1 How can Apache Storm be extended to support autonomous scalability and elasticity in heterogeneous edge environments with fluctuating data rates?

2 What mechanisms can be introduced to detect and mitigate CPU and bandwidth bottlenecks at the node level without requiring system restarts or manual intervention? (The choice of CPU and bandwidth bottlenecks is motivated by their significant impact on performance, as explained in Section 2.4.1)

3 How can real-time metrics, such as CPU usage, bandwidth availability, and latency, be leveraged to have resource-aware load distribution among heterogeneous nodes?

4 To what extent does the proposed approach reduce end-to-end latency compared to Storm's native replication mechanism and non-adaptive strategies?

5 To what extent can the proposed solution itself scale to larger clusters and more complex pipelines while preserving its effectiveness?

## 2.2 Hypothesis and Assumptions

The research is based on a number of assumptions to simplify the system design and to let us concentrate on key performance metrics. The assumptions are as follows:

- Sufficient Memory Resources: We assume the available memory of the system is adequate and does not bottleneck the execution of tasks. This assumption is supported by modern edge devices such as Raspberry Pi 4, with up to 8 GB RAM (Raspberry Pi Foundation), and NVIDIA Jetson Nano, with 4 GB RAM (NVIDIA Corporation (2025)), which typically provide enough memory for stateless stream processing operators and other lightweight data processing tasks.
- Stable Node-to-Node Latency: The system assumes that the latency between any two nodes remains constant during the system execution.
- Fixed Number of Nodes: It is considered that the number of nodes in the cluster is fixed and does not change dynamically during the execution. Every node in the cluster has Docker installed, which allows nodes to run a docker container that serves as a supervisor in an Apache Storm cluster.

- Multiple Operators per Node: We assume that multiple operators can be deployed on a single edge node. These operators may perform different processing tasks on incoming data streams, and in doing so, can alter the size of the outgoing messages.
- Latency Definition: For this work, (end-to-end) latency is defined as the total duration of the data flow through the entire system. It is defined as the difference between the time a data packet was sent by the spout and the time the processing is completed by the sink.
- No Node Failures: Currently our system is unable to handle situations where any node in a cluster fails. We plan to address this in our future work.
- Bolt-Only Replication: In the proposed congestion-handling approach, replication is not applied to either sinks or spouts.
- Stateless Operators: Currently, the solution only supports stateless operators. We intend to expand our work to include support for stateful operators.

## 2.3 High-Level Architecture

This section presents an overview of the high-level architecture of the system, after which the explanation of its individual parts will follow.

The system is composed of three main components. The first component is an extended version of the Apache Storm topology, which includes additional functionalities for metric calculation and load balancing among downstream operators. The second component is the Orchestrator, which monitors the performance of individual Apache Storm operators. It is responsible for detecting bottlenecks in the system and making all decisions related to system elasticity, including the creation and removal of replicas to ensure efficient operation. The third component, which is essential for communication, is the MQTT broker. It facilitates message passing between the Storm topology and the Orchestrator, allowing metric reports and control signals to be exchanged efficiently.

A key feature of our system is the introduction of generic bolts, which are idle bolts that can dynamically act as replicas for any active bolt in the system. A certain number of these generic bolts, which is configurable, are set up in advance and remain idle until they are needed.

The system design ensures that the generic bolts are reusable, which means they can turn back to being idle once other active bolts are able to handle the load. Then the very same generic bolt can serve as a replica of another bolt with a different task when needed again.

Figure 2.1 demonstrates the architecture of the system. Each system component is described in detail in the following sections.

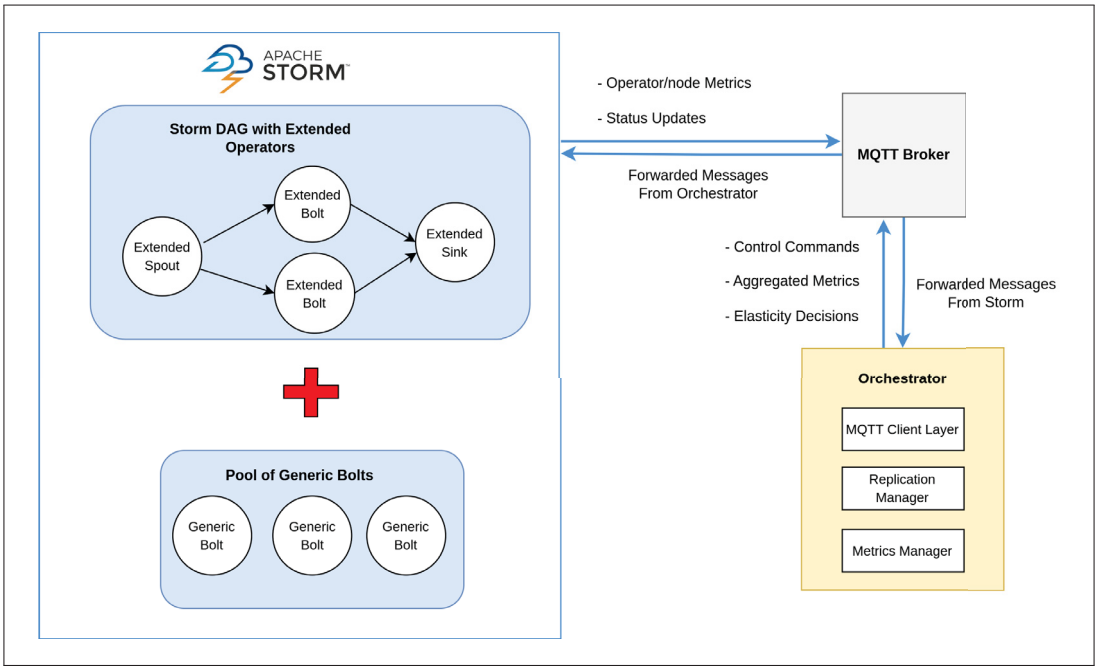



Figure 2.1 System Architecture

### 2.3.1 MQTT Broker

The MQTT Broker is a critical part of the system that provides real-time interactions between the Orchestrator and the Storm topology through the MQTT protocol. It serves as the communication backbone of the system, enabling the exchange of performance information and control messages among all parts of the system, including the Orchestrator and Storm operators. The lightweight

protocol ensures fast and efficient communication, which is an important prerequisite for quick decision-making and overall flexibility of the system.

### 2.3.2 Storm Pipeline

One of the core components of our system is an extended version of Apache Storm, which serves as the stream processing middleware. In this implementation, the base Storm classes for spouts and bolts are extended to support MQTT communication, interact with the Orchestrator, and extract metrics on a regular basis. The operators are further modified to publish these metrics along with other status updates. These extensions give the system visibility into both operator-level and node-level performance and allow operators to respond to control commands such as creating or removing replicas.

In our system, we employ Direct Grouping to connect Storm operators. In this approach, the upstream operator explicitly decides which downstream instance will process each tuple (Detailed explanation is provided in Section 1.4.3.3). This choice gives us fine-grained and deterministic control over tuple distribution, which is important for achieving effective load balancing under real-time system conditions.

However, Direct Grouping also has several disadvantages. First, the responsibility for workload distribution shifts to the developer, which can lead to imbalance. Second, the routing logic is hard-coded into the topology, making the system more tightly coupled and less flexible. Finally, it increases development complexity, since routing decisions must be handled manually rather than relying on Storm's built-in mechanisms.

In our implementation, we mitigate these drawbacks by encapsulating the routing logic in reusable libraries. This removes the burden from developers while preserving the benefits of Direct Grouping.

On top of that, our system also includes a pool of pre-configured bolts whose number will be subject to user control. These are bolts that are, for the time being, idle and not attached to any

operator. They can act as a replica of any active bolt in the system and are therefore known as Generic bolts. Such Generic bolts make the system flexible enough to handle changes in processing dynamically.

All operators, whether regular ones (such as spouts, bolts, or sinks) or generic bolts, communicate with the Orchestrator and with each other through the MQTT broker. They publish metrics and status updates to the broker, and they subscribe to control commands from the Orchestrator as well as the metrics of their downstream operators. This exchange ensures that each operator has visibility into the performance of its downstream operators, which is essential for effective load balancing.

### 2.3.3 Orchestrator

Another critical system component is the Orchestrator. It is an independent Java application that manages system performance, especially when bottlenecks or underutilizations occur.

The orchestrator gathers and monitors various types of information, which provides it with a comprehensive view of the entire system. This data allows it to make well-informed decisions based on the system's real-time status. The following are the key pieces of information it monitors:

- It has a complete overview of the current topology, including how all operators are connected and how data flows through them.
- The Orchestrator collects and stores detailed performance metrics for all operators, including CPU usage and input and output throughput.
- It knows every node's maximum input and output bandwidth.
- The Orchestrator is aware of the latency between each pair of nodes in the cluster.
- It derives node-level performance insights by combining information collected from individual operators.
- The Orchestrator knows which generic bolts are replicas of which bolts.

One of the roles of the Orchestrator is handling bottlenecks. When a bottleneck is detected, the Orchestrator evaluates whether a new replica of an operator should be created to restore smooth data flow. If needed, it also decides the best place to create that replica. On the other hand, to make sure resources aren't being wasted, the Orchestrator also handles underutilization. In these cases, the Orchestrator removes unnecessary replicas and determines which replica should be terminated.

## 2.4 Metrics

In our system, we measure a number of metrics to monitor performance and resource utilization across distributed stream processing operators. These metrics are collected periodically at configurable intervals (typically every 30 seconds) to provide insights into system behavior and enable intelligent scheduling and load balancing decisions. Below is an explanation of how each metric is calculated.

- **Operator CPU Usage:** Our system is launched on Docker, which means the Apache Storm components, including Nimbus and Supervisors, run inside Docker containers. The system is deployed in a way that each worker node in the cluster can run multiple Supervisor containers per topology, with each Supervisor container hosting at most one operator and having adjustable CPU resources allocated specifically for that operator. This containerized approach enables precise monitoring and resource management at the individual operator level. In addition, it allows multiple operators to coexist on the same physical node while maintaining isolation.

  The Docker API is an efficient way to access real-time information about Docker containers. The process works as follows:

  The system first sends two HTTP requests to the Docker API URL, which is created from the node's IP address and the container name, with a delay of 3 seconds between them. Each request returns a snapshot of the container, which gives us insight into the container's resource consumption, including the container's total CPU usage, the host's system CPU

usage, and the number of CPU cores available on the host. The CPU usage percentage is then calculated using the following formula:

$$\text{CPU\_usage\_percentage} = \frac{\text{cpu\_delta}}{\text{system\_cpu\_delta}} \times \frac{\text{node\_cpu\_cores}}{\text{container\_cpu\_limit}} \tag{2.1}$$

Where:

- cpu_delta is the difference between the total CPU usage of the container in the two snapshots.
- system_cpu_delta is the difference between the total system CPU usage in the two snapshots.
- node_cpu_cores is the number of CPUs available on the node.
- container_cpu_limit is the CPU limit allocated to the container (retrieved separately from the container's configuration).

This calculation first determines the container's CPU usage as a fraction of the total system CPU time, then multiplies by the number of online CPUs to get the absolute CPU usage, and finally normalizes this value against the container's allocated CPU limit. The result represents the CPU utilization as a percentage of the container's allocated resources, which can exceed 100% if the container uses more than its allocated limit.

- **Operator Input Throughput:** This metric measures the data rate each operator receives, calculated in megabits per second using the following formula:

$$\text{in\_throughput} = \frac{N_{in} \times S_{in}}{T} \tag{2.2}$$

Where:

- $N_{in}$ is the total number of incoming tuples received during this interval.
- $S_{in}$ is the size of the latest incoming tuple (in megabits).
- T is the interval duration (in seconds).

The system captures the size of the most recent incoming tuple and multiplies it by the total tuple count for the interval, then divides by the interval duration to determine throughput.

- **Operator Output Throughput:** The output throughput metric measures the amount of data in megabits that an operator processes and sends out per second. As shown below, the formula follows a similar structure to input throughput.

$$\text{out\_throughput} = \frac{N_{out} \times S_{out}}{T} \tag{2.3}$$

Where:

- $N_{out}$ is the total number of outgoing tuples processed and sent during the interval.
- $S_{out}$ is the size of the latest outgoing tuple (in megabits).
- T is the interval duration (in seconds).

- **Remaining Input Bandwidth of Node:** The Orchestrator calculates the remaining input bandwidth for each node by periodically summing the input throughput of all operators on that node and subtracting from the node's total available bandwidth:

$$\text{remaining\_BW}_{in} = \text{node\_BW}_{in} - \sum_{i=1}^{n} \text{in\_throughput}_{i} \tag{2.4}$$

Where:

- $\text{remaining\_BW}_{in}$ is the node's remaining input bandwidth (in megabits per second).
- $\text{node\_BW}_{in}$ is the total available input bandwidth of the node (in megabits per second).
- $\text{in\_throughput}_{i}$ is the input throughput of the *i*-th operator which is deployed on the node (in megabits per second).
- n is the total number of operators on the node.

Each operator receives periodic updates about the remaining input bandwidth on its host node, reported both as an absolute value and as a percentage of total capacity.

- **Remaining Output Bandwidth of Node:** The remaining output bandwidth follows the formula below, which is a calculation process similar to that of the remaining input bandwidth.

$$\text{remaining\_BW}_{out} = \text{node\_BW}_{out} - \sum_{i=1}^{n} \text{out\_throughput}_{i} \tag{2.5}$$

Where:

- remaining_$BW_{out}$ is the node's remaining output bandwidth.
- node_$BW_{out}$ is the total available output bandwidth of the node.
- out_$throughput_i$ is the output throughput of the *i*-th operator which is placed on the node.
- n is the total number of operators on the node.

Operators are kept informed through regular updates about how much output bandwidth remains available on their node, expressed both as an absolute value and as a percentage of total capacity.

Table 2.1 summarizes metrics considered in the system.

Table 2.1 Key metrics considered in the system

| **Metric** | **Scope/Level** | **Measured by** | **Unit** |
|---|---|---|---|
| CPU Usage | Operator | Operators | % |
| Input throughput | Operator | Operators | Mbit/s |
| Output throughput | Operator | Operators | Mbit/s |
| Remaining In-Bandwidth | Node | Orchestrator | Mbit/s |
| Remaining In-Bandwidth Percentage | Node | Orchestrator | % |
| Remaining Out-Bandwidth | Node | Orchestrator | Mbit/s |
| Remaining Out-Bandwidth Percentage | Node | Orchestrator | % |

### 2.4.1 Motivation for Metric Selection

The selection of these metrics is motivated by their central role in determining tuple latency and overall system responsiveness. One of the key metrics is CPU usage. At the node level, when a node approaches full utilization, queues begin to form and tuple latency increases rapidly.

We consider both inbound and outbound bandwidth at the node level, since bandwidth saturation has a direct impact on end-to-end latency. The in-bandwidth of a node refers to the maximum sustained incoming data rate it can handle, while its out-bandwidth represents the maximum supported outgoing data rate. Because every inter-node transfer depends simultaneously on the sender's out-bandwidth and the receiver's in-bandwidth, saturation of either bandwidth can slow down communication and increase end-to-end latency. To support more accurate resource-management decisions in heterogeneous environments, we record the remaining capacity in both absolute terms and as a percentage.

For these reasons, this combination of metrics provides a reliable basis for timely adaptation and for avoiding excessive latency in a stream processing system.

The metrics described above serve as inputs to the system's adaptive capabilities, which are detailed in the remainder of this chapter.

## 2.5 Adaptation Mechanisms

In this section, we present our two main adaptation strategies: Local Adaptation and Global Adaptation, which enhance system elasticity, optimize resource utilization, and enable efficient load balancing.

### 2.5.1 Local Adaptation

Local adaptation refers to a method of dynamically distributing tuples across downstream operators by taking into account the resource availability of their host nodes. This strategy is adopted by all topology operators in our system, except for sinks, to balance the workload more effectively.

Figure 2.2 illustrates how local adaptation distributes tuples among downstream replicas based on the resource availability of their host nodes. In this example, operator `Op 1` sends data to four downstream operators: the original bolt `B1` and its three replicas, `gb_1`, `gb_2`, and `gb_3`.

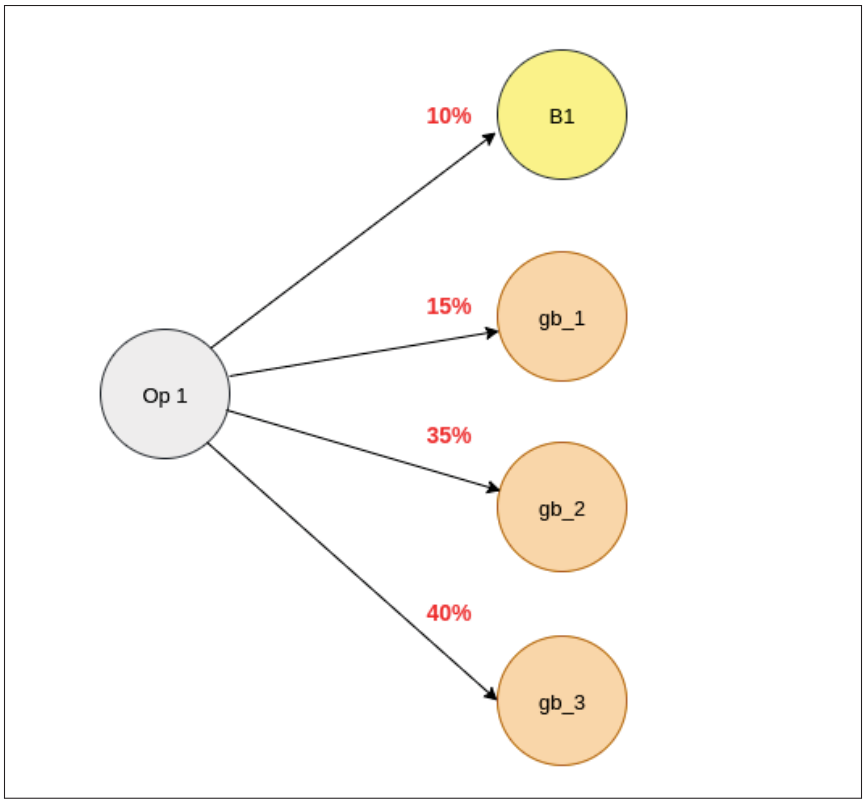



Figure 2.2 Example of local adaptation where operator Op 1 distributes tuples among the original bolt B1 and its replicas (gb_1, gb_2, and gb_3) based on resource availability

The percentages on each connection represent the proportion of tuples directed to each replica. These shares are dynamically adjusted according to real-time metrics. For instance, if `gb_3` runs on a node with higher available resources, it receives a larger portion of the stream, while `B1` and `gb_1` receive smaller shares to prevent overload.

Algorithm 2.1 presents the step-by-step procedure by which operators evaluate downstream node capacities and redistribute tuples accordingly.

The local adaptation relies on the Simple Additive Weighting (SAW) approach, which defines how much data flows to each bolt instance. The overall score of each bolt is the weighted sum of these metrics, and downstream bolts with higher scores receive a larger share of the input data. The weights reflect the relative importance of each metric in the scoring process, and

are constrained to sum to one, thereby maintaining a balanced trade-off among the considered factors.

Algorithm 2.1 Local Adaptation (Part 1)

**Input:** Downstream Replica set $R = \{op_1, op_2, \ldots, op_k\}$;
previous cycle shares $p^{\text{old}}$;
weights $W = \{w_{\text{cpu}}, w_{\text{in}}, w_{\text{out}}, w_{\text{lat}}\}$ with $\sum W = 1$;
thresholds $\Theta = \{\theta^{\text{hi}}_{\text{cpu}}, \theta^{\text{lo}}_{\text{in}}, \theta^{\text{lo}}_{\text{out}}\}$;
smoothing factor $\alpha \in [0, 1]$
**Output:** Updated load distribution $p^{\text{new}}$

**foreach** $op \in R$ **do**
  `// Collect raw performance metrics`
  $\text{cpuUtil}(op) \leftarrow \text{getCPUUtilization}(op)$ `// ∈ [0, 1]`
  $\text{cpuLimit}(op) \leftarrow \text{getCPULimit}(op)$ `// number of cores assigned`
  $\text{absCpuAvail}(op) \leftarrow \text{getAbsCPUAvailable}(op)$
  $\text{rIn}(op) \leftarrow \text{getRemainingInBW}(op)$ `// ∈ [0, 1]`
  $\text{rOut}(op) \leftarrow \text{getRemainingOutBW}(op)$
  $\text{lat}(op) \leftarrow \text{getLatency}(op)$
  `// Detect per-instance congestion`
  $\text{congFlags}[op] \leftarrow \{\text{cpuUtil}(op) \geq \theta^{\text{hi}}_{\text{cpu}}, \text{rIn}(op) \leq \theta^{\text{lo}}_{\text{in}}, \text{rOut}(op) \leq \theta^{\text{lo}}_{\text{out}}\}$;
  `// Set global congestion indicators (0 = no congestion, 1 = congestion exists)`
  **if** $\text{cpuUtil}(op) \geq \theta^{\text{hi}}_{\text{cpu}}$ **then**
    cpuCongExists ← 1
  **if** $\text{rIn}(op) \leq \theta^{\text{lo}}_{\text{in}}$ **then**
    inCongExists ← 1
  **if** $\text{rOut}(op) \leq \theta^{\text{lo}}_{\text{out}}$ **then**
    outCongExists ← 1
  `// Store metrics for each instance`
  $\text{metrics}[op] \leftarrow \{\text{cpuUtil}(op), \text{absCpuAvail}(op), \text{rIn}(op), \text{rOut}(op), \text{lat}(op)\}$;
**end foreach**

`// Adaptive weight adjustment`
$W' \leftarrow \text{AdaptiveWeightAdjustment}(W, \text{cpuCongExists}, \text{inCongExists}, \text{outCongExists}, D)$;
`// Update global min/max for each metric`
updateGlobalMetricRanges($R$)

The metrics considered in the local adaptation process are summarized below.

Algorithm 2.2 Local Adaptation (Part 2)

```
   // Calculate preference scores using normalized metrics
19 foreach op ∈ R do
       // Apply sigmoid normalization to metrics
20     x_cpu ← sigmoidNormalize(cpuAvail(op), benefit)
21     x_in ← sigmoidNormalize(rIn(op), benefit)
22     x_out ← sigmoidNormalize(rOut(op), benefit)
23     x_lat ← sigmoidNormalize(lat(op), cost)
       // Calculate weighted composite score
24     score[op] ← w'_cpu · x_cpu + w'_in · x_in + w'_out · x_out + w'_lat · x_lat
       // Apply additional severity-based penalties if needed
25     if severeCongestion(op) then
26         score[op] ← score[op] × penaltyFactor
27 end foreach
   // Convert scores to target percentages
28 p^raw ← derivePercentagesFromScores(score, 100)
   // Apply smoothing
29 foreach op ∈ R do
30     p^smooth(op) ← α · p^raw(op) + (1 − α) · p^old(op)
31 end foreach
32 normalizeToSum(p^smooth, 100)
   // Apply maximum change limit
33 L ← getMaxChangeLimit(R, cpuCongExists, inCongExists, outCongExists)
34 foreach op ∈ R do
35     if Δ > L then
36         p^new(op) ← min(p^old(op) + L, p^smooth(op))
37 end foreach
38 normalizeToSum(p^new, 100)
39 return p^new
```

- CPU usage and capacity: Bolts running on nodes with lower utilization and greater available CPU cores receive a higher score (lines 1-4).
- Remaining in-bandwidth: Bolts on nodes with more available bandwidth get a higher score (line 5).
- Remaining out-bandwidth: A higher score is assigned to bolts whose host nodes have a greater free outgoing bandwidth capacity (line 6).

- Latency: Bolts with lower latency are prioritized and receive a higher score (line 7).

The algorithm begins by detecting congestion through a comparison of current resource utilization against predefined thresholds (line 8). To capture these conditions, the system sets a global flag for each resource type: CPU, input bandwidth, and output bandwidth are each represented by a separate binary indicator. A flag is activated (set to 1) if at least one downstream replica exceeds the threshold for that resource; otherwise, it remains inactive (lines 9–14). These global flags are essential for guiding the weight adjustment process, as they provide a system-wide view of the types of congestion currently present. The `AdaptiveWeightAdjustment` method increases the weights of congested metrics, giving them greater importance in the scoring process (line 17). This algorithm will be explained in detail later.

In the next step, the algorithm updates the global minimum and maximum values observed for each metric (line 18). These ranges are essential for normalization, as they allow raw measurements to be transformed consistently into a comparable [0,1] scale (lines 20–23). Then, the algorithm computes the base score for each downstream replica by combining the normalized metric values with their corresponding weights (line 24). More details on the normalization approach are provided in Section 2.5.1.2.

If a downstream replica is affected by congestion, the algorithm applies a penalty to its base score (lines 25 and 26). This reduction ensures that congested replicas receive a lower priority in tuple distribution compared to those operating under normal conditions. The scores are then converted into percentage shares that sum to 100, giving each downstream replica its proportional allocation (line 29).

To avoid sudden jumps in load distribution and to keep the allocation stable over time, the new share of each replica is blended with its previous share using exponential smoothing (lines 31–33). The smoothing factor $\alpha \in [0, 1]$ determines how much weight is given to the new score versus the previous cycle. In our implementation, $\alpha$ is set to 0.7, meaning that 70% of the updated score influences the new share, while 30% comes from the prior cycle. This balance

allows the system to react promptly to changes in resource conditions without causing abrupt shifts in tuple routing.

The smoothing can be expressed as:

$$p^{smooth} = \alpha\, p^{raw} + (1 - \alpha)\, p^{old} \tag{2.6}$$

where $p^{smooth}$ is the adjusted share, $p^{raw}$ is the share derived from the current scores, and $p^{old}$ is the share from the previous cycle.

After smoothing, the algorithm applies a rate-limiting step to ensure that no replica's share changes too abruptly within a single cycle (lines 34–37). This step sets a maximum allowed increase or decrease in share, preventing sudden shifts that could destabilize the system. The exact limit is adjusted based on the active congestion flags. When congestion is detected the algorithm allows faster adaptation to quickly relieve bottlenecks. Also, when a new replica has just been activated the algorithm permits faster adjustment so that the replica can be integrated efficiently. Under normal conditions, however, changes remain gradual to maintain stability in tuple distribution.

In the final step, the adjusted shares are normalized using the sigmoid function to ensure the adjusted shares sum to 100 (line 38).

To enhance the effectiveness of local adaptation, two helper algorithms are introduced: `AdaptiveWeightAdjustment` and `SigmoidNormalize`. These supporting algorithms handle the dynamic weighting of metrics according to the state of the system and the normalization of data, respectively. The following parts describe each algorithm in detail, along with its pseudocode.

#### 2.5.1.1 Adaptive Weight Adjustment

This method updates the metric weights based on the current congestion state. As shown in Algorithm 2.3, it takes the global congestion flags and the weight vector $\{w_{\text{cpu}}, w_{\text{in}}, w_{\text{out}}, w_{\text{lat}}\}$ and returns an adjusted vector.

Algorithm 2.3 Adaptive Weight Adjustment Algorithm

```
// Function: Adaptive Weight Adjustment(W)
Input: Original weights W = {w_cpu, w_in, w_out, w_lat};
Global congestion flags {cpuCongExists, inCongExists, outCongExists}
Output: Adjusted weights W' = {w'_cpu, w'_in, w'_out, w'_lat}

// Count congestion types and determine shift strategy
numCongestedTypes ← |cpuCongExists| + |inCongExists| + |outCongExists|
if numCongestedTypes = 3 then
    offset ← 0.33 × w_lat
    w'_lat ← 0
else if numCongestedTypes = 2 then
    offset ← 0.5 × w_lat
    w'_lat ← 0
end if
else if numCongestedTypes = 1 then
    if allReplicasFree(cpu) then
        offset ← w_cpu, w'_cpu ← 0
    else if allReplicasFree(inBW) then
        offset ← w_in, w'_in ← 0
    end if
    else if allReplicasFree(outBW) then
        offset ← w_out, w'_out ← 0
    end if
    else
        offset ← w_lat, w'_lat ← 0
    end if
end if
else
    offset ← 0
end if

// Redistribute offset to congested metrics
w'_cpu ← w_cpu + offset × cpuCongExists
w'_in ← w_in + offset × inCongExists
w'_out ← w_out + offset × outCongExists
return W'
```

First, it checks which flags are active (line 1). If CPU, input bandwidth, and output bandwidth are all congested, a share of the latency weight is divided equally among them (lines 2–4, 26–28). If two types of congestion are present, the latency weight is split evenly between the two metrics (lines 5–7, 26–28). If only one flag is active, the algorithm checks whether the other two metrics are completely free of congestion. If so, the offset comes from that non-congested metric; otherwise, it is taken from the latency weight (lines 9–21). Finally, the updated weights are normalized so they sum to one (lines 26–28).

This adjustment ensures that whenever congestion is detected, the affected metric has more influence in the scoring process, while latency still remains part of the evaluation.

#### 2.5.1.2 Sigmoid Normalize

We normalize all metrics using SigmoidNormalize (Algorithm 2.4). It maps each raw metric to the [0,1] interval in a uniform way for both benefit and cost metrics so that they become comparable within the SAW scoring system. We favor a sigmoid over linear scaling because it is bounded and less sensitive at the extremes, which keeps shares smoother and more stable under congestion.

The `SigmoidNormalize` function works with five inputs. It receives the raw metric value, the lower and upper bounds ($min$ and $max$), the metric type defined as either benefit or cost, and a congestion indicator ($isCongested$). Benefit metrics are those where larger values indicate a more favorable condition for routing, such as remaining input bandwidth, remaining output bandwidth, or absolute free CPU. Cost metrics are those where smaller values are preferable, such as latency. The output is a normalized score in the interval [0, 1], ensuring comparability across different metrics.

Under normal conditions (not congested), we apply standard sigmoid normalization (lines 1-10). The function first looks at the current range (line 2). If the range is zero (or effectively zero), $min$ and $max$ are equal, which means all replicas reported the same value in that cycle. In that case, it returns a neutral score of 0.5 to avoid creating an artificial preference (lines 3-5).

Algorithm 2.4 Normalization Algorithm

```
// Function: sigmoidNormalize(value, min, max, type, isCongested)
// cost metrics ↓ lower = better
// benefit metrics ↑ higher = better
Input: raw metric value;
normalization range (min, max);
type ∈ {benefit, cost};
isCongested ;
Output: Normalized value ∈ [0, 1]
1  if ¬isCongested then
       // Standard sigmoid normalization path
2      range ← max − min;
3      if range < ϵ then
4          return 0.5                                   // identical values case
5      else
6          alignedValue ← { max − value   if type = cost
                          { value − min   if type = benefit
7          return sigmoidTransform(alignedValue)
8      end if
9  else
       // Aggressive penalty for congested nodes
10     forcedValue ← penalizeCongestedNode(value, type)
11     return sigmoidNormalize(forcedValue, min, max, type, false)
12 end if
```

For cost metrics, lower raw values should yield higher scores, so it uses (*max* - *value*). For benefit metrics, higher raw values should yield higher normalized scores, so it uses (*value* - *min*) instead (line 6). The aligned value is then passed to the sigmoid transform method to produce a smooth output.

During congestion (lines 9-12), the algorithm switches to an aggressive penalty mode. Congested nodes are deliberately assigned poor values before normalization. For benefit metrics, the value is forced toward the low end, and for cost metrics, it is pushed toward the high end. This creates strong penalties that quickly redirect traffic away from bottlenecked resources.

We use a sigmoid instead of a purely linear scale because it is bounded and less sensitive at the extremes. Linear scaling can exaggerate tiny differences when the observed range is narrow; the sigmoid prevents sudden spikes that may arise in small value ranges, and ensures the shares remain smooth and stable, particularly in congested situations.

### 2.5.2 Global Adaptation

Global adaptation is the collective name assigned to all mechanisms implemented by the Orchestrator at the topology level to provide elasticity to the system.

Unlike local adaptation, which redistributes load among existing operator instances, global adaptation dynamically adjusts the number of operator replicas in response to performance issues. The Orchestrator continuously monitors all operators in the topology and makes decisions based on comprehensive system-wide analysis rather than localized metrics.

When bottlenecks are detected, the Orchestrator first evaluates whether the collective capacity of all existing instances of that operator type can handle the current workload. Only when this analysis reveals insufficient aggregate capacity does the system proceed to spawn a new replica. The replica placement decision considers multiple factors including available node resources, network topology, and load distribution requirements.

Moreover, the Orchestrator monitors for underutilization scenarios where operator replicas do not consume much resources and do not process much data. Through systematic analysis, it identifies candidates for removal. The system ensures that replica removal will not compromise performance requirements before proceeding with deallocation, maintaining system efficiency while avoiding unnecessary resource waste.

To organize the discussion, we first describe when scale-out is triggered in Section 2.5.2.1 and how replicas are placed in Section 2.5.2.2, then outline the conditions for scale-down in Section 2.5.2.3 and the procedure for safe replica removal in Section 2.5.2.4.

#### 2.5.2.1 Scale-Out Triggers

Scale out triggers are the conditions that prompt the Orchestrator to create additional operator replicas to alleviate performance bottlenecks. The system employs threshold-based detection mechanisms to identify when existing operator instances can no longer handle the workload effectively.

When signs of bottlenecks appear in the system, the Orchestrator must determine if local adaptation can resolve the issue or if additional replicas are required. Each bottleneck type requires specific detection criteria and placement considerations to ensure effective resolution through replica creation

The following part examines how the system detects CPU bottlenecks and what conditions must be present for the Orchestrator to trigger the replication process.

##### 2.5.2.1.1 CPU Bottlenecks

A CPU bottleneck occurs when a operator's CPU utilization approaches maximum, indicating that sending more data may degrade performance. Thus, the operators need to inform the Orchestrator about potential bottlenecks to prevent performance issues.

Algorithm 2.5 determines whether an operator requires scaling out by calculating the aggregate CPU utilization across all its replicas. The process initializes three variables (lines 1-3) and retrieves all replicas of the operator (line 4).

The core computation occurs in the lines 5 to 10, where each replica contributes its absolute CPU consumption to the total. The algorithm multiplies each replica's CPU utilization percentage by its allocated CPU limit (lines 7-8), converting relative utilization into absolute CPU cores consumed. This approach correctly accounts for heterogeneous environments where replicas may have different CPU allocations.

Algorithm 2.5 CPU Bottleneck Detection

**Input:** operator repository $\mathcal{O}$ (metadata for all operators);
$\theta_{\text{avg}}^{\text{CPU}}$ (avg-replica CPU threshold);
**Trigger:** Periodic monitoring or when CPU bottleneck signal is received;
**Output:** needScaleOut $\in$ {true, false} ;

```
1  avgCPU ← 0.0
2  avgCPU_prev ← 0.0
3  totalLimit ← 0.0
4  replicas ← findReplicas(op)
5  foreach r in replicas do
6  |   limit ← O[r].getCpuLimit()
7  |   avgCPU ← avgCPU + (O[r].getCpu() × limit)
8  |   avgCPU_prev ← avgCPU_prev + (O[r].getCpuPrevCycle() × limit)
9  |   totalLimit ← totalLimit + limit
10 end foreach
11 avgCPU ← avgCPU / totalLimit
12 avgCPU_prev ← avgCPU_prev / totalLimit
13 if avgCPU ≥ θ_avg^CPU and avgCPU_prev ≥ θ_avg^CPU then
14 |   needScaleOut ← true
15 end if
16 else
17 |   needScaleOut ← false
18 end if
19 return needScaleOut
```

The accumulated absolute CPU consumption values are then divided by the total allocated CPU capacity (lines 11-12). This calculation produces a weighted average CPU utilization rate that represents the overall CPU demand across all replicas of the operator type. The resulting percentage indicates how intensively the operator type is using its collective allocated CPU resources, providing a single metric that accounts for both the individual utilization rates and the relative capacity of each replica.

Due to inherent inaccuracies in CPU measurement, the decision logic (line 13) examines both the current cycle and the previous cycle to ensure the high utilization signal is not merely noise. This dual-condition approach prevents unnecessary scaling triggered by temporary measurement fluctuations or brief CPU spikes. The algorithm sets $needScaleOut$ to true only when both

current and previous cycle averages exceed the threshold (line 14), confirming that sustained high utilization genuinely requires additional computational capacity. Otherwise, it sets the variable to false (line 17).

If the algorithm returns true, replication may be considered by the Orchestrator. The detailed process will be explained after both CPU and bandwidth triggers are discussed.

#### 2.5.2.1.2 Bandwidth Bottlenecks

Bandwidth bottlenecks represent another critical trigger for the replication process. While CPU congestion reflects limits in computational capacity, bandwidth bottlenecks arise when the communication channels of a node are close to saturation, either for incoming or outgoing traffic. If left unresolved, they can severely restrict the flow of data between operators and create system-wide delays. To address this, the Orchestrator periodically monitors bandwidth usage across all nodes and applies Algorithm 2.6 to detect when replication is necessary.

Algorithm 2.6 detects bandwidth bottlenecks across the cluster and determines which operators require replication. It operates in two phases: first, calculating bandwidth usage for all nodes, then identifying bottlenecks and validating replication needs. The process begins by iterating through each node in the cluster and calculating total bandwidth consumption. For each node, it sums the input throughput of all operators hosted on that node (line 2) and similarly sums the output throughput (line 3). These values are stored directly in the node registry $N$, making them available for subsequent analysis.

The second loop in Algorithm 2.6 processes each node to identify bandwidth constraints. Lines 6-7 calculate the remaining bandwidth percentages by subtracting used bandwidth from total capacity and normalizing by the total. This produces values between 0 and 1, where lower values indicate less remaining bandwidth available for use.

When remaining input bandwidth falls below the threshold (line 8), Algorithm 2.6 identifies the operator consuming the most input bandwidth on that node (line 9). However, rather than

Algorithm 2.6 Bandwidth Bottleneck Detection

**Input:** operator repository $\mathcal{O}$ (metadata for all operators),
Node registry $\mathcal{N}$ (node bandwidth capacity and usage),
$\theta_{\text{inBW}}$, $\theta_{\text{outBW}}$ (bandwidth usage thresholds),
$\theta_{\text{suff}}$ (sufficient capacity threshold);
**Trigger:** Executed periodically by Orchestrator;
**Output:** needScaleOut $\in$ {true, false}, $\text{op}_{\text{toReplicate}}$ ;

**foreach** node *in* cluster **do**
  $\mathcal{N}[\text{node}].\text{inBW}_{\text{used}} \leftarrow \sum_{\text{op} \in \text{node}} \mathcal{O}[\text{op}].\text{inThr}$
  $\mathcal{N}[\text{node}].\text{outBW}_{\text{used}} \leftarrow \sum_{\text{op} \in \text{node}} \mathcal{O}[\text{op}].\text{outThr}$
**end foreach**
**foreach** node *in* cluster **do**
  $\text{inBW}_{\text{rem}} \leftarrow (\mathcal{N}[\text{node}].\text{inBW}_{\text{total}} - \mathcal{N}[\text{node}].\text{inBW}_{\text{used}}) \,/\, \mathcal{N}[\text{node}].\text{inBW}_{\text{total}}$
  $\text{outBW}_{\text{rem}} \leftarrow (\mathcal{N}[\text{node}].\text{outBW}_{\text{total}} - \mathcal{N}[\text{node}].\text{outBW}_{\text{used}}) \,/\, \mathcal{N}[\text{node}].\text{outBW}_{\text{total}}$
  **if** $\text{inBW}_{\text{rem}} \leq \theta_{\text{inBW}}$ **then**
    $\text{op}_{\text{inBW\_intensive}} \leftarrow \text{findHighestInBW}(\text{node})$
    $\text{replicas} \leftarrow \text{findReplicas}(\text{op}_{\text{inBW\_intensive}})$
    $\text{avgRemBW} \leftarrow 0$
    **foreach** r *in* replicas **do**
      $\text{host} \leftarrow \mathcal{O}[\text{r}].\text{getHost}()$
      $\text{avgRemBW} \leftarrow \text{avgRemBW} + \mathcal{N}[\text{host}].\text{inBW}_{\text{rem}}$
    **end foreach**
    $\text{avgRemBW} \leftarrow \text{avgRemBW} \,/\, |\text{replicas}|$
    **if** $\text{avgRemBW} < \theta_{\text{suff}}$ **then**
      $\text{op}_{\text{toReplicate}} \leftarrow \text{op}_{\text{inBW\_intensive}}$
      $\text{needScaleOut} \leftarrow \text{true}$
    **end if**
  **end if**
  **if** $\text{outBW}_{\text{rem}} \leq \theta_{\text{outBW}}$ **then**
    $\text{op}_{\text{outBW\_intensive}} \leftarrow \text{findHighestOutBW}(\text{node})$
    $\text{replicas} \leftarrow \text{findReplicas}(\text{op}_{\text{outBW\_intensive}})$
    $\text{avgRemBW} \leftarrow 0$
    **foreach** r *in* replicas **do**
      $\text{host} \leftarrow \mathcal{O}[\text{r}].\text{getHost}()$
      $\text{avgRemBW} \leftarrow \text{avgRemBW} + \mathcal{N}[\text{host}].\text{outBW}_{\text{rem}}$
    **end foreach**
    $\text{avgRemBW} \leftarrow \text{avgRemBW} \,/\, |\text{replicas}|$
    **if** $\text{avgRemBW} < \theta_{\text{suff}}$ **then**
      $\text{op}_{\text{toReplicate}} \leftarrow \text{op}_{\text{outBW\_intensive}}$
      $\text{needScaleOut} \leftarrow \text{true}$
    **end if**
  **end if**
**end foreach**

immediately triggering replication, the algorithm performs a critical validation step. It retrieves all replicas of this operator type (line 10) and calculates the average remaining input bandwidth across all their host nodes (lines 11-16). This weighted assessment determines whether the operator type as a whole lacks capacity or if the issue can be solved by load redistribution through local adaptation.

Only when the average remaining bandwidth across all replicas drops below the defined threshold (line 17), the Algorithm marks the operator for replication (line 18) and sets the scale-out flag (line 19). This two-step check avoids creating replicas unnecessarily.

The output bandwidth analysis follows an identical pattern to the input bandwidth check. Lines 22-35 mirror the logic from lines 8-21, examining output bandwidth constraints and applying the same validation mechanism to ensure replication is genuinely needed.

If the algorithm returns false, the situation is eventually handled by local adaptation. Over the following cycles, local adaptation gradually reduces the flow of data sent to busy nodes, easing their load and preventing them from turning into bottlenecks. If it returns true, the Orchestrator proceeds to trigger the replication process to relieve pressure on the congested node.

If either bottleneck detection algorithm (Algorithm 2.5 or 2.6) indicates that scaling is needed, the Orchestrator initiates the replication process, as described in the following section.

#### 2.5.2.2 Replication Process

The replication process involves two main phases. First, the system selects the optimal node to host the new replica, and second, the replica is integrated into the topology. Each step is explained in detail below:

1. **Selecting the Best Candidate for Replication:**
   Algorithm 2.7 shows this phase. The process begins by identifying eligible nodes for replica placement. The algorithm first retrieves the set of nodes already hosting replicas of the

Algorithm 2.7 Candidate Selection and Scoring

**Input:** operator repository $\mathcal{O}$ (metadata for all operators);
Node registry $\mathcal{N}$ (node resources and availability);
Topology structure $\mathcal{T}$ (operator connections);
$\text{op}_{\text{toReplicate}}$ (operator requiring replication);
Resource thresholds and metric weights;
**Output:** Best candidate;

// Step 1: Identify candidate nodes
existingHosts $\leftarrow \{\mathcal{O}[\text{r}].\text{host} \mid \text{r} \in \text{findReplicas}(\text{op}_{\text{toReplicate}})\}$
**foreach** gb *in* genericBolts **do**
  **if** $\mathcal{O}[\text{gb}].\text{host} \notin$ existingHosts ***and***
  $\mathcal{N}[\mathcal{O}[\text{gb}].\text{host}].\text{CPU} < \theta_{\text{maxCPU}}$ ***and***
  $\mathcal{N}[\mathcal{O}[\text{gb}].\text{host}].\text{inBW}_{\text{rem}} > \theta_{\text{minBW}}$ **then**
    upstreams $\leftarrow$ findUpstreamOps($\text{op}_{\text{toReplicate}}$)
    downstreams $\leftarrow \mathcal{T}[\text{op}_{\text{toReplicate}}]$
    $\text{avgLatency}_{\text{up}} \leftarrow 0$
    **foreach** us *in* upstreams **do**
      $\text{avgLatency}_{\text{up}} \leftarrow \text{avgLatency}_{\text{up}} + \text{getLatency}(\text{us}, \text{gb})$
    **end foreach**
    $\text{avgLatency}_{\text{up}} \leftarrow \text{avgLatency}_{\text{up}} / |\text{upstreams}|$
    $\text{avgLatency}_{\text{down}} \leftarrow 0$
    **foreach** ds *in* downstreams **do**
      $\text{avgLatency}_{\text{down}} \leftarrow \text{avgLatency}_{\text{down}} + \text{getLatency}(\text{ds}, \text{gb})$
    **end foreach**
    $\text{avgLatency}_{\text{down}} \leftarrow \text{avgLatency}_{\text{down}} / |\text{downstreams}|$
    $\text{latency}_{\text{total}} \leftarrow \text{avgLatency}_{\text{up}} + \text{avgLatency}_{\text{down}}$
    candidates[gb] $\leftarrow \{\text{CPU}, \text{latency}_{\text{total}}, \text{inBW}_{\text{rem}}, \text{outBW}_{\text{rem}}\}$
  **end if**
**end foreach**
// Step 2: Score and select best candidate
$\text{CPU}_{\text{min}}, \text{CPU}_{\text{max}} \leftarrow$ findMinMax(candidates, "CPU")
$\text{lat}_{\text{min}}, \text{lat}_{\text{max}} \leftarrow$ findMinMax(candidates, "latency")
$\text{inBW}_{\text{min}}, \text{inBW}_{\text{max}} \leftarrow$ findMinMax(candidates, "inBW")
$\text{outBW}_{\text{min}}, \text{outBW}_{\text{max}} \leftarrow$ findMinMax(candidates, "outBW")
**foreach** cand *in* candidates **do**
  $\text{CPU}_{\text{norm}} \leftarrow \text{normalize}(\text{cand.CPU}, \text{CPU}_{\text{min}}, \text{CPU}_{\text{max}}, \text{cost})$
  $\text{lat}_{\text{norm}} \leftarrow \text{normalize}(\text{cand.latency}, \text{lat}_{\text{min}}, \text{lat}_{\text{max}}, \text{cost})$
  $\text{inBW}_{\text{norm}} \leftarrow \text{normalize}(\text{cand.inBW}, \text{inBW}_{\text{min}}, \text{inBW}_{\text{max}}, \text{benefit})$
  $\text{outBW}_{\text{norm}} \leftarrow \text{normalize}(\text{cand.outBW}, \text{outBW}_{\text{min}}, \text{outBW}_{\text{max}}, \text{benefit})$
  $\text{score} \leftarrow (w_{\text{CPU}} \times \text{CPU}_{\text{norm}}) + (w_{\text{lat}} \times \text{lat}_{\text{norm}}) + (w_{\text{inBW}} \times \text{inBW}_{\text{norm}}) + (w_{\text{outBW}} \times \text{outBW}_{\text{norm}})$
**end foreach**
bestCandidate $\leftarrow$ selectMaxScore(candidates)
**return** bestCandidate

operator type (line 1), ensuring that the new replica will be placed on a different node to maintain distribution.

For each generic bolt in the system, the algorithm evaluates whether it meets the placement criteria. A generic bolt qualifies as a candidate if its host node is not already running this operator type, the node's CPU usage is below the maximum threshold, and the node has sufficient remaining input bandwidth (lines 2-5). These constraints ensure that the replica will be placed on a node with adequate resources.

For each qualifying candidate, the algorithm calculates network proximity metrics. It retrieves the upstream operators (line 6) and downstream operators (line 7) of the operator being replicated. The algorithm then computes the average latency from all upstream operators to the candidate node (lines 8-12) and similarly calculates the average latency from the candidate to all downstream operators (lines 13-17). These latency values are summed to produce a total communication cost metric (line 18), which reflects how well-positioned the candidate is within the data flow path.

Once all candidates are identified (line 19), the algorithm proceeds to the scoring phase. First, it determines the minimum and maximum values for each metric across all candidates (lines 22-25), which are needed for normalization.

All these metrics are then normalized so that they may be comparable. CPU and latency are cost metrics where lower values are better, so they are normalized accordingly (lines 27-28). Input and output bandwidth are benefit metrics where higher values are preferred (lines 29-30). Each normalized metric is multiplied by its corresponding weight and summed to produce a composite score (line 31).

Finally, the algorithm selects the candidate with the highest score (line 33) and returns it as the optimal placement choice (line 34).

2. **Replica Integration:** Once the optimal candidate node is selected from Algorithm 2.7, Algorithm 2.8 manages the integration of the new replica into the topology. The integration process establishes all necessary communication channels between the replica and its neighboring operators.

Algorithm 2.8 Replica Integration

**Input:** operator repository $\mathcal{O}$ (metadata for all operators);
Topology structure $\mathcal{T}$ (operator connections);
$\text{op}_{\text{toReplicate}}$ (operator requiring replication);
Resource thresholds $\theta_{\text{maxCPU}}$, $\theta_{\text{minBW}}$;
$w_{\text{CPU}}$, $w_{\text{lat}}$, $w_{\text{inBW}}$, $w_{\text{outBW}}$ (Metric weights);
**Output:** Integrate the new replica;

1 $\text{op}_{\text{replica}} \leftarrow$ bestCandidate
2 downstreams $\leftarrow \mathcal{T}[\text{op}_{\text{toReplicate}}]$
3 **foreach** ds *in* downstreams **do**
4 | notify(ds, $\text{op}_{\text{replica}}, \mathcal{O}[\text{op}_{\text{replica}}]$)
5 **end foreach**
6 **foreach** ds *in* downstreams **do**
7 | $\mathcal{O}$[ds].serverPort $\leftarrow$ waitForPort(ds)
8 **end foreach**
9 activateReplica(downstreams)
10 upstreams $\leftarrow$ findUpstreamOps($\text{op}_{\text{toReplicate}}$)
11 **foreach** us *in* upstreams **do**
12 | $\mathcal{O}[\text{op}_{\text{replica}}]$.serverPort $\leftarrow$ waitForPort($\text{op}_{\text{replica}}$)
13 | notify(us, $\{\text{op}_{\text{replica}}, \mathcal{O}[\text{op}_{\text{replica}}]$)
14 | $\mathcal{T}$[us].add($\text{op}_{\text{replica}}$)
15 **end foreach**
16 $\mathcal{T}[\text{op}_{\text{replica}}] \leftarrow \mathcal{T}[\text{op}_{\text{toReplicate}}]$

The algorithm begins by assigning the selected candidate as the new replica (line 1) and retrieving all downstream operators that the original operator communicates with (line 2). These downstream operators need to be notified about the new replica so they can prepare to receive data from it. Each downstream operator is informed about the new replica's existence and host location (lines 3-5). This notification triggers each downstream operator to allocate a server port for receiving connections from the replica. The algorithm then waits to receive the allocated server port and stores this information in the operator repository under the downstream operator's entry (line 6-8). This ensures that the replica will know exactly where to send data to each of its downstream neighbors.

After collecting all downstream connection details, the replica is activated (line 9). At this point, the replica has all the information it needs to establish outgoing connections to its downstream operators and can begin processing tuples.

The algorithm then handles the upstream integration. It identifies all upstream operators that feed data to the original operator (line 10). For each upstream operator (lines 11-15), the replica allocates its own server port (line 12), which is stored in the replica's repository entry. This port information, along with the replica's host location, is communicated to each upstream operator (line 13). Each upstream operator adds the replica to its routing table, enabling it to begin directing tuples to the new replica according to its load balancing policy. Finally, the Orchestrator updates its global view of the topology to reflect the addition of the replica (line 14 and 16).

#### 2.5.2.3 Scale-Down Triggers

To prevent resource wastage while maintaining system performance, our scale-down mechanism identifies and removes underutilized bolt replicas. Each bolt in the system continuously evaluates its resource utilization against predefined thresholds. When the bolt itself recognizes underutilization, it signals the Orchestrator to trigger appropriate scaling actions if needed. The detection process follows a systematic approach as explained in Algorithm 2.9. It identifies replicas that can be safely removed from underutilized operators without causing performance degradation. The algorithm evaluates each replica by simulating its removal and verifying that remaining replicas can handle the workload.

The algorithm iterates through operators flagged as underutilized (line 1) and retrieves all replicas of each operator type (line 2). Then it calculates the total CPU consumption across all replicas using weighted averaging (lines 3-6). For each replica, it multiplies the CPU utilization percentage by the replica's allocated CPU limit, accumulating both current and previous cycle values (lines 5-6). The total CPU limit is also accumulated (line 7) for later use in projecting post-removal CPU usage.

Algorithm 2.9 Underutilization Detection

**Input:** operator repository $\mathcal{O}$ (metadata for all operators);
Node registry $\mathcal{N}$ (node bandwidth information);
underUtilizedOps (list of flagged operators);
Thresholds: $\theta_{\text{safeCPU}}$, $\theta_{\text{safeBW}}$
**Output:** candidates (list of candidates that can be removed);

**foreach** op *in* underUtilizedOps **do**
  replicas ← findReplicas(op)
  `// Calculate average CPU across all replicas`
  **foreach** r *in* replicas **do**
    limit ← $\mathcal{O}$[r].getCpuLimit()
    sumCPU ← sumCPU + ($\mathcal{O}$[r].getCpu() × limit)
    $\text{sumCPU}_{\text{prev}}$ ← $\text{sumCPU}_{\text{prev}}$ + ($\mathcal{O}$[r].getCpuPrevCycle() × limit)
    totalLimit ← totalLimit + limit
  **end foreach**
  `// Evaluate each replica for removal`
  **foreach** r *in* replicas **do**
    **if** r *is generic bolt* **then**
      `// Get hosts that would remain after removing r`
      hosts ← {}
      **foreach** rep *in* replicas **do**
        **if** rep ≠ r **then**
          hosts.add($\mathcal{O}$[rep].host)
        **end if**
      **end foreach**
      $\text{remInBW} \leftarrow \sum_{\text{h}\in\text{hosts}} \mathcal{N}[\text{h}].\text{inBW}_{\text{rem}}$
      $\text{remOutBW} \leftarrow \sum_{\text{h}\in\text{hosts}} \mathcal{N}[\text{h}].\text{outBW}_{\text{rem}}$
      $\text{totalInBW} \leftarrow \sum_{\text{h}\in\text{hosts}} \mathcal{N}[\text{h}].\text{inBW}_{\text{total}}$
      $\text{totalOutBW} \leftarrow \sum_{\text{h}\in\text{hosts}} \mathcal{N}[\text{h}].\text{outBW}_{\text{total}}$
      $\text{remInBW}_{\text{projected}} \leftarrow \text{remInBW} - \mathcal{O}[\text{r}].\text{inThr}$
      $\text{remOutBW}_{\text{projected}} \leftarrow \text{remOutBW} - \mathcal{O}[\text{r}].\text{outThr}$
      $\text{CPU}_{\text{projected}} \leftarrow \text{sumCPU} / (\text{totalLimit} - \mathcal{O}[\text{r}].\text{cpuLimit})$
      **if** $(\text{remInBW}_{\text{projected}} / \text{totalInBW}) > \theta_{\text{safeBW}}$ ***and***
      $(\text{remOutBW}_{\text{projected}} / \text{totalOutBW}) > \theta_{\text{safeBW}}$ ***and***
      $\text{CPU}_{\text{projected}} \leq \theta_{\text{safeCPU}}$ **then**
        candidates[r] ← {remInBW, remOutBW, $\mathcal{O}$[r].CPU}
      **end if**
    **end if**
  **end foreach**
**end foreach**
**return** candidates

For each replica in the operator type, the algorithm checks if it's a generic bolt eligible for removal. Original operators cannot be removed as they represent the base functionality (lines 10).

The algorithm builds a set of host nodes that would continue hosting replicas if candidate $r$ were removed (lines 12-15). It does this by going through all replicas and recording their hosts, except for the one being evaluated. In lines 20-23, the algorithm aggregates bandwidth information across all remaining hosts (excluding r). Using summation notation, it calculates both the available remaining bandwidth and total bandwidth capacity for input and output directions, representing the resources that would be available to the remaining replicas.

After that, it projects what resources would look like after removing candidate $r$. It subtracts the candidate's input and output throughput from the remaining bandwidth (lines 21-22), representing the additional load that remaining replicas would need to absorb. Similarly, it calculates the projected average CPU usage by dividing the total CPU consumption by the remaining CPU capacity after removing the candidate's allocation (line 23).

The removal is considered safe only when three conditions are satisfied. The projected remaining input bandwidth must exceed the safety threshold (line 24), the projected remaining output bandwidth must exceed the safety threshold (line 25), and the projected CPU usage must stay below the safety threshold (line 26). When these requirements are met, the replica is added to the candidate list with its metrics (lines 26-28).

After evaluating all underutilized operators and their replicas, the algorithm returns the set of safe removal candidates (line 32). Algorithm 2.10 can then score and prioritize them for removal.

Algorithm 2.10 presents a scoring-based approach for selecting which replica to remove from a set of underutilized candidates. It takes a list of removal candidates as input, along with their metrics and predefined weights for CPU, input bandwidth, and output bandwidth. It outputs the replica that is most suitable for removal.

Algorithm 2.10 Replica Selection for Removal

**Input:** candidates (map of removal candidates with metrics);
metric weights $w_{\text{CPU}}$, $w_{\text{inBW}}$, $w_{\text{outBW}}$;
**Output:** replicaToRemove (selected replica for removal);

```
   // Find max values for normalization
1  CPU_max ← findMax(candidates, CPU);
2  inBW_max ← findMax(candidates, remInBW);
3  outBW_max ← findMax(candidates, remOutBW);
   // Score each candidate
4  foreach c in candidates do
5  |  inBW_norm ← c.remInBW / inBW_max
6  |  outBW_norm ← c.remOutBW / outBW_max
7  |  CPU_norm ← 1 – (c.CPU / CPU_max)
8  |  candidates[c].score ← (w_inBW × inBW_norm) + (w_outBW × outBW_norm) + (w_CPU × CPU_norm)
9  end foreach
   // Select candidate with highest score
10 replicaToRemove ← selectMaxScore(candidates)
11 return replicaToRemove
```

The algorithm begins by identifying the maximum values for each metric across all candidates (lines 1-3). These maximum values are essential for normalization, ensuring that all metrics are scaled to comparable ranges before scoring.

For every candidate $c$ in the collection (line 4), the algorithm first normalizes its metrics. The input bandwidth is normalized by dividing the candidate's remaining input bandwidth by the maximum input bandwidth found earlier (line 5). Similarly, the output bandwidth is normalized in line 6. The CPU normalization in line 7 follows a different pattern. It subtracts the normalized CPU value from 1, effectively inverting the metric so that lower CPU usage results in a higher normalized value. This inversion is crucial because replicas with lower CPU usage are more suitable candidates for removal.

After normalizing all three metrics, line 8 computes a weighted score for each candidate. This score is stored in the candidates list.

After all candidates have been scored, the algorithm selects the candidate with the highest score as the replica to remove (line 10). This selection is performed by the `selectMaxScore` function, which returns the candidate with the maximum score value. The selected replica is then returned as the algorithm's output (line 11).

#### 2.5.2.4 Replica Removal Process

Once the system has determined that a replica needs to be removed and identified which replica to remove through the scoring mechanism described in Algorithm 2.10, it follows sequential steps to ensure a smooth transition.

Algorithm 2.11 Replica Removal Process

**Input:** Topology structure $\mathcal{T}$ (operator connections);
$r_{\text{toRemove}}$ (replica to be deleted);
**Output:** Replica removed;

```
1 upstreams ← findUpstreamOps(r_toRemove);
2 foreach us in upstreams do
3 |   notify(us, r_toRemove);
4 end foreach
  // Signal replica to stop
5 sendMQTT(r_toRemove, "stop");
6 waitForAcknowledge(r_toRemove);
7 update(T);
```

The process begins by identifying all upstream operators that send data to the replica being removed (line 1). For each upstream operator (lines 2-4), the Orchestrator sends a notification containing the replica name (line 3) so they can stop sending data to this replica. When the upstream operators receive the notification, they remove the replica from their list of downstream operators and trigger a local adaptation process to redistribute the data more efficiently among the remaining downstream operators. Moreover, they unsubscribe from the replica's metrics and close any sockets that are used to transfer data to it.

After the upstream operators are updated, the Orchestrator signals the replica to stop by sending an MQTT message with the topic "stop" (line 5) and waits for its confirmation before continuing

(line 6). Upon receiving the stop notification, the replica must revert back to being an idle generic bolt so that it can become a replica of another bolt later if needed. To do this, it removes all references to its downstream operators, unsubscribes from their metrics, and closes the sockets used for communication. Following that, it sends the Orchestrator the confirmation message.

Once the acknowledgment is received from the removed replica, the Orchestrator updates the topology (line 7) to reflect the latest changes, ensuring that all system components are synchronized with the new configuration.

# CHAPTER 3

# IMPLEMENTATION

This chapter provides details of system implementation. After explaining the tools, languages, and libraries used for development, we will take a close look at the system configuration and deployment over Apache Storm cluster. The chapter concludes with a technical overview of the main system components and their interactions.

## 3.1 Programming Languages, Libraries and Tools

In this section, we briefly discuss the programming languages and libraries used for the development of our system.

### 3.1.1 Java

Since our approach relies on Apache Storm, all enhancements and extensions are provided in Java to make it more flexible. Java is the programming language used for developing data processing pipelines and further extensions to ensure easy integration with Apache Storm. The Orchestrator is also implemented in Java.

### 3.1.2 Eclipse Paho MQTT

The Eclipse Paho MQTT is a client-side implementation of the MQTT protocol. We use this library in both the Storm topology and the Orchestrator to provide real-time message exchange between system components. It makes it possible for various components of our system to establish and maintain connections with an MQTT broker, sends out messages, and subscribe to relevant topics.

### 3.1.3 Maven

We use Maven for project dependency management and to compile both the Storm and Orchestrator projects, resulting in a corresponding JAR file for each project, which will later be used for deployment. The topology JAR is submitted to the Storm cluster for execution, while the Orchestrator JAR is launched separately to coordinate the system.

## 3.2 Deployment and Configuration

This section presents the deployment tools employed and details the configuration of the cluster environment.

### 3.2.1 Docker

One of the key technologies we used to simplify the deployment process is Docker. Docker is an open platform for developing, packaging, and running applications in an isolated way (Docker (2025a)). It allows applications, along with their dependencies, libraries, and configurations, to be packaged into a single unit called a Docker image. In other words, a Docker image can be considered as a snapshot of the application together with its entire runtime environment.

Docker images are created using Dockerfiles, which are scripts containing a sequence of instructions needed to set up the environment. These instructions typically include installing an operating system, adding libraries, copying files, and applying the necessary configurations.

Once built, Docker images can be stored and distributed through repositories such as Docker Hub or private registries. These repositories provide a centralized and reliable mechanism for sharing and maintaining images across multiple machines.

In a distributed environment, Docker ensures that applications execute consistently across all nodes in the cluster. Since each Docker image contains the application together with its required dependencies and configurations, differences in underlying operating systems, library versions, or

host-level settings do not affect execution. This capability reduces deployment errors, simplifies system administration, and provides a reliable foundation for large-scale distributed processing.

In our system, instead of manually installing Apache Storm and its dependencies (such as ZooKeeper) on each node, we built a customized Docker image of Apache Storm. This image also includes the topology JAR file along with the required configuration files, ensuring that it is readily available for deployment on all nodes. To maintain consistency and simplify distribution, we uploaded our Storm image to our repository on Docker Hub. With this approach, nodes can simply pull the image and deploy it with minimal effort.

### 3.2.2 Docker Swarm

To launch Apache Storm with all the necessary configurations and dependencies on a single machine, we can run a Docker image, which then executes as a container. In a distributed environment, where Apache Storm must run across multiple nodes, a container needs to be created and maintained on each machine. Managing these containers individually while also setting up and monitoring the network between them can quickly become time-consuming and prone to errors if done manually.

To reduce manual intervention and minimize the risk of errors, we utilize Docker Swarm, a platform for orchestrating and managing containers across multiple machines in a cluster (Docker (2025b)). Docker Swarm allows us to treat the cluster of nodes as a single virtual system, making it possible to deploy and manage services in a unified way.

A key advantage of Docker Swarm is its ability to work with a docker-compose.yml file. This file provides a declarative way to configure and run containers by specifying details such as the number of replicas for each service, placement rules for services, and resource allocations like CPU and memory. With this approach, services can be scaled, placed, and managed consistently without extensive manual setup.

In our system, Docker Swarm is responsible for managing several critical components, including ZooKeeper, Nimbus, Supervisors, and Mosquitto as an MQTT broker. By automating deployment and resource allocation, Docker Swarm ensures that these components run reliably across the cluster while improving scalability and overall system stability.

## 3.3 Implementation Class Hierarchy

Figure 3.1 shows the class hierarchy designed for the Storm pipeline implementation. The structure follows an object-oriented design that extends Apache Storm's original interfaces (`IRichBolt` and `IRichSpout`) while adding customized components to support the elasticity layer introduced in this work.

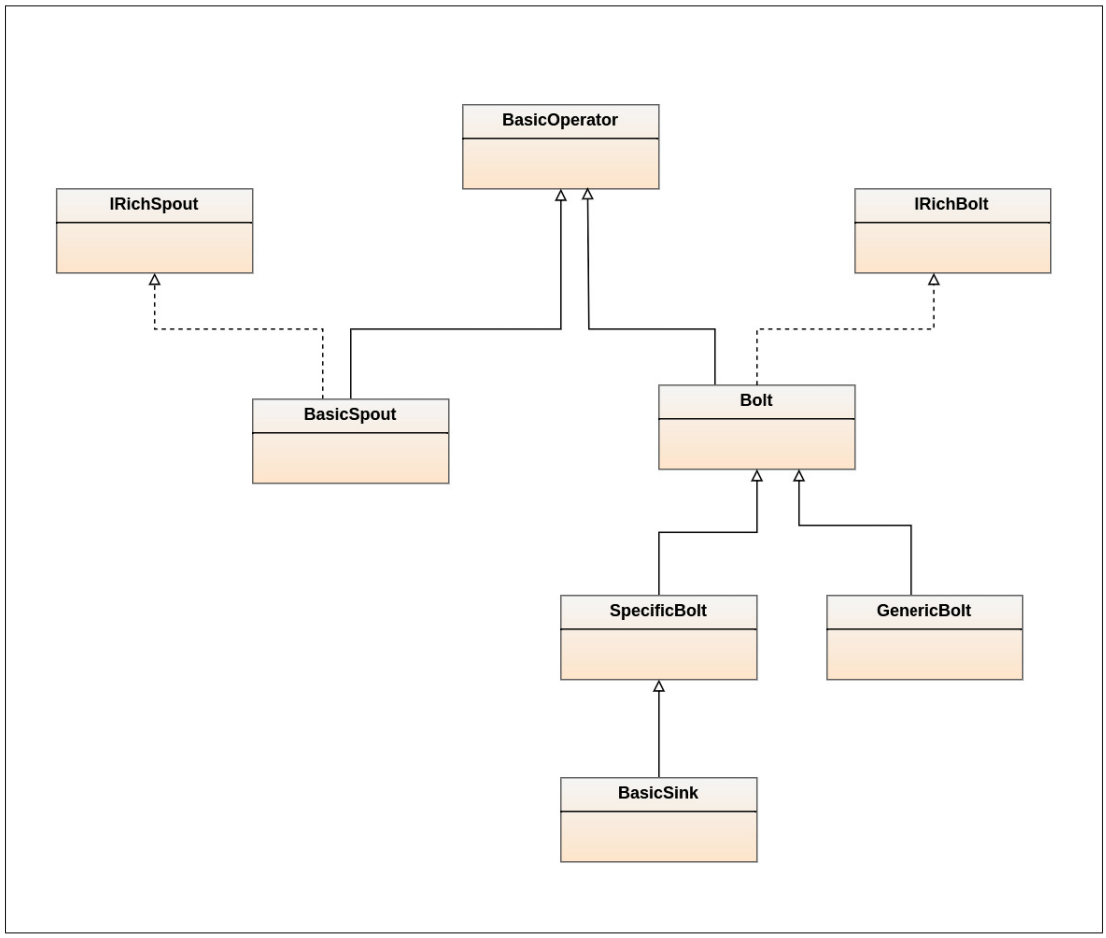



Figure 3.1 Class hierarchy of the Storm pipeline

At the top of the hierarchy, the `BasicOperator` class defines common attributes and methods shared by all operators, such as metric collection, communication handling, and initialization logic. Two subclasses, `BasicSpout` and `Bolt`, inherit from this base class. The `BasicSpout` implements Storm's `IRichSpout` interface, while the `Bolt` class implements `IRichBolt` and provides the foundation for more specialized operators.

The `SpecificBolt` class represents standard operators that execute the user-defined processing logic. In contrast, the `GenericBolt` class is designed to serve as an idle or potential replica, which can dynamically load the behavior of other bolts when replication is triggered. The `BasicSink` extends `SpecificBolt` and serves as the final component that receives and stores processed data.

Application-specific operators, such as custom spouts, sinks, and bolts, must extend `BasicSpout`, `SpecificBolt`, and `BasicSink`, respectively. This design ensures seamless integration between application-level logic and the elasticity layer, allowing developers to implement new processing elements without altering the core framework.

In theory, existing Storm topologies could be executed within this extended framework without major structural modifications, while benefiting from enhanced elasticity and resource management capabilities. Since the system is built as an extension of Apache Storm, all components remain compatible with the native Storm runtime.

# CHAPTER 4

# EVALUATION

This chapter evaluates the proposed solution for elastic stream processing built on Apache Storm, designed for heterogeneous edge deployments. The method integrates global and local adaptation to create or remove replicas as needed and to distribute load effectively. These mechanisms proactively prevent and effectively address CPU and bandwidth bottlenecks to ensure low latency and efficient resource usage.

The evaluation is structured around four experimental scenarios, each designed to isolate and highlight different aspects of system performance. Through these scenarios, we aim to provide empirical evidence that directly addresses the research questions discussed in Section 2.1.

The chapter begins with a description of the evaluation setup, followed by the presentation of the scenarios and their results.

## 4.1 Experimental Setup

The experimental evaluation was conducted on a distributed testbed consisting of 21 virtual machines (VMs). In this setup, each VM is treated as a node in the cluster, with Docker installed locally and the entire cluster managed through Docker Swarm for container orchestration. This setup allows deployment and managing all containers in a coordinated way across the different nodes. On each node, a single container was deployed and each container was restricted to a fixed number of CPU cores and a limited amount of memory. These resource limits were applied intentionally to reflect the kinds of constraints that exist in real edge environments.

The network between the nodes was also part of the setup. We measured the latency between each pair of nodes using `ping` and used those values as the node-to-node latency in the experiments. Since the `ping` command reports round-trip latency, we divided the value by two to estimate one-way latency. This one-way latency was then assigned to the corresponding node pair, so that regardless of which node was the source or destination, the same value was applied. A

complete table of measured latencies for all node pairs is provided in Appendix I. In addition, we configured the network so that each container had a maximum allowed inbound bandwidth and outbound bandwidth.

This setup was designed to model an edge environment in a realistic manner. By applying strict limits on CPU, memory, and network bandwidth, the containers mimicked the resource-constrained nature of real edge devices. Using Docker containers also ensured process isolation, meaning that no single process could consume all the resources of a machine.

This configuration made it possible to study how Apache Storm responds when resources become limited, particularly when congestion occurs in terms of CPU usage, inbound bandwidth, or outbound bandwidth, and how these conditions affect the end-to-end latency of the data.

## 4.2 Baselines

This section describes the proposed approach and the other baselines used for comparison in the evaluation.

The proposed approach relies on two complementary mechanisms, which are described below:

- Global adaptation: Global adaptation is responsible for deciding when and where new replicas should be created, as well as when and where replicas can be safely removed without harming performance. Further details on this point can be found in Section 2.5.2.
- Local adaptation: Local adaptation enables upstream operators to perform a resource-aware data routing to balance load among downstream replicas. For a complete description of this process, see Section 2.5.1.

Together, these mechanisms provide elasticity and load balancing at the operator level. This combined solution is referred to as the *heuristic* approach throughout the remainder of this chapter.

To evaluate the effectiveness of the *heuristic* approach, we compared it against several baselines. Each baseline isolates or simplifies certain aspects of the system, allowing us to highlight the

individual contributions of global and local adaptation, as well as to better understand the limitations of existing strategies. The following baselines were considered:

- **GlobalAdaptationOnly**: In this baseline, only the global adaptation mechanism is active. The topology is launched with no replicas at the start (initial number of replicas = 0). Replicas can later be created or removed by this method, but the load among replicas is distributed equally. For example, with three replicas of a bolt, each replica always receives one-third of the load. This baseline is included to highlight the importance of the local adaptation mechanism.
- **Storm_replication**: This baseline mimics the behavior of Storm's default parallelism mechanism with manual scaling capabilities disabled (Apache Software Foundation (2025)). In other words, the system cannot rebalance task placement during runtime unless manual intervention occurs. Also, it cannot change the level of parallelism once the topology is launched. The number of replicas is statically defined in the code before execution and remains fixed throughout the run. Once the topology starts, all replicas of an operator are active from the beginning, each handling part of the incoming data. The system does not automatically create additional replicas when the load increases and does not remove replicas when the workload decreases. Executors are assigned to available slots, and load is distributed equally among downstream replicas without any intelligent routing or resource awareness. This baseline illustrates the replication mechanism natively used in Storm.
- **NoLocal_noGlobal** In this baseline, both global adaptation and local adaptation are disabled. The topology runs without any replica management. Operators remain fixed, and no new replicas are created during execution. This baseline is included to show the behavior of the system when no elasticity mechanisms are applied.

It is important to note that this work is not directly compared with other studies in the literature. The main reason is the lack of prior approaches that target the same environment and objectives. Most existing works on elasticity in stream processing assume homogeneous clusters rather than heterogeneous edge deployments, which makes their results difficult to align with the challenges addressed here. In addition, the metrics evaluated in this study, particularly node CPU usage,

node bandwidth availability, and intra-node latency, are rarely considered together in related works. Finally, to the best of our knowledge, no existing study has explored operator-level replication in Apache Storm for edge scenarios, which makes a direct baseline comparison infeasible.

The following section introduces the experimental scenarios that were designed to evaluate the system.

## 4.3 Experiments

In order to evaluate the system, four experimental scenarios were designed. The first three scenarios cover bottleneck cases, where the system faces bandwidth limitations, CPU limitations, both separately or combined. Together, they provide the basis for answering Research Questions 1 to 4 concerning scalability and elasticity, congestion management, load balancing, and system responsiveness. The fourth scenario focuses on scalability and is intended to address Research Question 5.

To emulate the heterogeneity of edge devices, resource limits were configured for the containers and network connections. CPU and memory constraints were applied through Docker, while bandwidth limitations were controlled externally using traffic shaping scripts. This setup allows the available capacity of each device to differ across scenarios, making it possible to design experiments in a controlled way to trigger specific bottlenecks.

Since bandwidth plays a key role in several scenarios, it is important to clarify how communication between operators is handled. When two operators are deployed on different nodes, transferring tuples requires network communication. In this case, the outgoing data stream consumes the out-bandwidth of the sending node and the in-bandwidth of the receiving node. If the operators are located on the same node, this transfer occurs locally and does not consume network bandwidth. This mechanism is common across all scenarios and is therefore described here once for clarity.

To ensure fair comparison across baselines, a custom scheduler was implemented for the initial placement of operators. For each scenario, a mapping file specifies where the original operators of the topology are deployed. This mapping is applied consistently across all baselines within the same scenario, while different scenarios may use different mappings.

In all experiments, the ZooKeeper, Nimbus, and UI services are always deployed on Node 01. This configuration remains identical across all experiments to avoid introducing variability from system management components. In addition, the spouts and sinks are placed on a dedicated node (Node 02) and are not the subject of study, since the focus is on the behavior of bolts. The nodes hosting spouts and sinks are provisioned with high capacity to ensure they do not become limiting factors, allowing the evaluation to isolate the impact of bolts on overall performance.

To assess performance under these conditions, three node-level metrics are collected in all scenarios. These include in-bandwidth, out-bandwidth, and CPU usage, and their collection process is explained in Section 2.4. In addition, end-to-end latency is measured, which is defined as the time from when a tuple is emitted by a spout to when it is received by a sink, and it includes both processing and transmission delays. Each tuple is assigned a unique identifier. When the spout generates a tuple, the system time is recorded immediately before the tuple is emitted and sent as an MQTT message to the Orchestrator. Similarly, when the sink receives a tuple, the system time is captured just before storing the result and forwarded to the Orchestrator. Since spout and sink are deployed on the same machine, there is no concern about clock synchronization. The Orchestrator later integrates these records to compute the difference between emission and reception times, which yields the end-to-end latency. These metrics provide the foundation for evaluating system performance and are used as decision inputs for the adaptation mechanisms.

What follows is a detailed description of the four scenarios, along with the experimental design and the topology used.

### 4.3.1 Scenario 1: Bandwidth Bottleneck

The first scenario examines system elasticity where the incoming and outgoing bandwidth of the edge processing nodes is constrained. Table 4.1 presents the CPU and bandwidth resources assigned to each node, highlighting the heterogeneity of the nodes in this setup. We begin by

Table 4.1 In-bandwidth and Out-bandwidth resources assigned to each VM in Scenario 1

| VM ID | CPU Cores | In-bandwidth (Mbit/s) | Out-bandwidth (Mbit/s) |
|---|---|---|---|
| Node 02 | 6 | 40 | 70 |
| Node 03 | 4 | 6 | 5 |
| Node 04 | 4 | 6 | 3 |
| Node 05 | 5 | 4 | 3 |
| Node 06 | 4 | 4 | 3 |
| Node 07 | 2.5 | 3 | 4 |
| Node 08 | 2 | 3 | 4 |
| Node 09 | 2.5 | 4 | 7 |
| Node 10 | 2.7 | 6 | 6 |
| Node 11 | 3.5 | 5 | 8 |
| Node 12 | 3 | 7 | 6 |
| Node 13 | 3.5 | 7 | 6 |
| Node 14 | 3 | 6 | 8 |
| Node 15 | 3.5 | 8 | 7 |
| Node 16 | 3 | 7 | 6 |
| Node 17 | 4 | 16 | 14 |
| Node 18 | 4 | 14 | 16 |
| Node 19 | 4 | 12 | 14 |
| Node 20 | 4 | 12 | 15 |
| Node 21 | 4.5 | 13 | 16 |

describing the plots that summarize the system behavior in this scenario. These descriptions are then followed by a more detailed performance analysis.

#### 4.3.1.1 Description of Results

The topology used is an image processing pipeline with four sequential bolts performing resize, blur, brightness adjustment, and watermarking. This linear pipeline maintains a one-to-one

mapping between input and output, which means for every input, the system produces a single corresponding output. Figure 4.1 shows the topology used in this experiment.

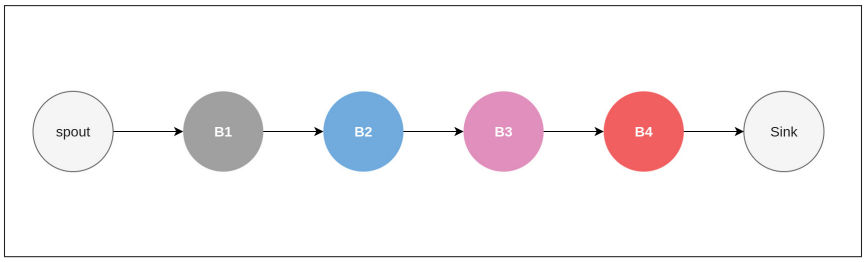



Figure 4.1 Stream processing topology used for the bandwidth bottleneck experiment (Scenario 1)

To evaluate system performance, the available resources of each node (CPU cores, in-bandwidth, and out-bandwidth) are continuously monitored. In addition, different incoming data rates are applied during execution to stress the system under varying load conditions.

#### 4.3.1.1.1 CPU Utilization

Figures 4.2 to 4.5 report CPU utilization across nodes over time for the four baselines. Each row represents a worker node (Node 03–21) and each column corresponds to a execution cycle, with one cycle equal to 30 seconds in this experiment. The color scale indicates the level of utilization: values between 0–80 % are considered safe (green), values between 80–90 % indicate that the node is approaching saturation (yellow–orange), and values between 90–100 % are classified as critical (red), meaning that the node is saturated.

To make operator placement visible in the CPU utilization plots, each operator (B1–B4) is represented using the same distinct colors that were introduced in the topology diagram (Figure 4.1). The CPU plots include vertical lines in operator-specific colors, which show precisely when an operator is activated or deactivated on a given node. Two types of markers are used: a solid vertical line denotes the start of execution of a given operator on a particular node, while a dashed vertical line denotes the end of its execution. With this convention, it becomes

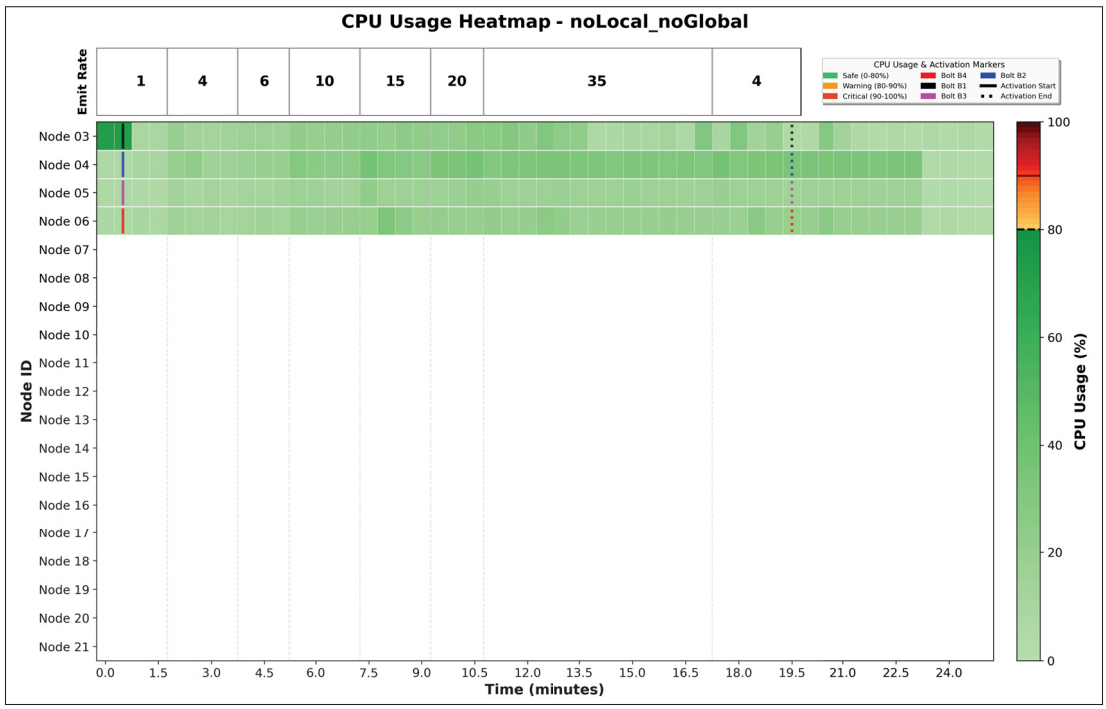



Figure 4.2 CPU utilization across nodes over time in Scenario 1 for the noLocal_noGlobal baseline

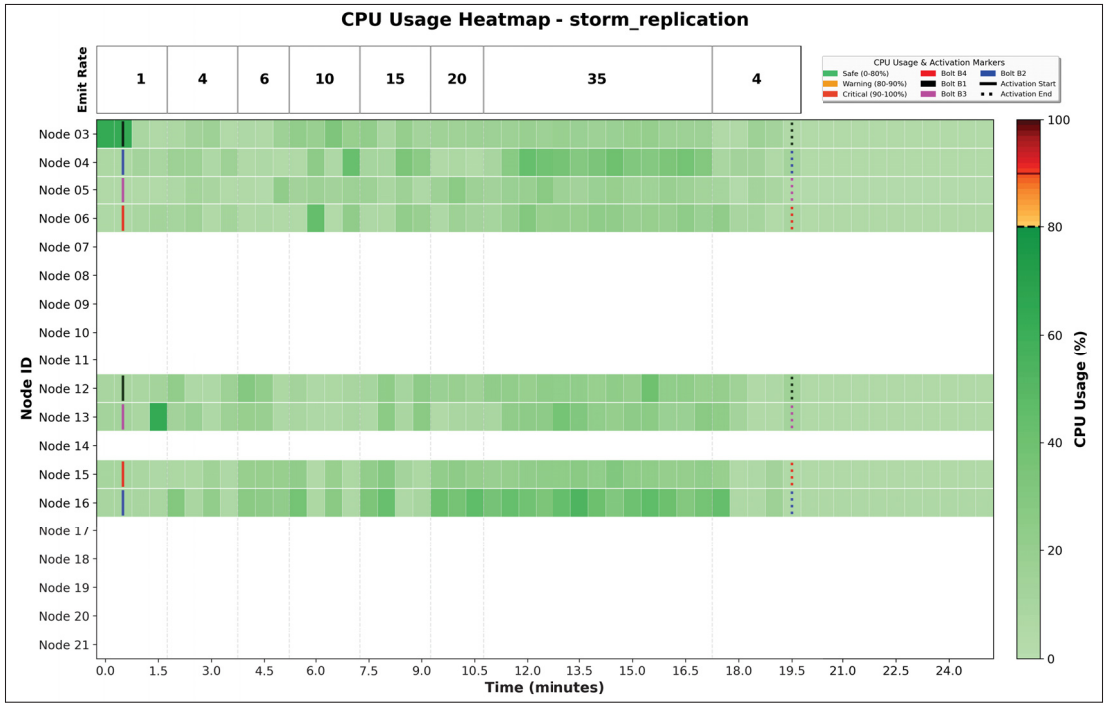



Figure 4.3 CPU utilization across nodes over time in Scenario 1 for the storm_replication baseline

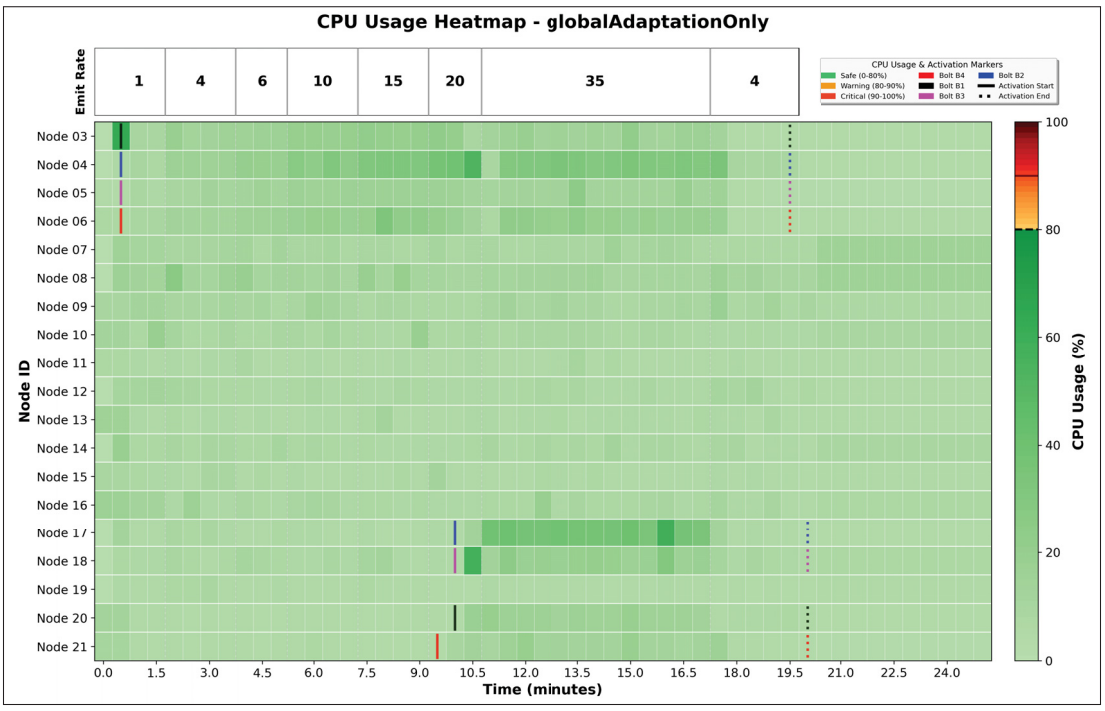



Figure 4.4 CPU utilization across nodes over time in Scenario 1 for the globalAdaptationOnly baseline

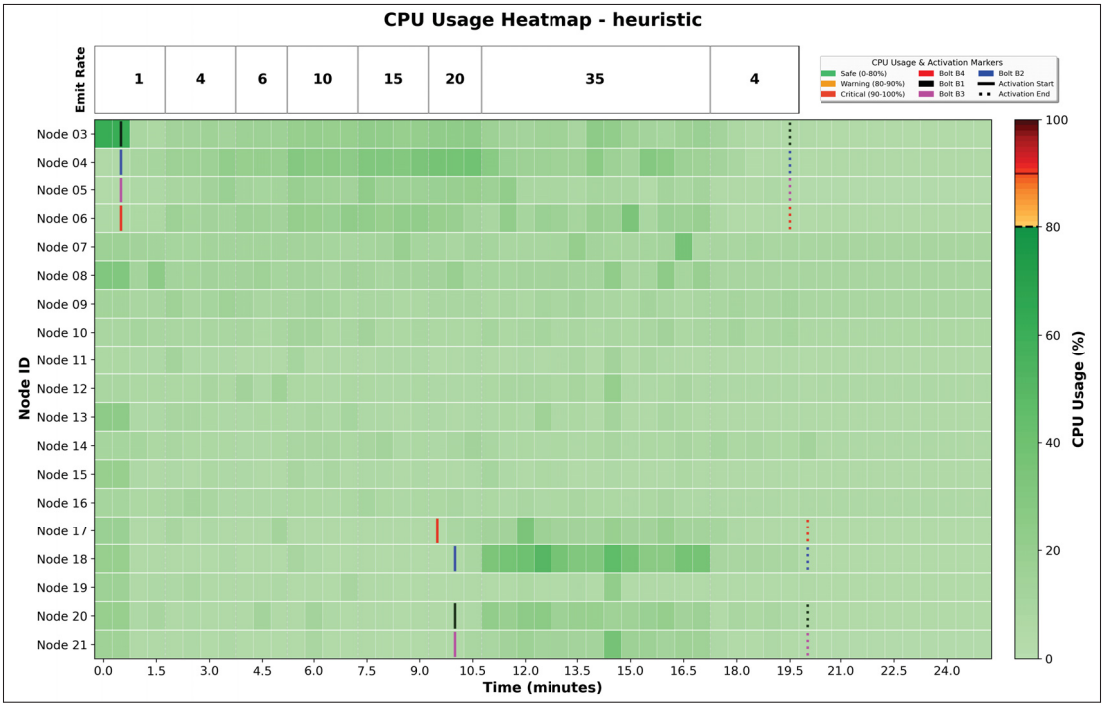



Figure 4.5 CPU utilization across nodes over time in Scenario 1 for the Heuristic approach

straightforward to identify which operator is running on each node at any given time, and to follow activation and deactivation events throughout the experiment.

At the top of each figure, the emit rate is shown in terms of the number of images per second entering the system through the spout. In this experiment, the input rate was gradually increased from 1 image/s to 35 images/s and then decreased back to 4 image/s , before the input load was completely terminated.

The CPU utilization plots for all baselines show that nodes consistently operated below the critical threshold and remained in safe or moderate ranges throughout the experiments. This confirms that the workload is not CPU-intensive and that available processing capacity is sufficient to prevent congestion.

The evaluation of CPU usage offers only a partial view of system performance. To obtain a more complete picture, we next analyze bandwidth usage in this experiment.

#### 4.3.1.1.2 Bandwidth Utilization

Figure 4.6 shows bandwidth utilization results for the four baselines. In each case, two subplots are provided, with in-bandwidth usage shown on the left and out-bandwidth usage shown on the right. The y-axis represents bandwidth utilization expressed as a percentage of the total capacity, while the x-axis corresponds to execution cycles (30 seconds per cycle). For each subplot, the shaded area in the foreground spans the minimum and maximum bandwidth usage observed across all worker nodes in the cluster, providing a view of the overall range of bandwidth consumption.

The red dashed horizontal line indicates the 80% threshold, which was empirically observed to be the point where bandwidth saturation begins to cause noticeable increases in end-to-end latency.

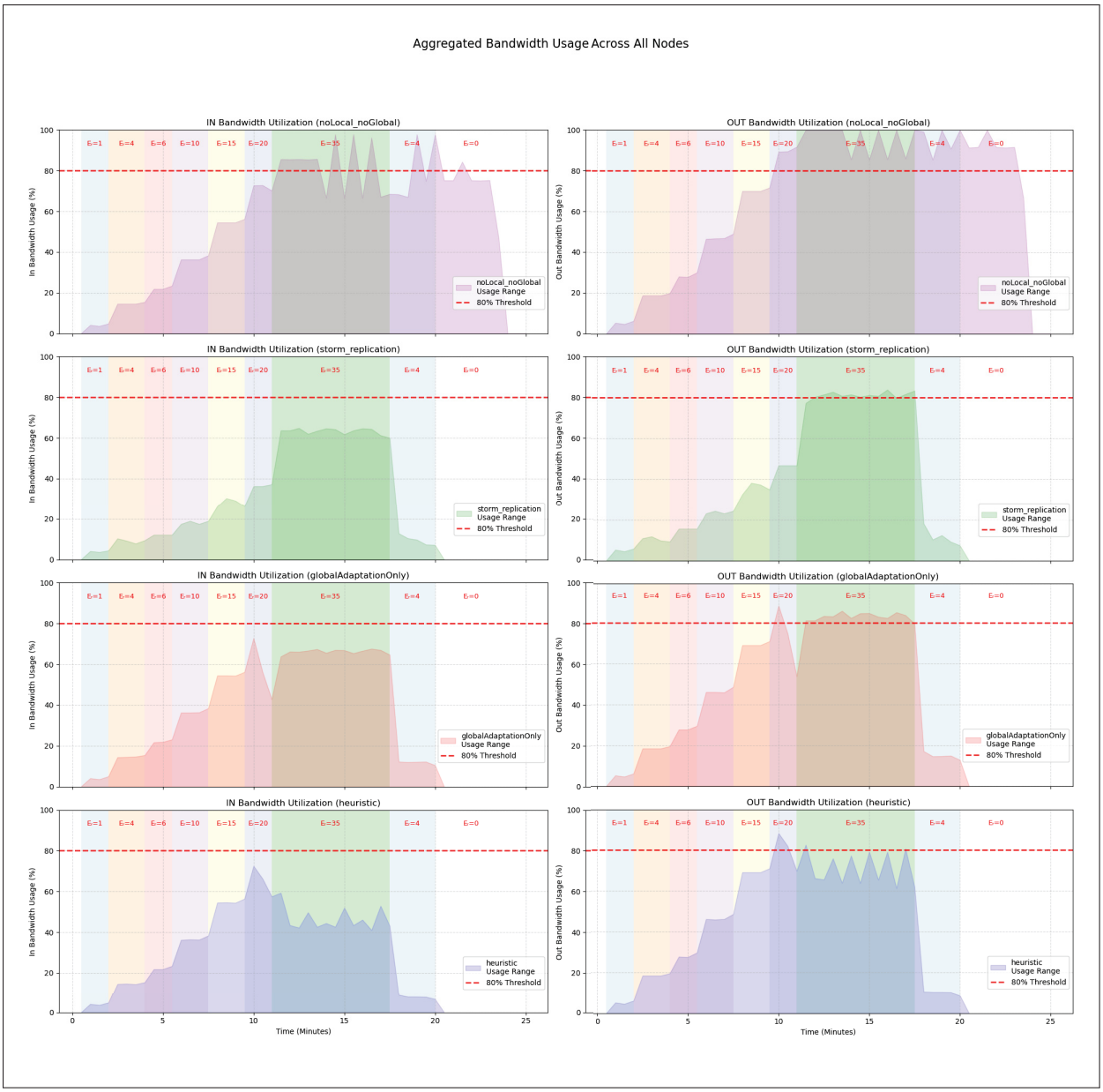



Figure 4.6 Aggregated in-bandwidth (left) and out-bandwidth (right) utilization across all nodes in Scenario 1 for the four baselines. The shaded regions indicate usage over time at different incoming rates, with the red dashed line marking the 80% threshold

The background shading in each subplot marks the different workload phases, and the label $E_r$ indicates the tuple generation rate in images per second (images/s) entering the system. This plot design highlights the relationship between workload intensity and network usage.

The main purpose of this figure is to show the number of cycles during which at least one node reaches saturation. This is particularly important in distributed stream processing, where congestion or delay on a single node can directly impact the overall end-to-end latency. At the same time, the minimum values show that other nodes may be lightly loaded or even idle, which reflects the uneven distribution of bandwidth usage across the cluster.

In the *globalAdaptationOnly*, the system resolves outgoing congestion when the emit rate is 20 images/s, but when the emit rate reaches 35 images/s, a continuous out-bandwidth bottleneck appears and persists until the input load decreases. In the *heuristic* approach, two out-bandwidth congestions are observed. One appears at 20 images/s , which is resolved quickly, and the second at 35 images/s, which is addressed in the following cycle. Afterward, no further bottlenecks occur. In the *noLocal_noGlobal* baseline, multiple in-bandwidth and out-bandwidth congestions are observed, and once the emit rate reaches 20 images/s, the out-bandwidth congestion persists for the rest of the run. Finally, in the *storm_replication* case, the outgoing bandwidth stays at the 80% threshold during peak input rates and remains at that level for the entire high-load phase.

To understand how each baseline responds to these bandwidth pressures, we next examine the number of replicas created and removed over time.

#### 4.3.1.1.3 Replica Count

Figure 4.7 shows the total number of replicas over time for all baselines. The x-axis represents the execution cycles (30 seconds per cycle), and the y-axis shows the number of active replicas in the system. Here, active replicas refer to generic bolts that become activated dynamically during execution and are distinct from the original operators specified in the topology.

In the *noLocal_noGlobal* baseline, the replica count remains fixed at zero throughout the experiment. In the *storm_replication* baseline, four replicas are defined in the code prior to execution and remain constant during runtime. In *GlobalAdaptationOnly* and *heuristic*, the initial replica count is zero, and replicas are created or removed dynamically based on global adaptation policies.

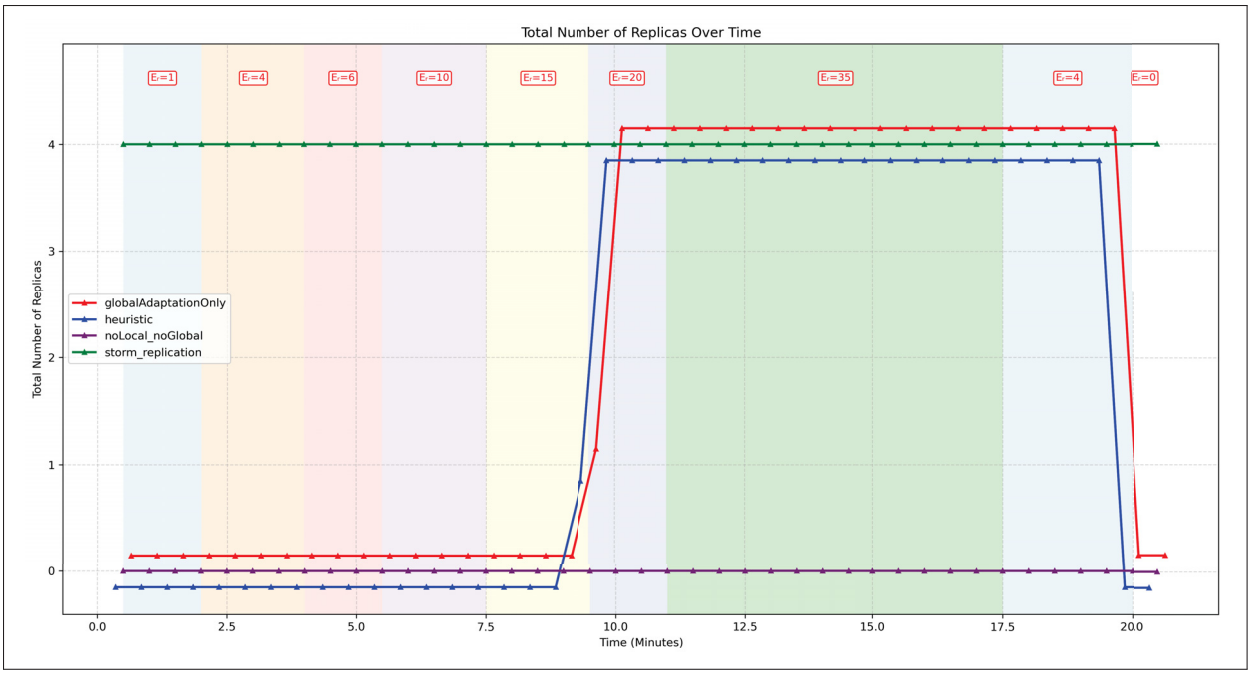



Figure 4.7 Total number of replicas over time in Scenario 1 for the four baselines. The shaded regions correspond to different incoming rates during execution

To understand how different metrics shape system behavior and how congestion and scaling ultimately influence end-to-end latency, we now look into latency results in the following section.

#### 4.3.1.1.4 End-to-End Latency

The average end-to-end latency of tuples per cycle for the four baselines is depicted in Figure 4.8. The x-axis represents execution cycles and the y-axis shows the mean latency of tuples emitted during each cycle, measured in milliseconds. Each shaded block in the background represents a distinct phase of the input load, enabling the reader to follow the workload pattern over time. Labels of the form $E_r$ are placed within the shaded regions to indicate the emission rate, expressed in images per second (images/s), for the corresponding phase.

In the *noLocal_noGlobal* baseline (purple line), latency climbed sharply once the emit rate reaches 35 images/s, peaking above 300,000 ms before decreasing as the input is reduced.

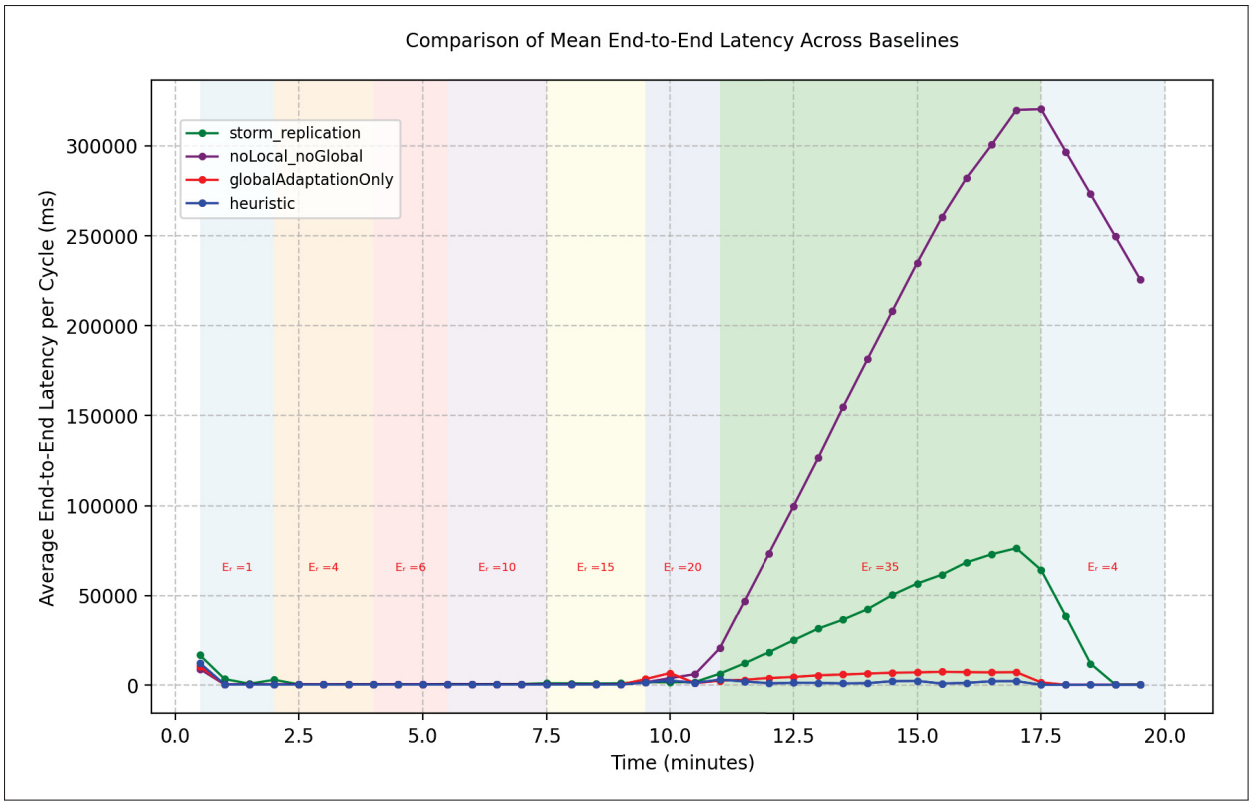



Figure 4.8 Average end-to-end latency of tuples per cycle over time in Scenario 1 for the four baselines. Shaded regions indicate different incoming data rates applied during execution

Latency in the *storm_replication* baseline (green), rises under heavier load, reaching a maximum of approximately 70,000 ms, and then drops once the emit rate decreases. By contrast, both the *globalAdaptationOnly* (red) and *heuristic* (blue) baselines maintain low average end-to-end latency, with *globalAdaptationOnly* consistently higher than *heuristic* during the maximum load period.

#### 4.3.1.2 Analysis

Table 4.2 summarizes the overall performance of the four baselines in terms of end-to-end latency. The input counts are nearly identical across most baselines, and no message loss was observed, which ensures that comparisons are fair. The only exception is the *noLocal_noGlobal* baseline, which shows a slightly lower input count. This reduction is caused by backpressure,

Table 4.2 Summary of input/output statistics and overall mean end-to-end latency for each baseline in Scenario 1

| Approach | Input Count | Output Count | Mean End-to-End Latency (s) |
|---|---|---|---|
| noLocal_noGlobal | 19760 | 19760 | 125.34 |
| storm_replication | 20166 | 20166 | 30.01 |
| globalAdaptationOnly | 20166 | 20166 | 4.28 |
| heuristic | 20166 | 20166 | 1.34 |

as congestion prevents the system from accepting new tuples at the same rate as the other approaches.

When comparing mean execution times, the contrasts between baselines become evident. The *heuristic* method achieves a mean execution time of just 1.34 s, representing a major improvement over all other approaches. The *globalAdaptationOnly* baseline also shows a strong reduction (4.28 s), whereas *Storm Replication* results in a mean execution time of 30.01 s, and *noLocal_noGlobal* shows the poorest performance at 125.34 s.

CPU resources were not the cause of these differences, as utilization levels remained well below saturation during the experiment. Instead, the results are explained by bandwidth bottlenecks, which significantly degrade performance once nodes exceed their in or out-bandwidth capacity. In a stream processing pipeline, congestion at a single node quickly affects the entire system, leading to increasing delays.

The *noLocal_noGlobal* baseline does not create any replicas, which highlights how negatively the absence of elasticity impacts performance. The *storm_replication* baseline, although starting with extra replicas from the beginning, still crosses the saturation threshold. The primary reason is that replicas are placed on poorly suited hosts, and the static nature of this baseline prevents spawning further replicas once the initial set proves unable to absorb the growing load. In contrast, *globalAdaptationOnly* reduces latency by adding replicas when bandwidth becomes constrained. The hosts it selects for replica creation are generally capable of handling the additional load. However, since it distributes traffic equally among replicas without considering

node conditions, it still experiences bottlenecks. The *heuristic* approach achieves the best results, as it combines global decisions about where to place replicas with local load balancing, preventing saturation from persisting on individual nodes.

Overall, the *heuristic* approach demonstrates the best performance, both in average behavior and during peak load, by activating the same number of replicas as other baselines but only when they are truly needed. This efficient use of replication ensures low latency while avoiding unnecessary resource consumption.

### 4.3.2 Scenario 2: CPU Bottleneck

The second scenario examines cases where performance is limited by processing capacity. Nodes were assigned different amounts of CPU cores and bandwidth to reflect heterogeneity as listed in Table 4.3. Bandwidth values were set high enough to prevent network congestion, ensuring that performance differences can be caused by CPU limitation. The pipeline was designed to be CPU-intensive so that computation becomes the main source of pressure.

#### 4.3.2.1 Description of Results

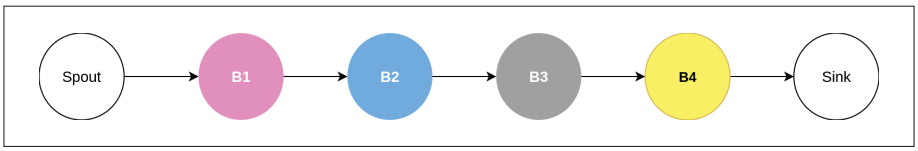



Figure 4.9 Stream processing topology used for the CPU bottleneck experiment (Scenario 2)

In this experiment, the topology is a full linear pipeline (illustrated in Figure 4.9) that produces exactly one output for each input. To simplify interpretation of the results, each bolt is represented with a unique color that is consistent across all plots. This coloring makes it straightforward to follow when and where operators are activated during execution.

The subsequent section examines the overall bandwidth usage within the cluster.

Table 4.3 Bandwidth and CPU resources assigned to each node in Scenario 2

| VM ID | CPU Cores | In-bandwidth (Mbit/s) | Out-bandwidth (Mbit/s) |
|---|---|---|---|
| Node 02 | 6 | 210 | 360 |
| Node 03 | 1.5 | 60 | 35 |
| Node 04 | 1.8 | 40 | 25 |
| Node 05 | 0.9 | 30 | 25 |
| Node 06 | 1.5 | 30 | 25 |
| Node 07 | 1 | 25 | 30 |
| Node 08 | 1.2 | 25 | 30 |
| Node 09 | 1.8 | 30 | 45 |
| Node 10 | 1.5 | 40 | 40 |
| Node 11 | 1 | 35 | 50 |
| Node 12 | 1.6 | 45 | 40 |
| Node 13 | 2 | 45 | 40 |
| Node 14 | 2.2 | 40 | 50 |
| Node 15 | 2.5 | 50 | 45 |
| Node 16 | 2.2 | 45 | 40 |
| Node 17 | 4.5 | 90 | 80 |
| Node 18 | 4 | 80 | 90 |
| Node 19 | 4.2 | 70 | 80 |
| Node 20 | 4 | 70 | 85 |
| Node 21 | 4.5 | 75 | 90 |

#### 4.3.2.1.1 Bandwidth Utilization

In-bandwidth and out-bandwidth utilization for the four baselines in this scenario is shown in Figure 4.10. The left subplot represents in-bandwidth, while the right subplot represents out-bandwidth. The shaded background indicates the incoming data rate in images per second.

From the plots, it can be observed that bandwidth utilization in this scenario remains well below the 80% threshold for all baselines.

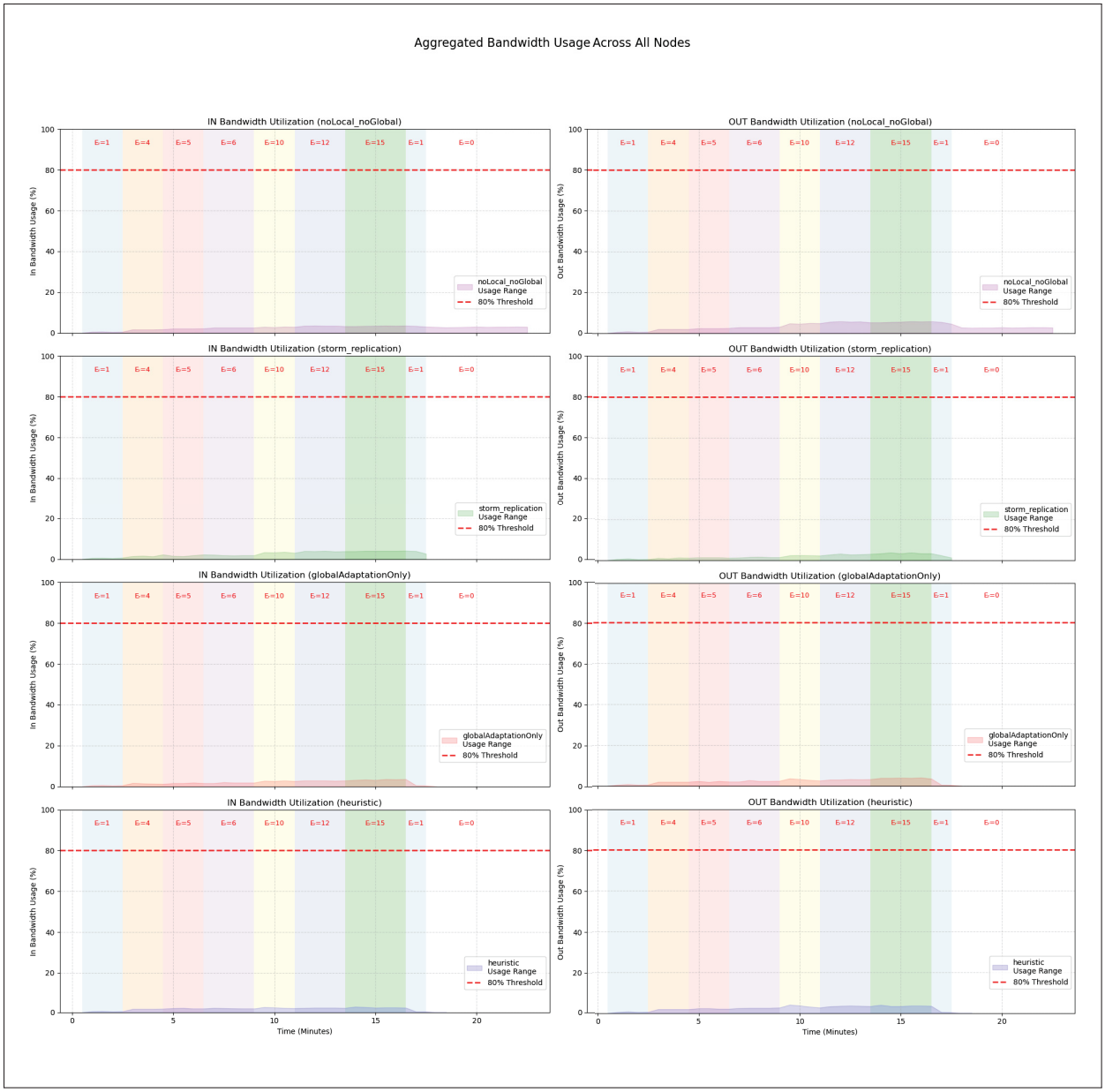



Figure 4.10 Aggregated in-bandwidth (left) and out-bandwidth (right) utilization across all nodes in Scenario 2 for the four baselines. The shaded regions indicate usage over time at different incoming rates, with the red dashed line marking the 80% threshold

#### 4.3.2.1.2 CPU Utilization

Given the CPU-intensive nature of this scenario, we now study how processing resources are consumed across baselines and how effectively each baseline manages the increased CPU demand.

Figures 4.11 to 4.14 show CPU utilization across all nodes for the four baselines. Rows correspond to individual nodes, and color intensity reflects their CPU usage levels. Vertical lines in operator-specific colors, which are consistent with the topology figure (Figure 4.9), indicate when operators are activated or deactivated on each node.

Unlike Scenario 1, where CPU load stayed well below critical thresholds, here multiple nodes reach very high utilization, with some even exceeding saturation. In the *noLocal_noGlobal* baseline, once the emit rate reaches 10 images/s, Node 03 becomes saturated and remains in this state until the end of execution. Additional pressure is also visible on Nodes 04 and 05, with Node 05 remaining in the warning zone for much of the run. In the *storm_replication* baseline, Node 03 similarly becomes saturated at 10 images/s. The congestion eases when the rate increases to 12 images/s, but reappears occasionally at 15 images/s. Moreover, Node 08 experiences a persistent CPU bottleneck once the emit rate surpasses 6 images/s. In comparison, the *globalAdaptationOnly* baseline displays less CPU congestion. Here, most saturation is again concentrated on Node 3, which does not fully recover, but congestion observed on other nodes such as 05, and 06 is mitigated to some extent after only a few cycles. Lastly, the *heuristic* approach exhibits the least frequent and least severe saturation, even during the peak emission rate. When congestion appears, particularly at the peak load, it is alleviated within maximum two cycles, demonstrating a quicker recovery compared to the other baselines.

The following sections on replica counts (Figure 4.15) and latency (Figure 4.16) show how limited processing capacity affects system behavior and how each baseline manages load when CPU resources become the main bottleneck.

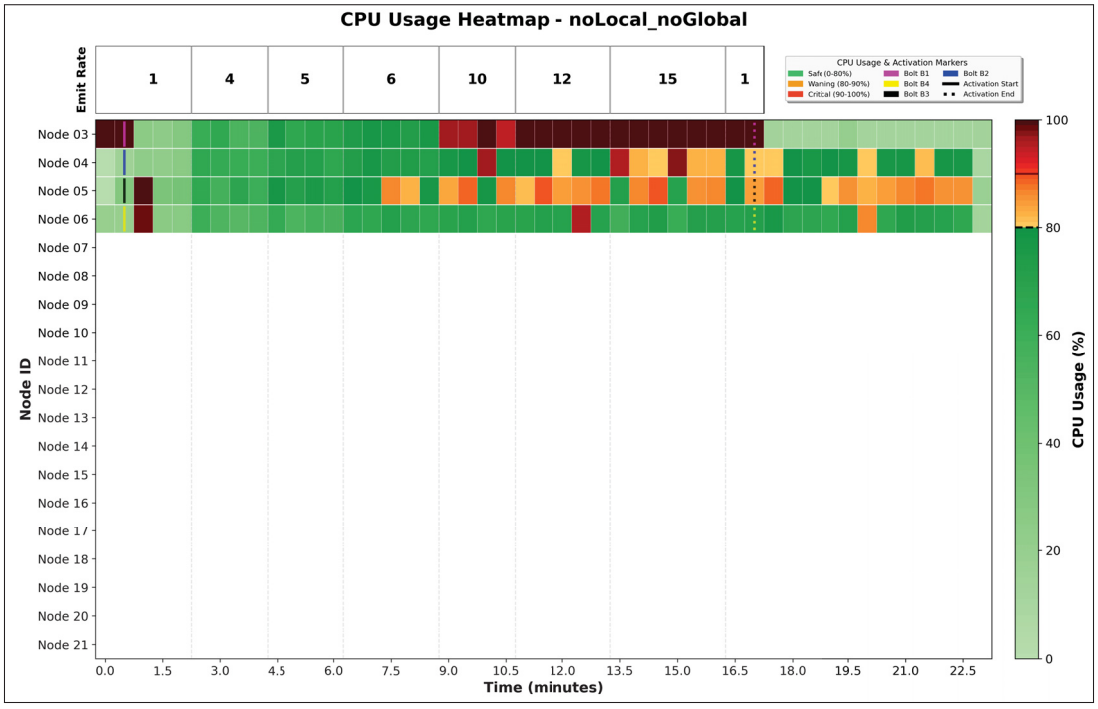



Figure 4.11 CPU utilization across nodes over time in Scenario 2 for the noLocal_noGlobal baseline

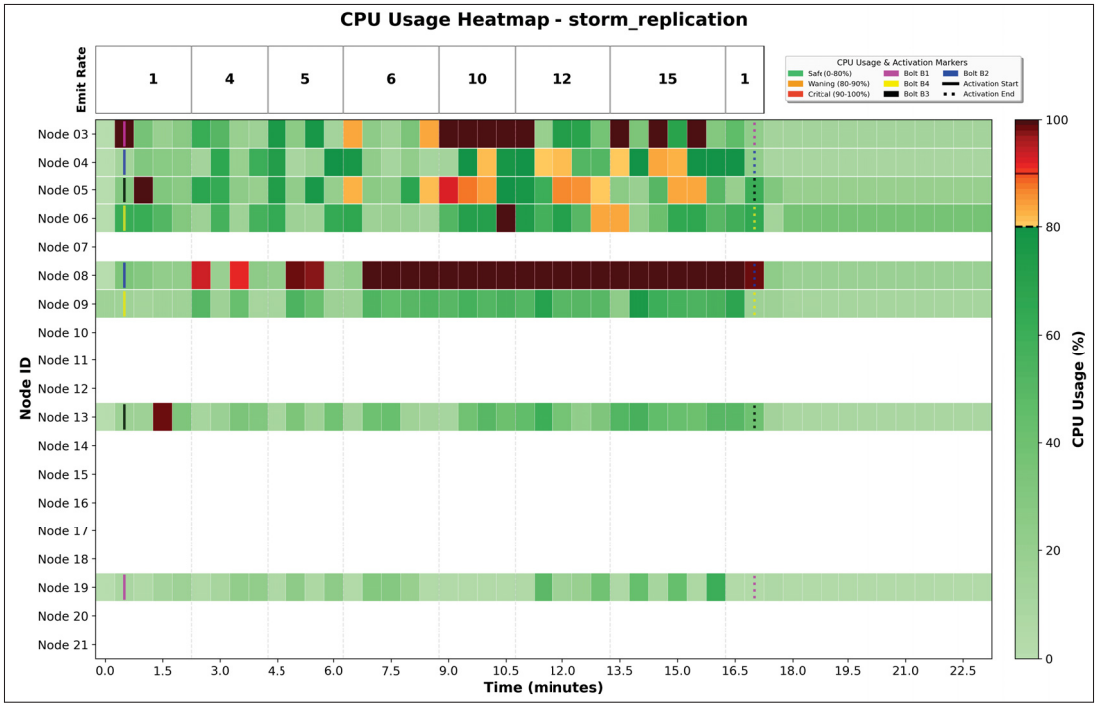



Figure 4.12 CPU utilization across nodes over time in Scenario 2 for the storm_replication baseline

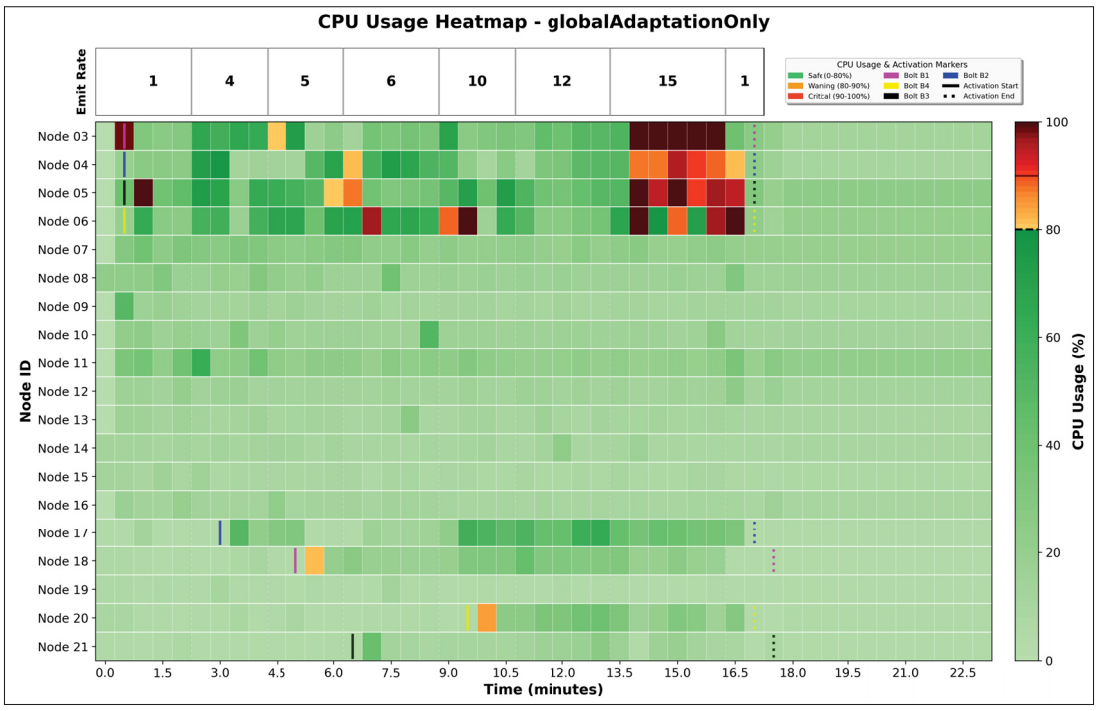



Figure 4.13 CPU utilization across nodes over time in Scenario 2 for the globalAdaptationOnly baseline

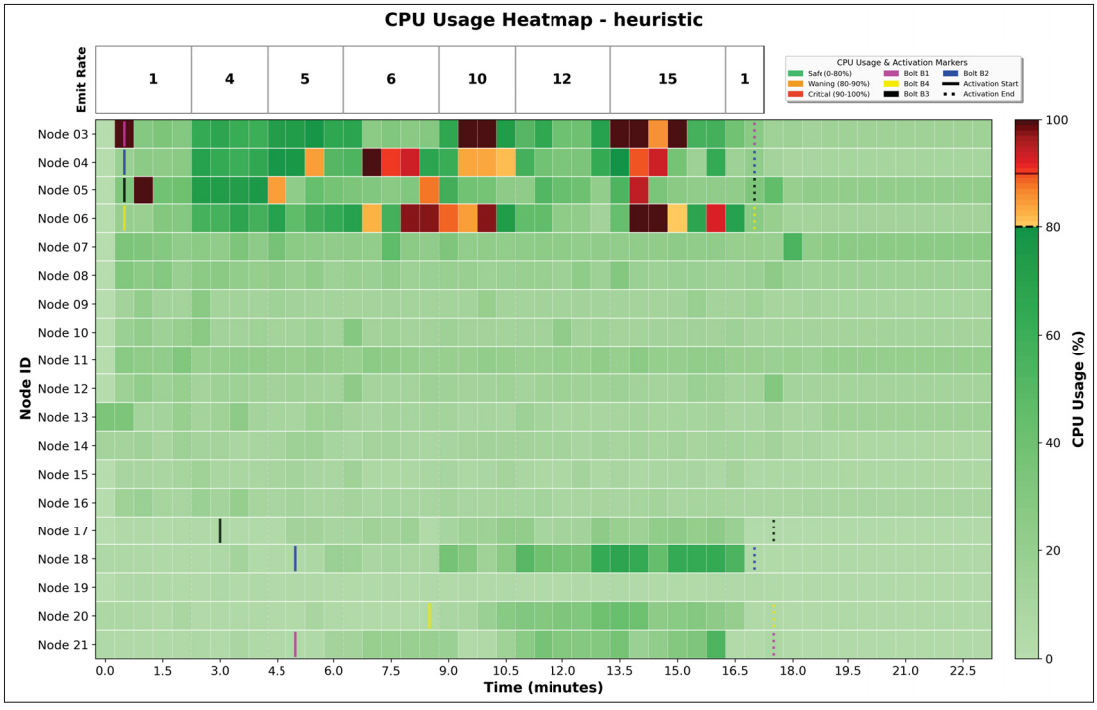



Figure 4.14 CPU utilization across nodes over time in Scenario 2 for the heuristic approach

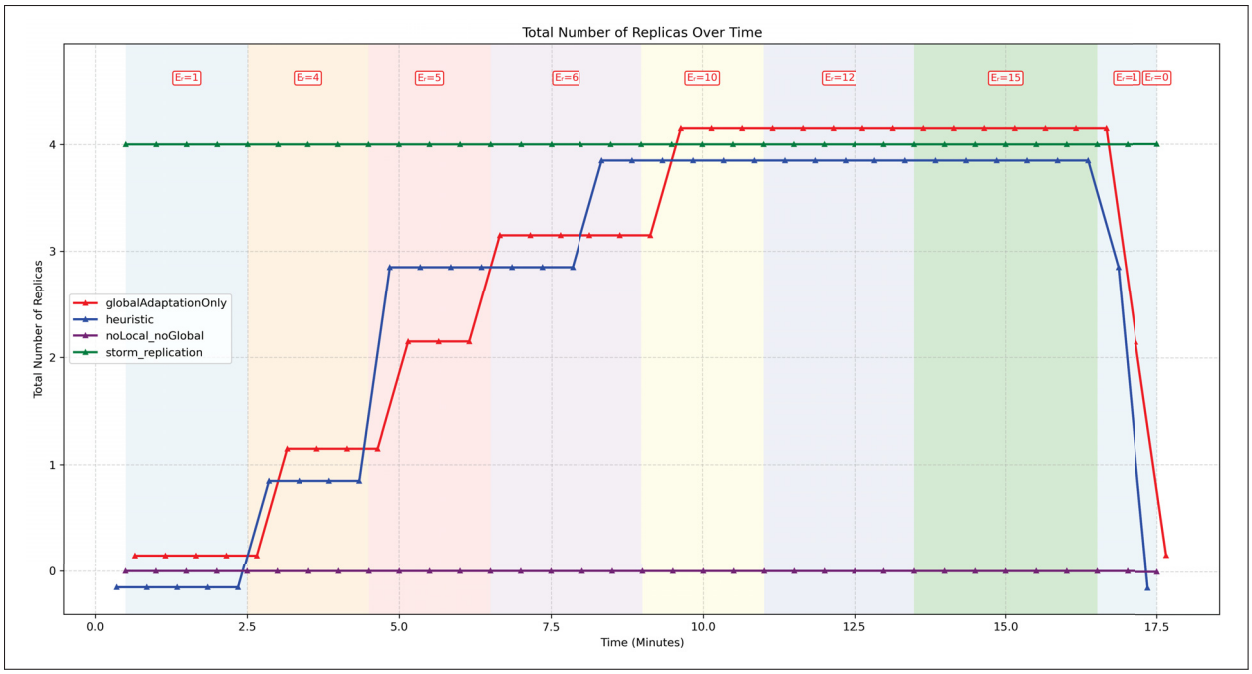



Figure 4.15 Total number of replicas over time in Scenario 2 for the four baselines. The shaded regions correspond to different incoming rates during execution

#### 4.3.2.1.3 Replica Count

The *storm_replication* baseline maintains a fixed number of four replicas throughout the entire timeline, reflecting the predefined configuration in the code. In contrast, the *noLocal_noGlobal* baseline stays flat at zero, indicating that no replication occurs at any point. Unlike the flat patterns seen in *noLocal_noGlobal* and *storm_replication*, both *globalAdaptationOnly* and *heuristic* begin with zero replicas and then gradually increase in a stepwise manner. They eventually reach 4 replica count during periods of high demand, before dropping back to zero once the data emission rate becomes very low.

#### 4.3.2.1.4 End-to-End Latency

The *noLocal_noGlobal* line rises steadily and sharply, peaking at around 350,000 ms when the system processes 15 images/s. The *storm_replication* line remains low at first, then grows

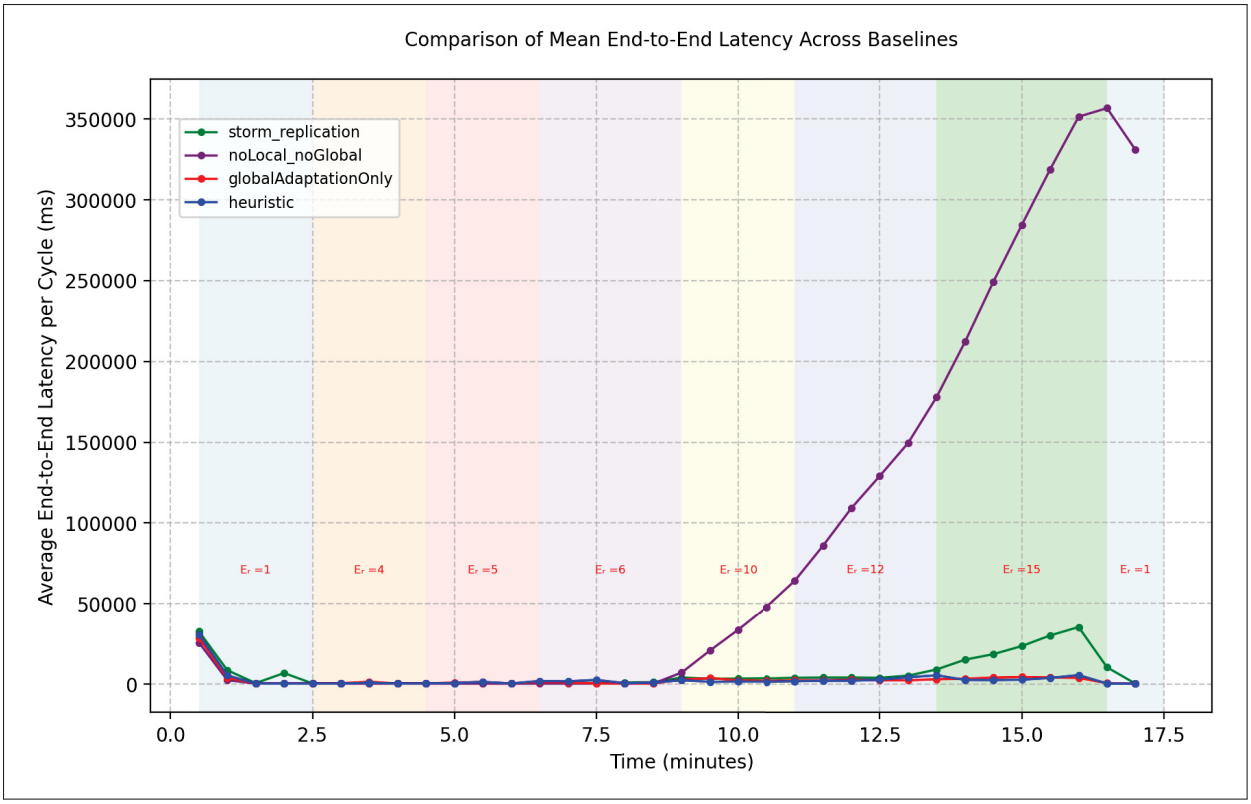



Figure 4.16 Average end-to-end latency of tuples per cycle over time in Scenario 2 for the four baselines. Shaded regions indicate different incoming data rates applied during execution

moderately, approaching roughly 30,000 ms at the peak input rate before decreasing again. Whereas both the *globalAdaptationOnly* and *heuristic* lines stay almost flat throughout the experiment.

The next section explores in great depth how each baseline responds when CPU resources become saturated.

#### 4.3.2.2 Analysis

Table 4.4 provides an overview of input/output counts and mean end-to-end latency for the four baselines in Scenario 2. Across all approaches, the number of processed tuples is consistent and

no data loss occurred and the backpressure mechanism did not restrict the number of of tuples that were processed.

Table 4.4 Summary of input/output statistics and overall mean end-to-end latency for each baseline in Scenario 2

| Approach | Input Count | Output Count | Mean End-to-End Latency (s) |
|---|---|---|---|
| noLocal_noGlobal | 7867 | 7867 | 122.79 |
| storm_replication | 7867 | 7867 | 9.56 |
| globalAdaptationOnly | 7867 | 7867 | 2.70 |
| heuristic | 7867 | 7867 | 2.67 |

The baselines differ significantly in terms of average end-to-end latency. Without replication, *noLocal_noGlobal* baseline suffers from latency above 120 seconds. Using a fixed number of replicas in *storm_replication* lowers latency to about 10 seconds, but this is still far from optimal. In this baseline, latency remains moderate at lower input rates but rises greatly once the emit rate reaches 15 images per second. At this point, the average end-to-end latency climbs to approximately 35 seconds. Adaptive replication strategies prove to be far more effective, with *globalAdaptationOnly* reaching 2.70 seconds and the *heuristic* method achieving the lowest latency at 2.67 seconds.

The latency observed in this scenario cannot be the result of bandwidth congestion since bandwidth was provisioned generously and no bottlenecks appeared in any baseline. Instead, latency is entirely due to CPU limitations, which we examine further through the CPU utilization plots for all baselines.

In the *noLocal_noGlobal* baseline as shown in Figure 4.11, Node 03, which hosts the B1 operator, remains persistently overloaded. This node becomes the primary point of congestion, slowing down the entire pipeline. A similar issue is visible in the *storm_replication* baseline (Figure 4.12). Although one replica is created for each operator, Node 08 consistently displays excessive CPU consumption and turns into a bottleneck that negatively affect the end-to-end latency.

In the *globalAdaptationOnly* baseline as presented in Figure 4.13, replicas are created dynamically when needed, rather than being statically defined. In this case, the Orchestrator makes informed decisions about where to place replicas. Instead of assigning them to limited-capacity nodes such as Nodes 08, 09, 13, and 19, as in the *storm_replication* baseline, it chooses Nodes 17, 18, 20, and 21, which combined provide substantially greater CPU capacity. However, the original operators, especially B1 on Node 03, still keep too much of the load (50%), leading to CPU congestion. As observed in Figure 4.14, the *heuristic* approach fixes this by balancing the load better between replicas and original operators, which removes the persistent bottlenecks.

It is worth noting that CPU utilization measurements can occasionally show variability even under the same input distribution. For instance, in the *heuristic* baseline (Figure 4.14), cycles 29 and 30 display different CPU percentages despite receiving data at the same rate. This is not a measurement error, but a normal variation in CPU sampling. The important point is that the *heuristic* method eventually stabilizes and reaches the correct distribution, even if plots occasionally show high usage.

### 4.3.3 Scenario 3: Bandwidth and CPU Bottleneck

Unlike the previous scenarios, which were designed to isolate the effects of a single type of bottleneck, Scenario 3 reflects real-world conditions where different types of congestion occur in different parts of the pipeline. In this setup, some nodes are limited by CPU capacity, while others are constrained by bandwidth.

The resource configuration for this scenario is given in Table 4.5, which highlights the variation in CPU and bandwidth allocations across nodes.

#### 4.3.3.1 Description of Results

To further increase complexity, a non-linear topology was used, as shown in Figure 4.17. In this setup, B1 distributes its output simultaneously to B2 and B3, each responsible for different types of processing. This means that every input handled by B1 produces two outputs, one sent to each

Table 4.5 Bandwidth and CPU resources assigned to each node in Scenario 3

| VM ID | CPU Cores | In-bandwidth (Mbit/s) | Out-bandwidth (Mbit/s) |
|---|---|---|---|
| Node 02 | 6.0 | 40 | 70 |
| Node 03 | 4.0 | 19 | 4 |
| Node 04 | 2.5 | 25 | 30 |
| Node 05 | 1.5 | 28 | 33 |
| Node 06 | 4.5 | 8 | 25 |
| Node 07 | 2.5 | 3 | 4 |
| Node 08 | 2.0 | 3 | 4 |
| Node 09 | 2.5 | 4 | 7 |
| Node 10 | 2.7 | 6 | 6 |
| Node 11 | 1.5 | 5 | 8 |
| Node 12 | 1.0 | 7 | 6 |
| Node 13 | 1.3 | 7 | 6 |
| Node 14 | 1.0 | 6 | 8 |
| Node 15 | 0.8 | 8 | 7 |
| Node 16 | 2.0 | 7 | 6 |
| Node 17 | 4.5 | 19 | 21 |
| Node 18 | 4.0 | 17 | 23 |
| Node 19 | 3.0 | 15 | 22 |
| Node 20 | 4.0 | 15 | 24 |
| Node 21 | 4.5 | 16 | 25 |

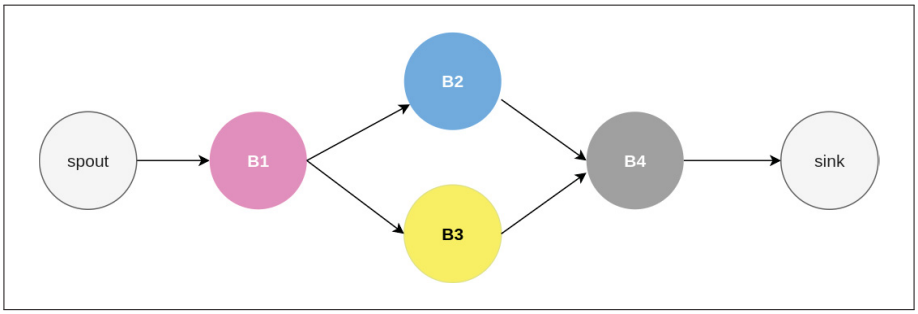



Figure 4.17 Stream processing topology used for the bandwidth bottleneck experiment (Scenario 3)

downstream bolt. Consequently, the output of B1 is heavily utilized, and since both B2 and B3 forward their results to B4, the B4 becomes a critical point of pressure in terms of in-bandwidth.

As in earlier experiments, CPU utilization, in-bandwidth usage, and out-bandwidth usage were measured at the node level, while end-to-end latency was recorded from spout to sink.

#### 4.3.3.1.1 Bandwidth Utilization

The in-bandwidth and out-bandwidth utilization for all four baselines in Scenario 3 is presented in Figure 4.18. As before, the left subplot corresponds to in-bandwidth and the right to out-bandwidth, with shaded regions marking changes in input rate.

As the plot shows both in-bandwidth and out-bandwidth bottlenecks occur in this scenario, but the extent to which they are resolved varies across baselines. The *noLocal_noGlobal* baseline repeatedly encounters out-bandwidth saturation, which continues throughout the experiment. In the *storm_replication* baseline, no bottleneck is visible on the input side, but persistent out-bandwidth saturation appears once the workload increases. The *globalAdaptationOnly* baseline is able to eliminate in-bandwidth congestion, yet on the output side it only provides a temporary relief, as a persistent bottleneck emerges once the incoming rate reaches 17 images/s. By contrast, the *heuristic* baseline quickly addresses congestion on both input and output bandwidth. It could further prevent saturation even under higher loads.

#### 4.3.3.1.2 CPU Utilization

CPU utilization for all baselines is depicted in Figures 4.19 to 4.22 provide a view of how processing limits affect system behavior. In the *noLocal_noGlobal* case, shown in Figure 4.19, nodes generally stay well below critical levels, with only a few brief peaks that never turn into long-lasting saturation. For *storm_replication* (Figure 4.20), some nodes do reach high CPU usage from time to time, but these spikes are infrequent and fade quickly. The situation is quite different in *globalAdaptationOnly*. Figure 4.21 shows that once the input rate climbs, node 05 becomes stuck in critical usage for an extended period, and this continues even after a replica of its operator is added. Finally, displayed in Figure 4.22,the *heuristic* baseline reveals a much

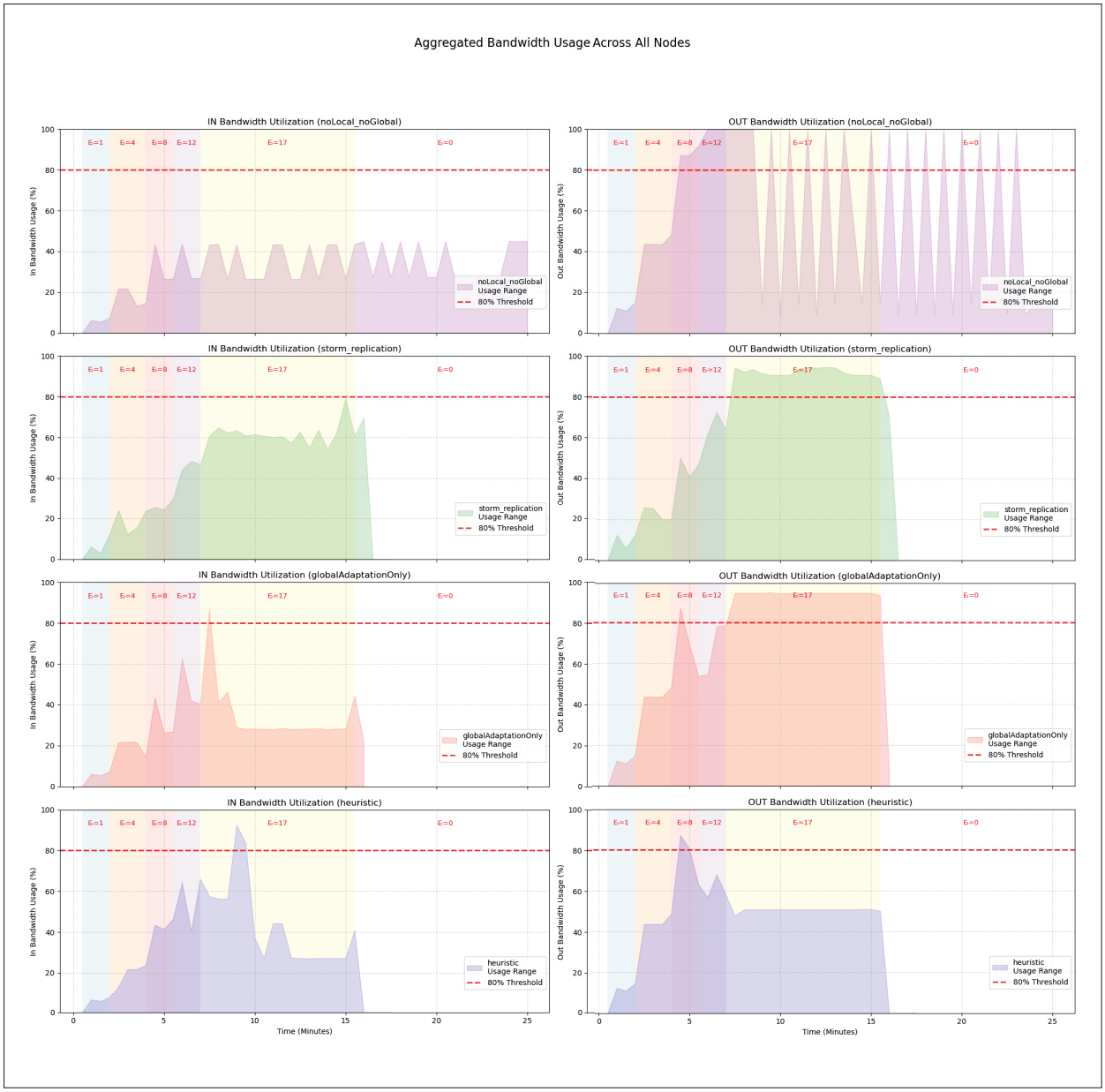



Figure 4.18 Aggregated in-bandwidth (left) and out-bandwidth (right) utilization across all nodes in Scenario 3 for the four baselines. The shaded regions indicate usage over time at different incoming rates, with the red dashed line marking the 80% threshold

more balanced response. While there are short bursts of high load, they don't last, and most nodes remain in the safe range throughout the experiment.

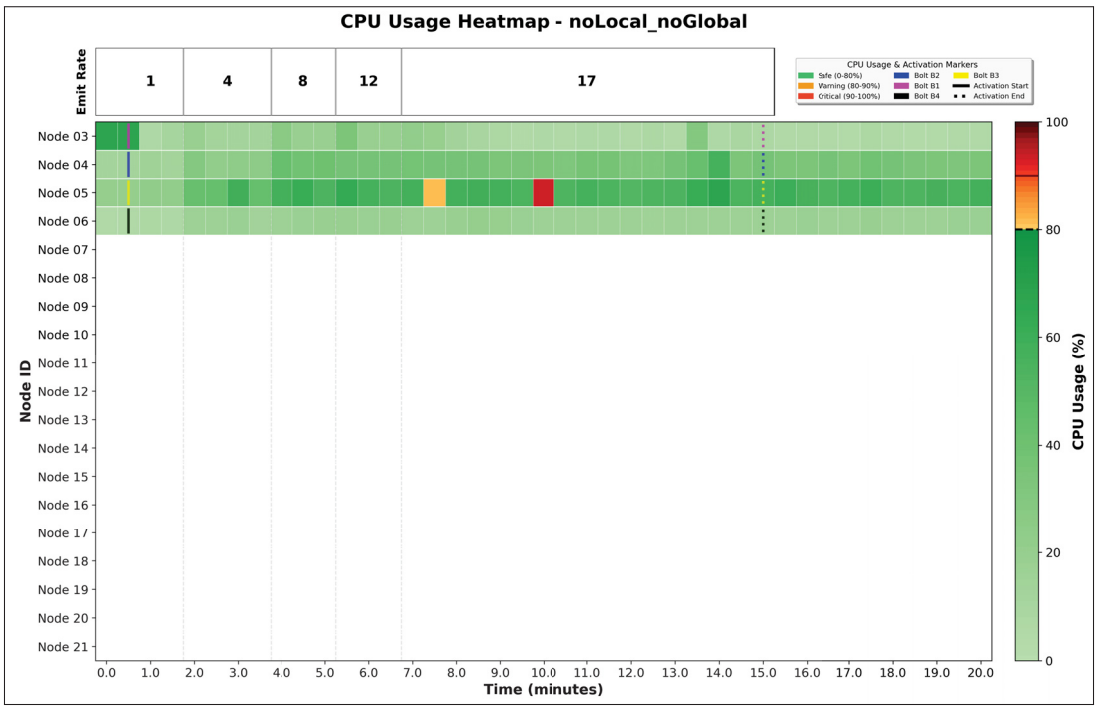



Figure 4.19 CPU utilization across nodes over time in Scenario 3 for the noLocal_noGlobal baseline

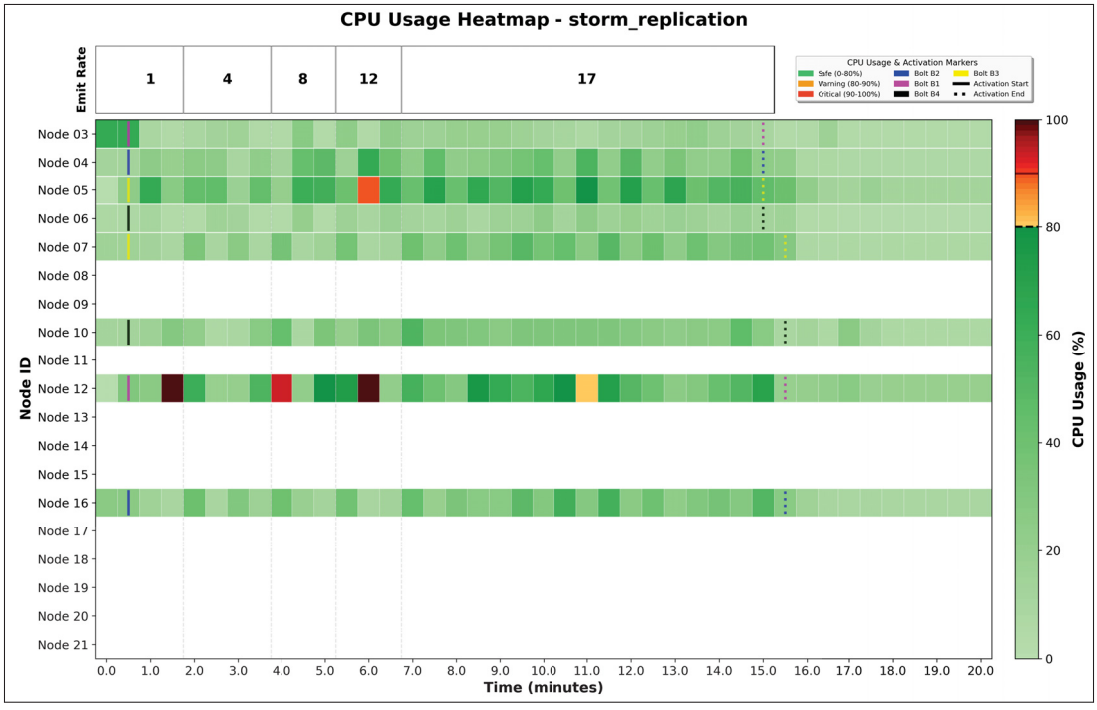



Figure 4.20 CPU utilization across nodes over time in Scenario 3 for the storm_replication baseline

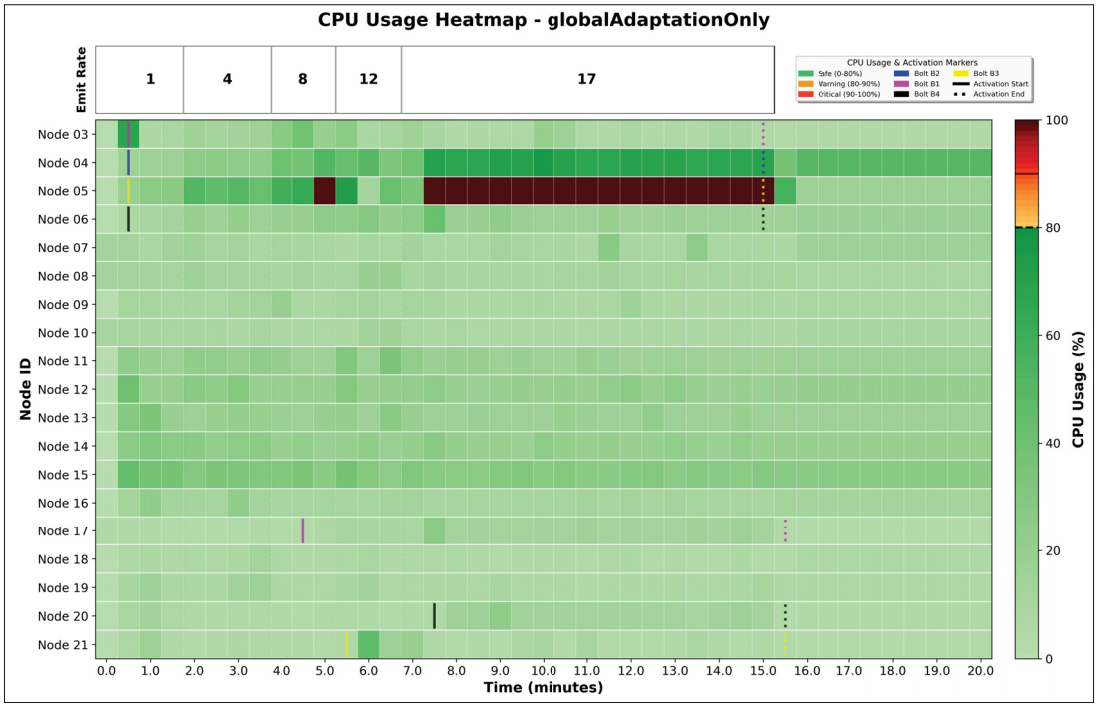



Figure 4.21 CPU utilization across nodes over time in Scenario 3 for the globalAdaptationOnly baseline

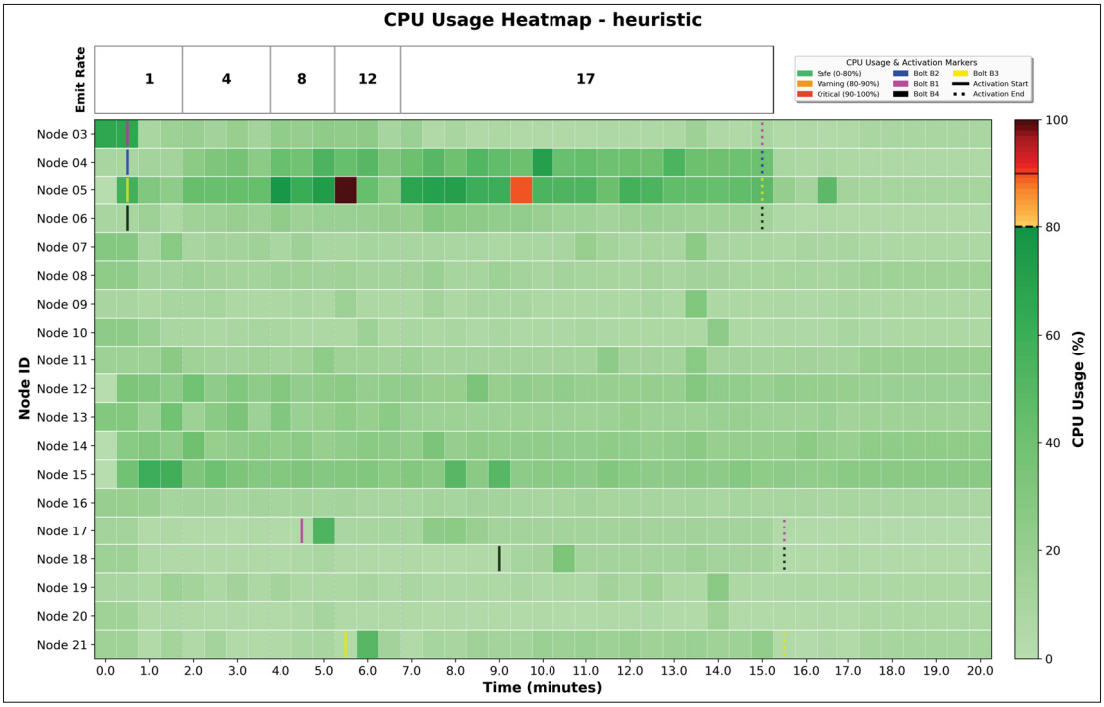



Figure 4.22 CPU utilization across nodes over time in Scenario 3 for the heuristic approach

#### 4.3.3.1.3 Replica Count

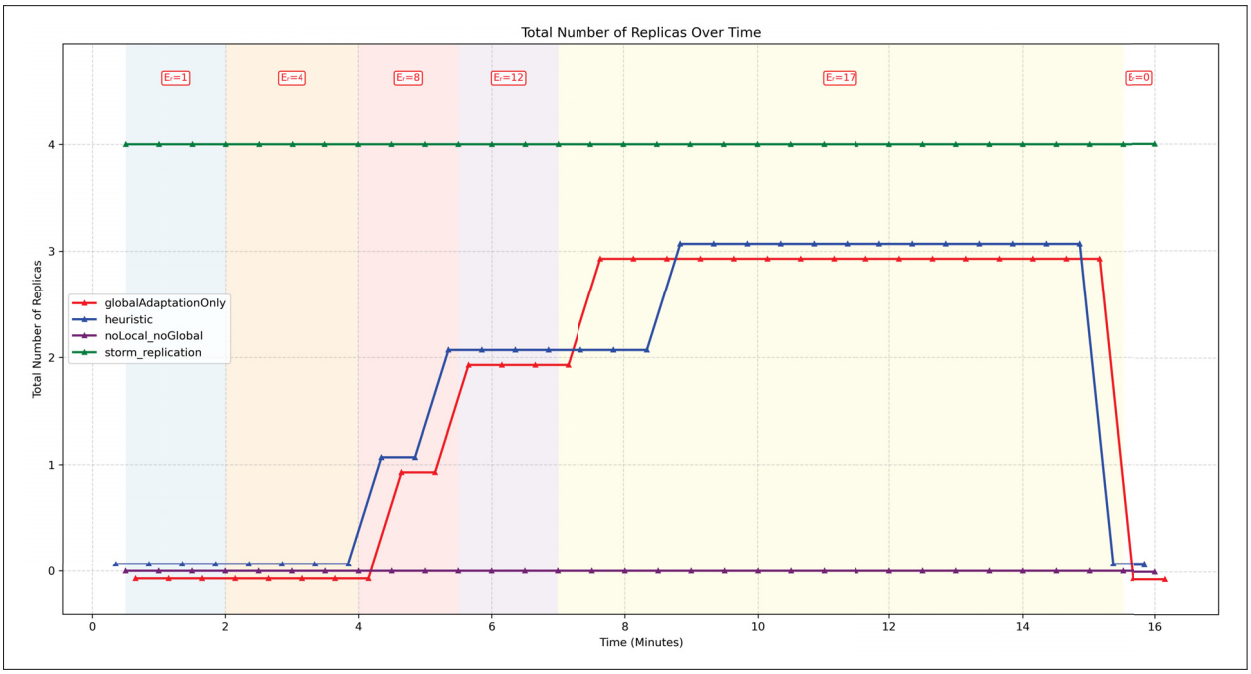
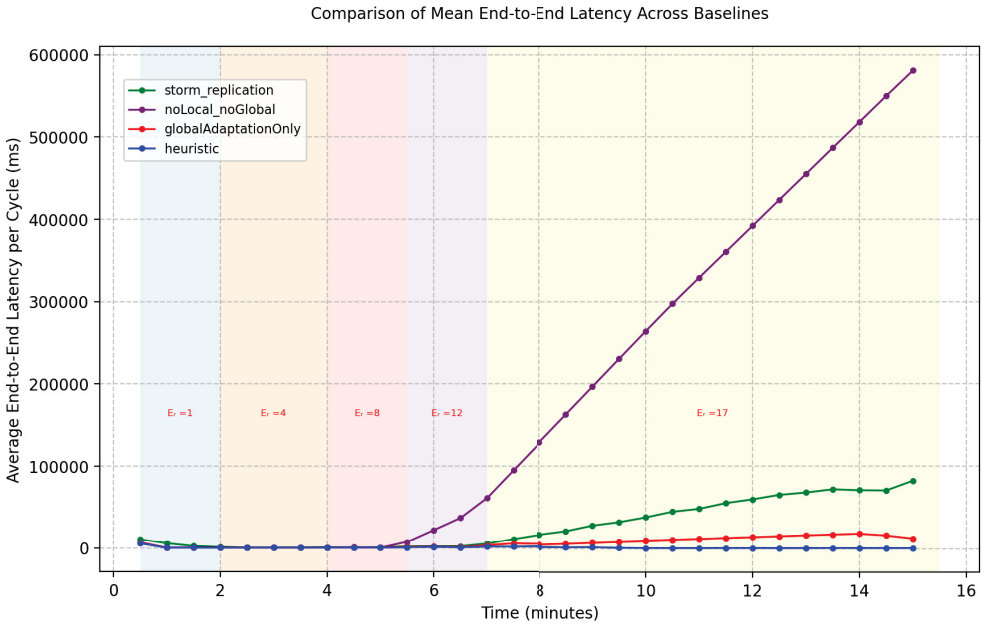
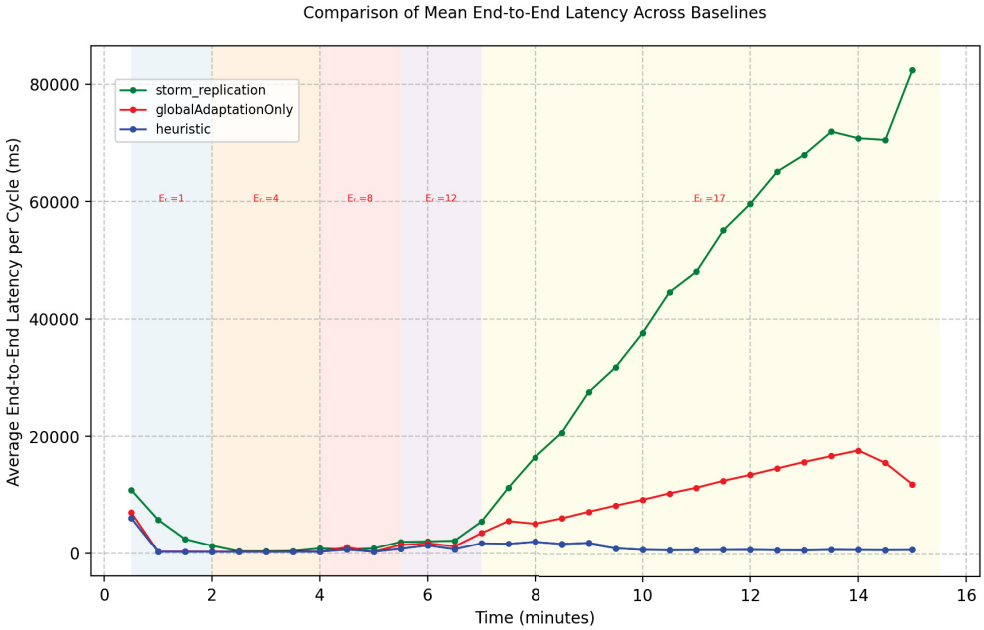



Figure 4.23 Total number of replicas over time in Scenario 3 for the four baselines. The shaded regions correspond to different incoming rates during execution

Figure 4.23 shows the total number of replicas over time for all four approaches. In the *noLocal_noGlobal* baseline, the count remains fixed at zero since no replicas are ever created. In *storm_replication*, the number of replicas is constant at four throughout the entire experiment. This setup was chosen to demonstrate that, in practice, it is not straightforward to predict ahead of time which operators will require replication, so one replica was assigned to each operator from the beginning. By contrast, both *globalAdaptationOnly* and *heuristic* adjust the replica count dynamically, reaching about three active replicas during peak load. These two approaches also remove replicas when the workload decreases and free up resources that are no longer in use. In *storm replication*, however, all replicas remain active regardless of demand.




a) Average end-to-end latency of tuples per cycle over time in Scenario 3 for the four baselines




b) Average end-to-end latency of tuples per cycle over time in Scenario 3 for storm_replication, globalAdaptationOnly, and heuristic

Figure 4.24 Comparison of average end-to-end latency in Scenario 3. (a) shows results for all four baselines, and (b) provides a closer view of storm replication, globalAdaptationOnly, and heuristic

#### 4.3.3.1.4 End-to-End Latency

Figure 4.24a displays the average end-to-end latency of tuples measured across all baselines in this scenario. In the *noLocal_noGlobal* baseline, latency increases sharply once the input rate rises, climbing continuously once the incoming rate exceeds 8 images/s. It eventually reaches extreme values of nearly 600,000 ms.

Figure 4.24b zooms in on the performance of *storm_replication*, *globalAdaptationOnly*, and *heuristic*. It excludes *noLocal_noGlobal* for readability. The results highlight the performance gap more clearly. *Storm_replication* shows a steady and significant rise in latency as the input rate grows, reaching values above 80,000 ms. The *globalAdaptationOnly* performs considerably better, keeping latency below 20,000 ms, but it continues to grow under heavy load. The *heuristic* delivers the most stable outcome, keeping latency near zero during the experiment.

In the next section, we take a closer look at the performance details of each baseline to better understand their individual behaviors.

### 4.3.3.2 Analysis

Table 4.6 provides a comparison of the four methods in terms of mean end-to-end latency. Mean execution times differ substantially across the baselines. For 22,088 output tuples,

Table 4.6 Summary of input/output statistics and overall mean end-to-end latency for each baseline in Scenario 3

| Approach | Input Count | Output Count | Mean End-to-End Latency (s) |
|---|---|---|---|
| noLocal_noGlobal | 11044 | 22088 | 257.72 |
| storm_replication | 11044 | 22088 | 36.65 |
| globalAdaptationOnly | 11044 | 22088 | 8.69 |
| heuristic | 11044 | 22088 | 0.96 |

*noLocal_noGlobal* shows the poorest performance at 257.72 seconds. *Storm_replication*

improves this to 36.65 seconds, *globalAdaptationOnly* further reduces it to 8.69 seconds, and *heuristic* delivers the lowest latency at only 0.96 seconds.

An important aspect to highlight is the latency under peak load. Figures 4.24a and 4.24b show that the three baselines all experience sharp growth in mean execution time as the input rate reaches its highest levels. The *heuristic* method, however, maintains steady latency even under peak load and consistently provides real-time message processing.
The latency table provides the big picture, but to explain why the baselines behave so differently, it is necessary to zoom in on resource usage across nodes.

The high latency observed in the *noLocal_noGlobal* baseline is due to absence of replicas. With only a single operator instance responsible for processing all incoming tuples, the system cannot keep pace with the data rate. As the plots indicate, this baseline frequently encounters out-bandwidth bottlenecks, which contribute to longer delays.

In the *storm_replication* baseline, replicas are available from the start, yet the system continues to suffer from out-bandwidth congestion. Even though this baseline uses one replica per operator, it fails to achieve good latency because replication alone is not sufficient; the placement of replicas and the capacity of their host nodes are equally important. The bandwidth plots show that while some resources remain idle, the static replication strategy prevents the system from leveraging them. Since replicas are fixed at the start, the system cannot dynamically place operators where capacity is available, leading to persistent congestion.

In the *globalAdaptationOnly* baseline, replicas are created dynamically, which helps resolve the out-bandwidth bottleneck when the emit rate reaches 8 images/s and the in-bandwidth pressure at the beginning of 17 images/s. However, the out-bandwidth bottleneck reappears and persists for the rest of the experiment. On the CPU side, Node 05, which runs operator B3, remains heavily loaded even though another replica of B3 is active on Node 21. Because there is no local load balancing, traffic is always split equally, without paying attention to node capacity. As a result, Node 05 receives more data than it can handle while the replica node stays underused. The global adaptation mechanism does not allocate further replicas because it determines that

the existing nodes have sufficient aggregate capacity. However, the lack of intelligent data routing allows congestion to persist and ultimately leads to high latency.

In the *heuristic* approach, the combined use of global and local adaptation enables the system to create replicas where they are most needed and remove them once they are no longer required. Local adaptation further distributes the workload intelligently across replicas, resolving bottlenecks and keeping latency low. This approach also optimizes resource usage by ensuring that available capacity on existing nodes is fully utilized, rather than creating extra replicas and involving additional nodes when the current ones can already handle the load.

### 4.3.4 Scenario 4: Scalability Test

Scenario 4 serves as a scalability test, designed to show that the system can create and manage multiple replicas of the same operator without running into instability or performance problems. Unlike the earlier scenarios, which examined how different replication and load-balancing strategies influence system response and end-to-end latency, the emphasis here is on demonstrating the safe and effective scaling of operators under load.

To align with the goal of this scenario, the resource utilization thresholds that trigger replication were deliberately lowered, and the global adaptation mechanism was configured to run more frequently. Triggering replication earlier through lower thresholds makes it possible to observe the scalability behavior of the system directly, without interference from unrelated bottlenecks caused by factors such as I/O that are outside the scope of this work. The resource configuration for each node in scenario 4 is presented in Table 4.7, which lists the CPU capacity, in-bandwidth, and out-bandwidth assigned to each node.

The upcoming section gives further details of the experiment together with the resource usage plots.

Table 4.7 Bandwidth and CPU resources assigned to each node in Scenario 4

| VM ID | CPU Cores | In-bandwidth (Mbit/s) | Out-bandwidth (Mbit/s) |
|---|---|---|---|
| Node 02 | 6 | 40 | 70 |
| Node 03 | 0.9 | 2.2 | 9 |
| Node 04 | 2.5 | 12.5 | 30 |
| Node 05 | 2 | 14 | 33 |
| Node 06 | 3 | 48 | 45 |
| Node 07 | 2.5 | 43 | 44 |
| Node 08 | 2.5 | 39 | 39 |
| Node 09 | 2.2 | 1.3 | 5 |
| Node 10 | 1.7 | 2.5 | 7 |
| Node 11 | 3.7 | 1.6 | 8 |
| Node 12 | 1 | 2.9 | 5 |
| Node 13 | 1.3 | 3 | 8 |
| Node 14 | 1 | 4 | 6 |
| Node 15 | 0.8 | 2.5 | 9 |
| Node 16 | 2 | 1.1 | 6 |
| Node 17 | 3.5 | 2 | 8 |
| Node 18 | 3.2 | 2.1 | 7 |
| Node 19 | 3 | 2.6 | 6 |
| Node 20 | 3.5 | 1.7 | 8 |
| Node 21 | 4 | 2.1 | 9 |

#### 4.3.4.1 Description of Results

In this experiment, a non-linear topology with two spouts, as shown in Figure 4.25, was used to increase the input rate and place additional pressure on the system.

##### 4.3.4.1.1 CPU Utilization

Figure 4.26 shows the CPU utilization across nodes during the execution of Scenario 4. For the sake of readability, the combined incoming rate from both spouts is displayed at the top of the plot to make it easier to track the input rate at each cycle.

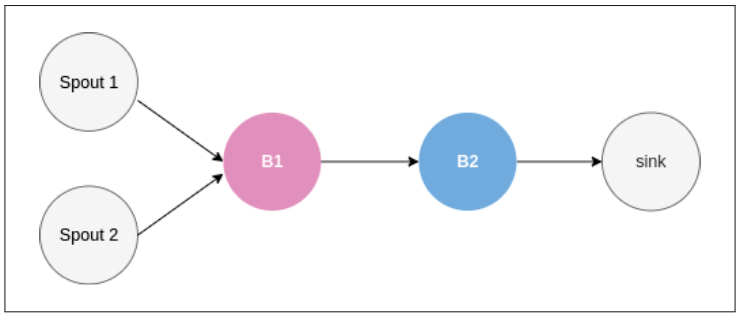



Figure 4.25 Stream processing topology used for the scalability test (Scenario 4)

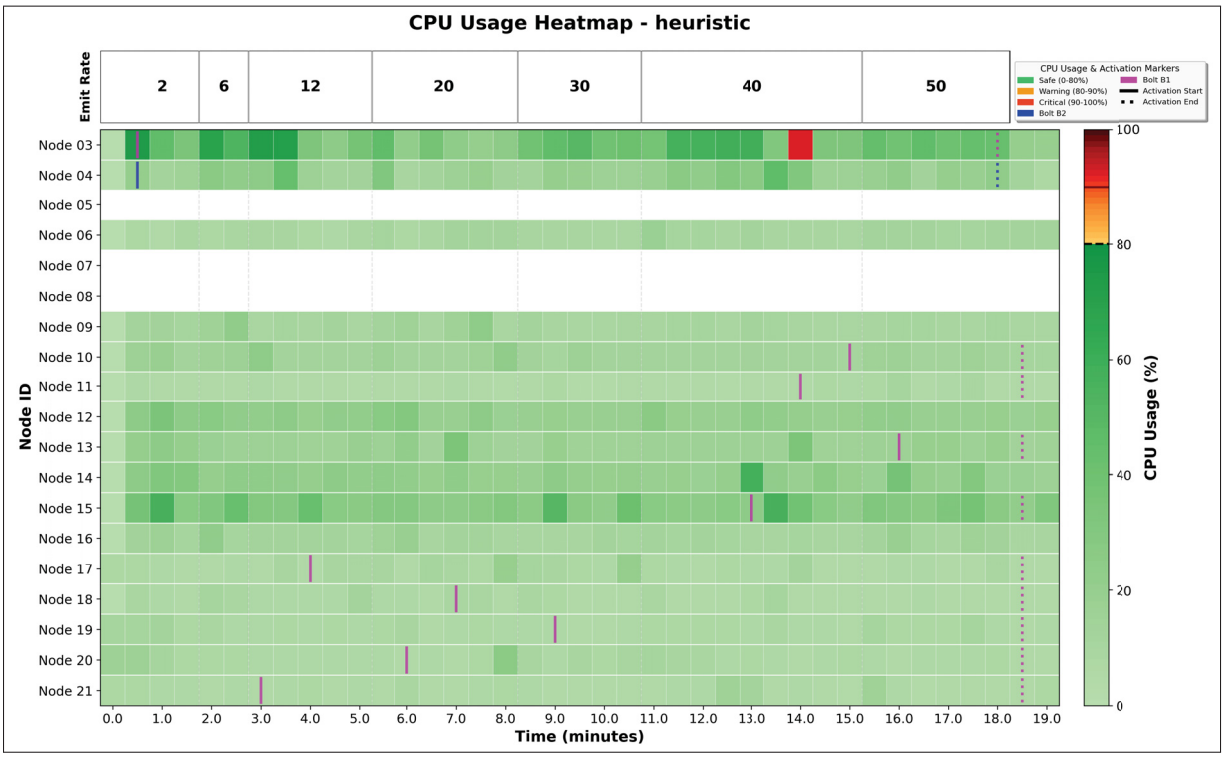



Figure 4.26 CPU utilization across nodes over time in Scenario 4 for the heuristic approach

The results indicate that CPU usage is generally well balanced, with no evidence of lasting congestion. Nearly all nodes remain within safe operating levels throughout the experiment, aside from a single cycle that shows temporary congestion.

#### 4.3.4.1.2 Bandwidth Utilization

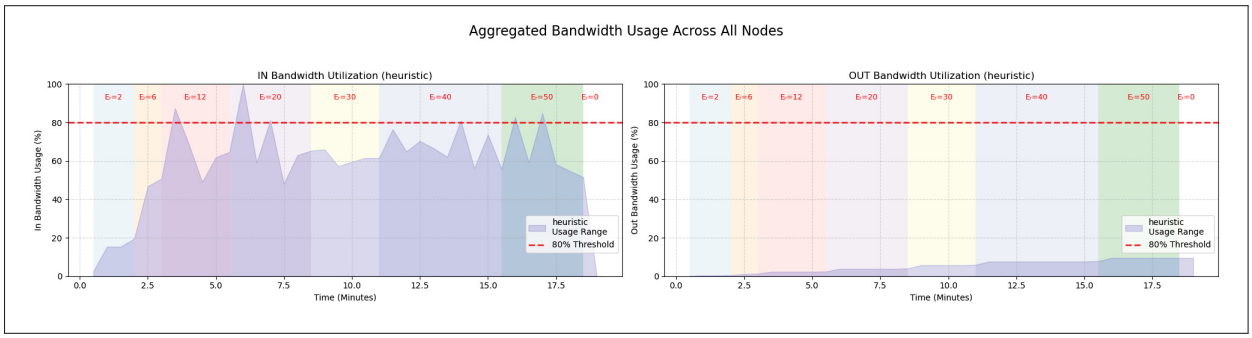



Figure 4.27 Aggregated in-bandwidth (left) and out-bandwidth (right) utilization across all nodes in Scenario 4 for the heuristic approach. The shaded regions indicate usage over time at different incoming rates, with the red dashed line marking the 80% threshold

Figure 4.27 presents the bandwidth usage across all nodes for scenario 4. In this plot, the shaded background indicates the total incoming rate.

No out-bandwidth bottleneck is visible, as usage stays well below the 80% threshold, whereas in-bandwidth occasionally approaches and surpasses this limit, revealing signs of temporary congestion.

#### 4.3.4.1.3 Replica Count

Figure 4.28 shows that operator B1 scales out to nine replicas during peak load and later removes replicas as the load drops. This replication behavior keeps latency low, as the next section will further illustrate.

#### 4.3.4.1.4 End-to-End Latency

Figure 4.29 presents the trend of average end-to-end latency of tuples per cycle for the *heuristic* baseline in Scenario 4. Latency is initially high when the experiment begins, peaking above

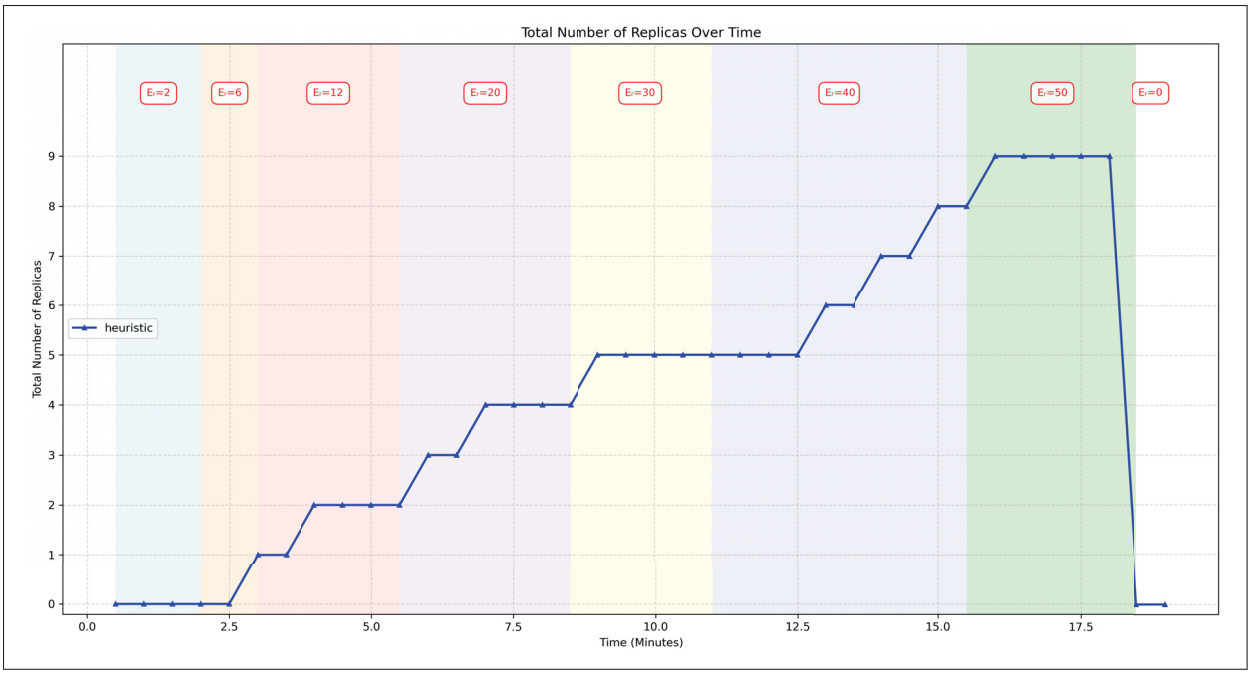



Figure 4.28 Total number of replicas over time in Scenario 4 for the heuristic approach. The shaded regions in the background correspond to different incoming rates during execution

1400 ms. However, after the system stabilizes, latency remains consistently low, staying below 400 ms for the rest of the run even as the input rate reaches its maximum.

#### 4.3.4.2 Analysis

Table 4.8 Summary of input/output statistics and overall mean end-to-end latency for each baseline in Scenario 4

| Approach | Input Count | Output Count | Mean End-to-End Latency (s) |
|---|---|---|---|
| heuristic | 30254 | 30254 | 0.22 |

Table 4.8 reports the performance of Scenario 4 in terms of input and output counts and overall mean end-to-end latency. A total of 30,254 tuples injected into the system and the same number exited, with no data loss observed. The overall mean end-to-end latency is just 0.22 seconds.

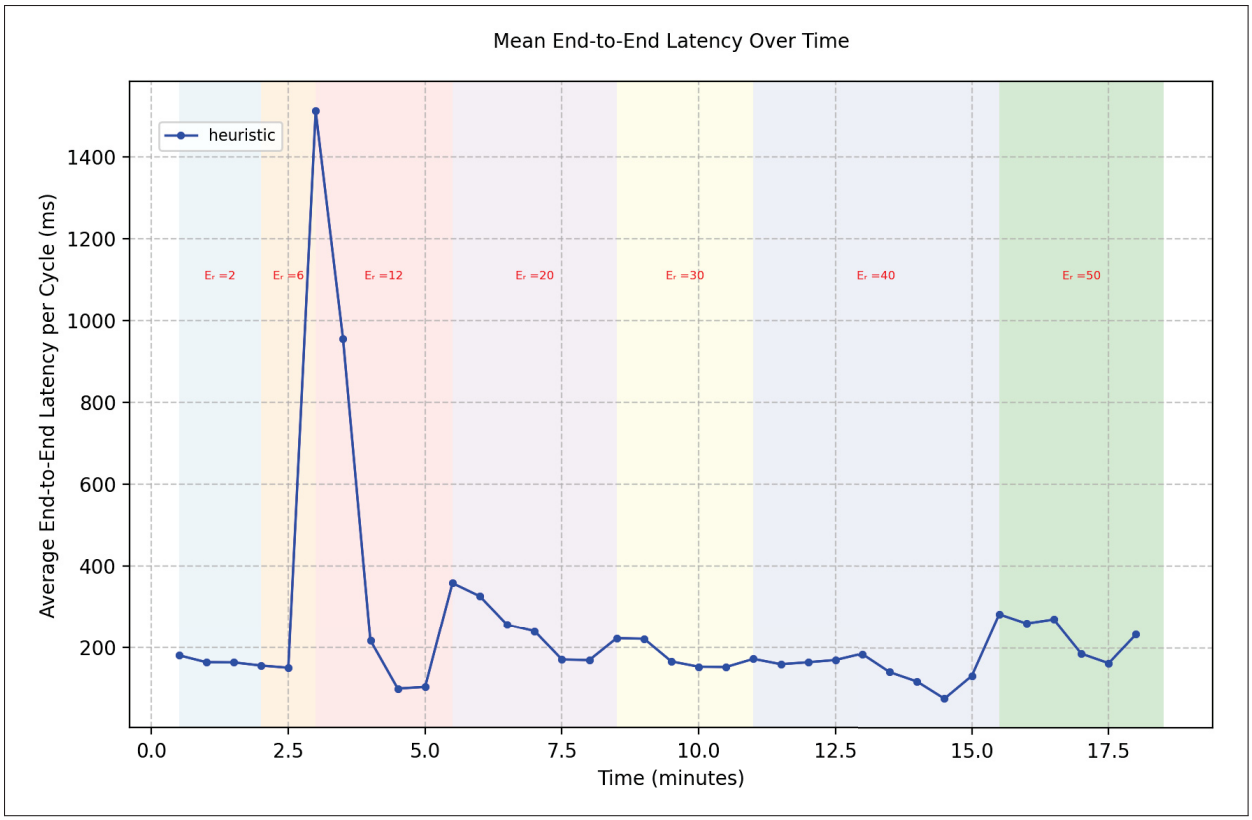



Figure 4.29 Average end-to-end latency of tuples per cycle over time in Scenario 4 for the heuristic approach. Shaded regions indicate different incoming data rates applied during execution

This confirms that the system is capable of creating multiple replicas of the same operator without compromising correctness or efficiency, and it highlights the scalability of the proposed solution.

## 4.4 Discussion

While the proposed approach has shown promising results in improving elasticity and reducing latency, several limitations remain that open opportunities for further research and enhancement. A current limitation of the system is that it cannot automatically take advantage of new nodes added to the cluster, as doing so would require redeploying the topology. In addition, the

maximum number of potential replicas (or generic bolts) is predefined in the code, which restricts elasticity beyond the configured limit.

Future work could extend this approach to support stateful operators, where maintaining consistency across replicas is more challenging. Another important direction is considering additional metrics, such as memory usage, which may further improve the accuracy of adaptation decisions.

Addressing these issues would allow the system to adapt more dynamically to changing cluster conditions and further enhance its scalability and flexibility in real deployments.

# CONCLUSION AND RECOMMENDATIONS

This thesis presented an approach to improving the elasticity of Apache Storm in distributed and heterogeneous environments. The main contribution is the design and implementation of an additional system layer that enables Apache Storm to dynamically adapt to fluctuating workloads without manual intervention or disruption of execution. By addressing bottlenecks in inbound bandwidth, outbound bandwidth, and CPU usage, the proposed solution aims to minimize overall processing latency and improve system responsiveness.

Two complementary mechanisms were introduced to achieve elasticity. The global adaptation mechanism identifies when new replicas are needed and determines where to place them, while also removing underutilized replicas in a safe manner that preserves throughput. The local adaptation mechanism distributes workload intelligently across downstream replicas based on resource availability, preventing bottlenecks from recurring. Implemented on Docker, the system demonstrated the ability to create and remove replicas on the fly, effectively detecting and resolving bottlenecks in real time.

Experimental evaluation confirmed that the proposed approach not only reduces average end-to-end latency but also improves overall resource utilization. Through local adaptation, the system successfully distributes load across existing replicas, ensuring that available CPU and bandwidth resources are used efficiently without unnecessarily creating new replicas. At the same time, global adaptation dynamically removes replicas when the workload decreases, preventing resource wastage. This balance between efficient load distribution and adaptive replica management demonstrates that the system can achieve elasticity while maintaining high performance and optimal resource usage. Furthermore, the experiments showed that the proposed approach can scale effectively, as demonstrated by replicating an operator up to 10 instances.

Overall, this thesis offers a practical and effective method for enhancing elasticity in distributed stream processing, achieving both system responsiveness and efficient use of resources.

## APPENDIX I

## APPENDIX I: LATENCY TABLE

This appendix provides the measured Round Trip Time (RTT) and one-way latency values between all node pairs in the cluster. The RTT values were obtained directly using the standard `ping` utility, which records the time required for a packet to travel from a source node to a destination node and back. To estimate the one-way latency, the RTT values were divided by two under the assumption of symmetric network paths. The complete latency between each pair of worker nodes is summarized in Table I-1.

Table-A I-1 RTT and Latency values between node pairs

| Node Pair | RTT (ms) | One-Way Latency (ms) |
|---|---|---|
| Node 02 – Node 03 | 5.36 | 2.68 |
| Node 02 – Node 04 | 10.08 | 5.04 |
| Node 02 – Node 05 | 57.44 | 28.72 |
| Node 02 – Node 06 | 7.52 | 3.76 |
| Node 02 – Node 07 | 47.04 | 23.52 |
| Node 02 – Node 08 | 39.36 | 19.68 |
| Node 02 – Node 09 | 76.00 | 38.00 |
| Node 02 – Node 10 | 78.72 | 39.36 |
| Node 02 – Node 11 | 38.80 | 19.40 |
| Node 02 – Node 12 | 16.00 | 8.00 |
| Node 02 – Node 13 | 72.96 | 36.48 |
| Node 02 – Node 14 | 106.24 | 53.12 |
| Node 02 – Node 15 | 46.16 | 23.08 |
| Node 02 – Node 16 | 80.72 | 40.36 |
| Node 02 – Node 17 | 4.2 | 2.1 |
| Node 02 – Node 18 | 8.3 | 4.15 |

| Node Pair | RTT (ms) | One-Way Latency (ms) |
|---|---|---|
| Node 02 – Node 19 | 7.8 | 3.9 |
| Node 02 – Node 20 | 2.5 | 1.25 |
| Node 02 – Node 21 | 12.6 | 6.3 |
| Node 03 – Node 04 | 5.12 | 2.56 |
| Node 03 – Node 05 | 76.08 | 38.04 |
| Node 03 – Node 06 | 86.56 | 43.28 |
| Node 03 – Node 07 | 54.24 | 27.12 |
| Node 03 – Node 08 | 22.32 | 11.16 |
| Node 03 – Node 09 | 40.32 | 20.16 |
| Node 03 – Node 10 | 77.36 | 38.68 |
| Node 03 – Node 11 | 19.76 | 9.88 |
| Node 03 – Node 12 | 32.00 | 16.00 |
| Node 03 – Node 13 | 34.88 | 17.44 |
| Node 03 – Node 14 | 119.36 | 59.68 |
| Node 03 – Node 15 | 76.72 | 38.36 |
| Node 03 – Node 16 | 88.08 | 44.04 |
| Node 03 – Node 17 | 20.4 | 10.2 |
| Node 03 – Node 18 | 22.8 | 11.4 |
| Node 03 – Node 19 | 11.2 | 5.6 |
| Node 03 – Node 20 | 7.54 | 3.77 |
| Node 03 – Node 21 | 17.98 | 8.99 |
| Node 04 – Node 05 | 3.84 | 1.92 |
| Node 04 – Node 06 | 46.4 | 23.20 |
| Node 04 – Node 07 | 65.36 | 32.68 |
| Node 04 – Node 08 | 67.6 | 33.80 |
| Node 04 – Node 09 | 83.52 | 41.76 |
| Node 04 – Node 10 | 38.56 | 19.28 |

| Node Pair | RTT (ms) | One-Way Latency (ms) |
|---|---|---|
| Node 04 – Node 11 | 81.36 | 40.68 |
| Node 04 – Node 12 | 24.00 | 12.00 |
| Node 04 – Node 13 | 110.96 | 55.48 |
| Node 04 – Node 14 | 99.52 | 49.76 |
| Node 04 – Node 15 | 41.68 | 20.84 |
| Node 04 – Node 16 | 38.64 | 19.32 |
| Node 04 – Node 17 | 4.6 | 2.3 |
| Node 04 – Node 18 | 11.4 | 5.7 |
| Node 04 – Node 19 | 19.6 | 9.8 |
| Node 04 – Node 20 | 8 | 4 |
| Node 04 – Node 21 | 15 | 7.5 |
| Node 05 – Node 06 | 6.88 | 3.44 |
| Node 05 – Node 07 | 39.12 | 19.56 |
| Node 05 – Node 08 | 18.72 | 9.36 |
| Node 05 – Node 09 | 56.96 | 28.48 |
| Node 05 – Node 10 | 63.2 | 31.60 |
| Node 05 – Node 11 | 46.8 | 23.40 |
| Node 05 – Node 12 | 72.00 | 36.00 |
| Node 05 – Node 13 | 76.24 | 38.12 |
| Node 05 – Node 14 | 48.24 | 24.12 |
| Node 05 – Node 15 | 115.04 | 57.52 |
| Node 05 – Node 16 | 66.16 | 33.08 |
| Node 05 – Node 17 | 6.4 | 3.2 |
| Node 05 – Node 18 | 12.2 | 6.1 |
| Node 05 – Node 19 | 17.8 | 8.9 |
| Node 05 – Node 20 | 5.6 | 2.8 |
| Node 05 – Node 21 | 18 | 9 |

| Node Pair | RTT (ms) | One-Way Latency (ms) |
|---|---|---|
| Node 06 – Node 07 | 15.6 | 7.80 |
| Node 06 – Node 08 | 71.12 | 35.56 |
| Node 06 – Node 09 | 45.2 | 22.60 |
| Node 06 – Node 10 | 86.24 | 43.12 |
| Node 06 – Node 11 | 32.48 | 16.24 |
| Node 06 – Node 12 | 56.00 | 28.00 |
| Node 06 – Node 13 | 109.04 | 54.52 |
| Node 06 – Node 14 | 31.92 | 15.96 |
| Node 06 – Node 15 | 92.88 | 46.44 |
| Node 06 – Node 16 | 55.52 | 27.76 |
| Node 06 – Node 17 | 9 | 4.5 |
| Node 06 – Node 18 | 12.6 | 6.3 |
| Node 06 – Node 19 | 14.4 | 7.2 |
| Node 06 – Node 20 | 7.8 | 3.9 |
| Node 06 – Node 21 | 11.2 | 5.6 |
| Node 07 – Node 08 | 39.84 | 19.92 |
| Node 07 – Node 09 | 42.48 | 21.24 |
| Node 07 – Node 10 | 94.4 | 47.20 |
| Node 07 – Node 11 | 57.04 | 28.52 |
| Node 07 – Node 12 | 16.00 | 8.00 |
| Node 07 – Node 13 | 65.44 | 32.72 |
| Node 07 – Node 14 | 37.92 | 18.96 |
| Node 07 – Node 15 | 86.16 | 43.08 |
| Node 07 – Node 16 | 75.92 | 37.96 |
| Node 07 – Node 17 | 17.4 | 8.7 |
| Node 07 – Node 18 | 4.2 | 2.1 |
| Node 07 – Node 19 | 19.2 | 9.6 |

| Node Pair | RTT (ms) | One-Way Latency (ms) |
|---|---|---|
| Node 07 – Node 20 | 2.5 | 1.25 |
| Node 07 – Node 21 | 12.6 | 6.3 |
| Node 08 – Node 09 | 47.04 | 23.52 |
| Node 08 – Node 10 | 67.12 | 33.56 |
| Node 08 – Node 11 | 76.48 | 38.24 |
| Node 08 – Node 12 | 56.00 | 28.00 |
| Node 08 – Node 13 | 56.00 | 28.00 |
| Node 08 – Node 14 | 103.28 | 51.64 |
| Node 08 – Node 15 | 58.56 | 29.28 |
| Node 08 – Node 16 | 88.24 | 44.12 |
| Node 08 – Node 17 | 7.2 | 3.6 |
| Node 08 – Node 18 | 15.6 | 7.8 |
| Node 08 – Node 19 | 10.2 | 5.1 |
| Node 08 – Node 20 | 16.6 | 8.3 |
| Node 08 – Node 21 | 4.8 | 2.4 |
| Node 09 – Node 10 | 8.4 | 4.20 |
| Node 09 – Node 11 | 76.32 | 38.16 |
| Node 09 – Node 12 | 41.6 | 20.80 |
| Node 09 – Node 13 | 45.52 | 22.76 |
| Node 09 – Node 14 | 51.28 | 25.64 |
| Node 09 – Node 15 | 114.24 | 57.12 |
| Node 09 – Node 16 | 71.52 | 35.76 |
| Node 09 – Node 17 | 18.4 | 9.2 |
| Node 09 – Node 18 | 9.8 | 4.9 |
| Node 09 – Node 19 | 13.2 | 6.6 |
| Node 09 – Node 20 | 6.2 | 3.1 |
| Node 09 – Node 21 | 14.2 | 7.1 |

| Node Pair | RTT (ms) | One-Way Latency (ms) |
|---|---|---|
| Node 10 – Node 11 | 66.00 | 33.00 |
| Node 10 – Node 12 | 77.6 | 38.80 |
| Node 10 – Node 13 | 79.28 | 39.64 |
| Node 10 – Node 14 | 52.4 | 26.20 |
| Node 10 – Node 15 | 38.16 | 19.08 |
| Node 10 – Node 16 | 105.12 | 52.56 |
| Node 10 – Node 17 | 11.6 | 5.8 |
| Node 10 – Node 18 | 16 | 8 |
| Node 10 – Node 19 | 5.2 | 2.6 |
| Node 10 – Node 20 | 14.6 | 7.3 |
| Node 10 – Node 21 | 8.2 | 4.1 |
| Node 11 – Node 12 | 32.00 | 16.00 |
| Node 11 – Node 13 | 102.96 | 51.48 |
| Node 11 – Node 14 | 43.84 | 21.92 |
| Node 11 – Node 15 | 85.12 | 42.56 |
| Node 11 – Node 16 | 60.88 | 30.44 |
| Node 11 – Node 17 | 5.8 | 2.9 |
| Node 11 – Node 18 | 19 | 9.5 |
| Node 11 – Node 19 | 9.6 | 4.8 |
| Node 11 – Node 20 | 12.66 | 6.33 |
| Node 11 – Node 21 | 7 | 3.5 |
| Node 12 – Node 13 | 30.64 | 15.32 |
| Node 12 – Node 14 | 87.36 | 43.68 |
| Node 12 – Node 15 | 48.96 | 24.48 |
| Node 12 – Node 16 | 39.44 | 19.72 |
| Node 12 – Node 17 | 8.6 | 4.3 |
| Node 12 – Node 18 | 13 | 6.5 |

| Node Pair | RTT (ms) | One-Way Latency (ms) |
|---|---|---|
| Node 12 – Node 19 | 6.8 | 3.4 |
| Node 12 – Node 20 | 14.2 | 7.1 |
| Node 12 – Node 21 | 10.4 | 5.2 |
| Node 13 – Node 14 | 98.96 | 49.48 |
| Node 13 – Node 15 | 51.68 | 25.84 |
| Node 13 – Node 16 | 81.68 | 40.84 |
| Node 13 – Node 17 | 7.6 | 3.8 |
| Node 13 – Node 18 | 18.8 | 9.4 |
| Node 13 – Node 19 | 9.2 | 4.6 |
| Node 13 – Node 20 | 13.4 | 6.7 |
| Node 13 – Node 21 | 8.72 | 4.36 |
| Node 14 – Node 15 | 117.68 | 58.84 |
| Node 14 – Node 16 | 60.48 | 30.24 |
| Node 14 – Node 17 | 7.6 | 3.8 |
| Node 14 – Node 18 | 5.8 | 2.9 |
| Node 14 – Node 19 | 11.2 | 5.6 |
| Node 14 – Node 20 | 7.48 | 3.74 |
| Node 14 – Node 21 | 14.2 | 7.1 |
| Node 15 – Node 16 | 94.8 | 47.40 |
| Node 15 – Node 17 | 14.4 | 7.2 |
| Node 15 – Node 18 | 4.6 | 2.3 |
| Node 15 – Node 19 | 18.2 | 9.1 |
| Node 15 – Node 20 | 10.8 | 5.4 |
| Node 15 – Node 21 | 7.6 | 3.8 |
| Node 16 – Node 17 | 17.8 | 8.9 |
| Node 16 – Node 18 | 9 | 4.5 |
| Node 16 – Node 19 | 13.4 | 6.7 |

| Node Pair | RTT (ms) | One-Way Latency (ms) |
|---|---|---|
| Node 16 – Node 20 | 6 | 3 |
| Node 16 – Node 21 | 19.6 | 9.8 |
| Node 17 – Node 18 | 5.2 | 2.6 |
| Node 17 – Node 19 | 12 | 6 |
| Node 17 – Node 20 | 11.8 | 5.9 |
| Node 17 – Node 21 | 16.6 | 8.3 |
| Node 18 – Node 19 | 8.4 | 4.2 |
| Node 18 – Node 20 | 6.8 | 3.4 |
| Node 18 – Node 21 | 15.8 | 7.9 |
| Node 19 – Node 20 | 9.6 | 4.8 |
| Node 19 – Node 21 | 18.8 | 9.4 |
| Node 20 – Node 21 | 11.4 | 5.7 |

## BIBLIOGRAPHY


Abbas, N., Zhang, Y., Taherkordi, A. & Skeie, T. (2017). Mobile edge computing: A survey. *IEEE Internet of Things Journal*, 5(1), 450–465.

Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R. J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E. et al. (2015). The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12), 1792–1803.

Alharthi, S., Alshamsi, A., Alseiari, A. & Alwarafy, A. (2024). Auto-Scaling Techniques in Cloud Computing: Issues and Research Directions. *Sensors*, 24(17), 5551.

Apache Software Foundation. [Accessed: 2025-08-11]. (2025). Apache Storm. Retrieved from: https://storm.apache.org/.

Apache Software Foundation. [Accessed: 2025-11-27]. (2025a). Apache Flink. Retrieved from: https://flink.apache.org/.

Apache Software Foundation. [Accessed: 2025-11-27]. (2025b). Apache Samza. Retrieved from: https://samza.apache.org/.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I. & Zaharia, M. (2010). A view of cloud computing. *Commun. ACM*, 53(4), 50–58. doi: 10.1145/1721654.1721672.

Bokhari, M. U., Makki, Q. & Tamandani, Y. K. (2018). A survey on cloud computing. *Big data analytics: proceedings of CSI 2015*, pp. 149–164.

Borkowski, M., Hochreiner, C. & Schulte, S. (2019). Minimizing cost by reducing scaling operations in distributed stream processing. *Proceedings of the VLDB Endowment*, 12(7), 724–737.

Cao, H., Wu, C. Q., Bao, L., Hou, A. & Shen, W. (2020). Throughput optimization for Storm-based processing of stream data on clouds. *Future Generation Computer Systems*, 112, 567–579.

Carbone, P., Fragkoulis, M., Kalavri, V. & Katsifodimos, A. (2020). Beyond analytics: The evolution of stream processing systems. *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*, pp. 2651–2658.

Cardellini, V., Lo Presti, F., Nardelli, M. & Russo Russo, G. (2018). Optimal operator deployment and replication for elastic distributed data stream processing. *Concurrency and Computation: Practice and Experience*, 30(9), e4334.

Chiang, M. & Zhang, T. (2016). Fog and IoT: An overview of research opportunities. *IEEE Internet of things journal*, 3(6), 854–864.

Cloud, H. (2011). The nist definition of cloud computing. *National institute of science and technology, special publication*, 800(2011), 145.

Dautov, R. & Distefano, S. (2020). Stream processing on clustered edge devices. *IEEE Transactions on Cloud Computing*, 10(2), 885–898.

de Assuncao, M. D., da Silva Veith, A. & Buyya, R. (2018). Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. *Journal of Network and Computer Applications*, 103, 1–17.

Dias de Assuncao, M. (2021). Towards elastic and sustainable data stream processing on edge infrastructure. *Companion of the ACM/SPEC International Conference on Performance Engineering*, pp. 19–20.

Docker. [Accessed: 2025-09-24]. (2025a). Docker overview. Retrieved from: https://docs.docker.com/get-started/overview/.

Docker. [Accessed: 2025-09-24]. (2025b). Swarm mode overview. Retrieved from: https://docs.docker.com/engine/swarm/.

Fragkoulis, M., Carbone, P., Kalavri, V. & Katsifodimos, A. (2024). A survey on the evolution of stream processing systems. *The VLDB Journal*, 33(2), 507–541.

Fu, X., Ghaffar, T., Davis, J. C. & Lee, D. (2019). {EdgeWise}: A better stream processing engine for the edge. *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 929–946.

Grolinger, K., Higashino, W. A., Tiwari, A. & Capretz, M. A. (2013). Data management in cloud environments: NoSQL and NewSQL data stores. *Journal of Cloud Computing: advances, systems and applications*, 2(1), 22.

Han, Y., Chen, L., Wang, H., Chen, Z., Zhang, Y., Yang, C., Hao, K. & Yang, Z. (2025). Learning from the Past: Adaptive Parallelism Tuning for Stream Processing Systems. *arXiv preprint arXiv:2504.12074*.

Hanif, M., Lee, C. & Helal, S. (2020). Predictive topology refinements in distributed stream processing system. *PloS one*, 15(11), e0240424.

Heinze, T., Jerzak, Z., Hackenbroich, G. & Fetzer, C. (2014). Latency-aware elastic scaling for distributed data stream processing systems. *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, pp. 13–22.

Henning, S. & Hasselbring, W. (2021). How to measure scalability of distributed stream processing engines? *Companion of the ACM/SPEC international conference on performance engineering*, pp. 85–88.

Hunt, P., Konar, M., Junqueira, F. P. & Reed, B. (2010). {ZooKeeper}: Wait-free coordination for internet-scale systems. *2010 USENIX Annual Technical Conference (USENIX ATC 10)*.

Jayalakshmi, D. (2021). Adaptive Scheduler in Apache Storm. *International Journal of Engineering Research & Technology (IJERT)*, 10(2), 172–182. doi: 10.17577/IJERTV10IS020075.

Kumar, H. & Ismail, M. A. (2022). Big data streaming platforms: A review. *Iraqi Journal for Computer Science and Mathematics*, 3(2), 10.

Liu, P., Da Silva, D. & Hu, L. (2021). {DART}: A scalable and adaptive edge stream processing engine. *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 239–252.

Liu, S., Liu, L., Tang, J., Yu, B., Wang, Y. & Shi, W. (2019). Edge computing for autonomous driving: Opportunities and challenges. *Proceedings of the IEEE*, 107(8), 1697–1716.

Lombardi, F., Muti, A., Aniello, L., Baldoni, R., Bonomi, S. & Querzoni, L. (2019). Pascal: An architecture for proactive auto-scaling of distributed services. *Future Generation Computer Systems*, 98, 342–361.

Lorido-Botran, T., Miguel-Alonso, J. & Lozano, J. A. (2014). A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, 12(4), 559–592.

Marston, S., Li, Z., Bandyopadhyay, S., Zhang, J. & Ghalsasi, A. (2011). Cloud computing — The business perspective. *Decision Support Systems*, 51(1), 176-189. doi: https://doi.org/10.1016/j.dss.2010.12.006.

Mukkath, S. (2025). Adaptive Load Balancing in Distributed Stream Processing: A Predictive AI-Driven Framework. *European Modern Studies Journal*, 9, 318-325. doi: 10.59573/emsj.9(3).2025.27.

Nagy, S. (2022). The motivations for and barriers to cloud technologies in the field of transportation. *Institutiones Administrationis–Journal of Administrative Sciences*, 2(1), 188–194.

Nasiri, H., Nasehi, S., Divband, A. & Goudarzi, M. (2023). A scheduling algorithm to maximize storm throughput in heterogeneous cluster. *Journal of Big Data*, 10(1), 103.

NVIDIA Corporation. [Accessed: 2025-09-08]. (2025). NVIDIA Jetson Nano Developer Kit. Retrieved from: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/product-development/.

Paganelli, A. I., Mondéjar, A. G., da Silva, A. C., Silva-Calpa, G., Teixeira, M. F., Carvalho, F., Raposo, A. & Endler, M. (2022). Real-time data analysis in health monitoring systems: A comprehensive systematic literature review. *Journal of Biomedical Informatics*, 127, 104009.

Peng, B., Hosseini, M., Hong, Z., Farivar, R. & Campbell, R. (2015). R-storm: Resource-aware scheduling in storm. *Proceedings of the 16th annual middleware conference*, pp. 149–161.

Puttaswamy, R. K. (2018). Scale-Out Algorithm For Apache Storm In SaaS Environment.

Raspberry Pi Foundation. [Accessed: 2025-09-08]. Raspberry Pi 4 Model B Specifications. Retrieved from: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/.

Requeno, J. I., Merseguer, J., Bernardi, S., Perez-Palacin, D., Giotis, G. & Papanikolaou, V. (2019). Quantitative analysis of apache storm applications: the newsasset case study. *Information Systems Frontiers*, 21(1), 67–85.

Röger, H. & Mayer, R. (2019). A comprehensive survey on parallelization and elasticity in stream processing. *ACM Computing Surveys (CSUR)*, 52(2), 1–37.

Salehe, M., Hu, Z., Mortazavi, S. H., Mohomed, I. & Capes, T. (2019). Videopipe: Building video stream processing pipelines at the edge. *Proceedings of the 20th international middleware conference industrial track*, pp. 43–49.

Satyanarayanan, M. (2017). Edge computing: Vision and challenges. *USENIX Association, Santa Clara, USA*.

Sharma, M., Tomar, A. & Hazra, A. (2024). Edge Computing for Industry 5.0: Fundamental, Applications, and Research Challenges. *IEEE Internet of Things Journal*, 11(11), 19070-19093. doi: 10.1109/JIOT.2024.3359297.

Sharma, R. (2024). Enhancing Industrial Automation and Safety Through Real-Time Monitoring and Control Systems. *International Journal on Smart & Sustainable Intelligent Computing*, 1(2), 1–20.

Shi, W., Cao, J., Zhang, Q., Li, Y. & Xu, L. (2016). Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5), 637–646.

Stonebraker, M., Çetintemel, U. & Zdonik, S. (2005). The 8 requirements of real-time stream processing. *ACM Sigmod Record*, 34(4), 42–47.

Sun, D., Chen, H., Gao, S. & Buyya, R. (2024). Orchestrating scheduling, grouping and parallelism to enhance the performance of distributed stream computing system. *Expert Systems with Applications*, 254, 124346. doi: https://doi.org/10.1016/j.eswa.2024.124346.

Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J. M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J. et al. (2014). Storm@ twitter. *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 147–156.

Van Der Veen, J. S., Van Der Waaij, B., Lazovik, E., Wijbrandi, W. & Meijer, R. J. (2015). Dynamically scaling apache storm for the analysis of streaming data. *2015 IEEE First International Conference on Big Data Computing Service and Applications*, pp. 154–161.

Wladdimiro, D., Arantes, L., Sens, P. & Hidalgo, N. (2024). PA-SPS: A predictive adaptive approach for an elastic stream processing system. *Journal of Parallel and Distributed Computing*, 192, 104940.

Wu, M., Sun, D., Gao, S., Li, K. & Buyya, R. (2024). Elastic Scaling of Stateful Operators Over Fluctuating Data Streams. *IEEE Transactions on Services Computing*.

Xu, J. & Palanisamy, B. (2021). Model-based reinforcement learning for elastic stream processing in edge computing. *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pp. 292–301.

Xu, J., Palanisamy, B., Wang, Q., Ludwig, H. & Gopisetty, S. (2022). Amnis: Optimized stream processing for edge computing. *Journal of Parallel and Distributed Computing*, 160, 49–64.

Zhao, Q., Cheng, C.-Y., Wu, C.-Y., Yang, Y., Qureshi, M. A., Liu, H. & Chen, G. (2025). Resiliency and robustness study of the Apache storm-based distributed resilient remote sensing platform. *Sensors and Systems for Space Applications XVIII*, 13483, 126–136.