CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Event Detection in Smart Meter Data with Complex Event Processing

In collaboration with Göteborg Energi

Master's thesis in Computer science and engineering

Vaibhav Talari, Nadia Papa

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

Event Detection in Smart Meter Data with Complex Event Processing

In collaboration with Göteborg Energi

Vaibhav Talari, Nadia Papa

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Event Detection in Smart Meter Data with Complex Event Processing
In collaboration with Göteborg Energi
Vaibhav Talari, Nadia Papa

Supervisor: Vincenzo Massimiliano Gulisano, Department of Computer Science and Engineering
Advisor: Joris van Rooij, Göteborg Energi
Examiner: Vincenzo Massimiliano Gulisano, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Event Detection in Smart Meter Data with Complex Event Processing
In collaboration with Göteborg Energi
Vaibhav Talari, Nadia Papa
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Smart Grids with their accompanying Smart Meters are increasingly taking over from conventional electrical grids and manually inspected electricity meters. This trend gives rise to large amounts of data being collected every day by electricity service providers. Each Smart Meter may emit several readings each hour and can be located at both customers and producers, as well as throughout the infrastructure in locations such as substations.

Through analysing this data, a service provider can respond more swiftly to changes in supply and demand, as well as detect anomalies in the grid and at meters. But the large amount of data that is generated quickly exceeds what could be manually inspected and requires the use of techniques made to analyse large quantities of data. One approach is to analyse the data as it arrives in the form of a data stream, without the need to first save it to permanent memory. Two prominent techniques for analysing data streams are those of stream processing and complex event processing.

This thesis is conducted in collaboration with Göteborg Energi. It investigates the performance difference of stream processing and complex event processing for pattern detection in Smart Meter data. The findings are further used to guide the implementation of a pattern that combines both techniques and to support the creation of pattern templates. The tests are conducted on three patterns, with a primary focus on comparing the effects on latency and throughput under different levels of source parallelism in the jobs. The results show that while stream processing has a performance advantage over complex event processing on Smart Grid data, combining the two techniques can achieve comparable performance. Using stream processing as an aggregation step before complex event processing can maintain high performance while offering potential reusability and simplifying the creation of future patterns.

Keywords: Complex Event Processing, Stream Processing, Smart Grid, pattern matching.

Acknowledgements

We want to thank our supervisor at Chalmers, Vincenzo, for his timely supervision and insightful input. We also want to give our special thanks to Joris and coworkers at Göteborg Energi for the heartfelt welcoming and support.

Vaibhav Talari & Nadia Papa, Gothenburg, 2025-09-01

Declaration of Generative AI

During the post-processing of this work, AI tools have been used to refine grammar and sentence structure in passages of the text. All edited content has been reviewed, and the authors take full responsibility for the content of the published thesis.

Vaibhav Talari & Nadia Papa, Gothenburg, 2025-09-01

List of Acronyms and Vocabulary

Below is the list of acronyms that have been used throughout this thesis, listed in alphabetical order:

AMI	Advanced Metering Infrastructure
API	Application Programming Interface
CEP	Complex Event Processing
GE	Göteborg Energi
IoT	Internet of Things
JVM	Java Virtual Machine
MCU	Meter Concentrator Unit
MID	Meter ID
SG	Smart Grid
SID	Series ID
SM	Smart Meter
SP	Stream Processing
TS	Timestamp
VAL	Value

Contents

List of Figures

List of Tables

1

Introduction

Smart Grids (SGs) are among the many information and communication technologies that our society increasingly depends on, enabling communication through a network of devices throughout the electricity grid's infrastructure [1], [2]. A fundamental component of an SG is a Smart Meter (SM), which continuously transmits real-time data on energy consumption and grid status. This information, conveyed as discrete events, supports efficient monitoring and dynamic control of the grid [3]. To fully leverage the capabilities of SGs, it is essential to analyse and process the high-volume, continuous flow of data (data streams) generated by these interconnected devices [2].

Service providers can respond to supply-demand fluctuations and adapt to emerging trends in real time by carefully analysing these data streams [2], [3]. Furthermore, data streams are utilized to detect anomalies in SM operations, including device malfunctions, localized power outages, and instances of energy theft [3], [4].

Detecting such occurrences within the SG requires analysing event sequences, which provide richer contextual insights into the grids operational status at both micro and macro levels. These event sequences can be organized into patterns that can be used to detect and alert the service provider as they occur. An example of such a pattern could be that of a SM notification that the device cover has been removed, followed by a change in the consumption trend, which together may indicate tampering with the meter. Techniques suitable for detecting patterns in these data streams include Stream Processing (SP) and Complex Event Processing (CEP). Both methods are employed in this thesis and discussed in detail in Chapter 2.

Streaming queries can be constructed using either SP or CEP paradigms, with CEP offering a higher-level abstraction over SP. Consequently, a streaming application can be implemented using one or both approaches. Understanding the performance and trade-off between SP and CEP is valuable for service providers aiming to build efficient monitoring systems. Moreover, both paradigms support the automation of decision-making processes for frequently occurring operational issues, thereby reducing the reliance on manual inspection and intervention. Motivated by these advantages, this thesis explores and evaluates three representative event patterns using both SP and CEP approaches.

The first pattern detects voltage deviations between phases that exceed a predefined threshold. Monitoring these deviations is crucial, as they may cause over-

heating, malfunction, or permanent damage to electrical equipment. The second pattern identifies malfunctioning meters that may prevent the accurate reporting of a customer's consumption or production. The third pattern captures events where current levels surpass safe operating limits, potentially causing a meter fault that may require intervention by the service provider. These patterns are described in further detail in Section 3.1.

To evaluate the performance of these patterns, two experimental setups are used: a single-job setup and a multi-job setup. In the single-job setup, a single processing job is deployed with multiple separate data sources. In contrast, the multi-job setup involves deploying multiple independent processing jobs, each utilizing its own fixed data source. In both setups, key performance metrics, including throughput, latency, CPU utilization, and memory usage, are monitored to assess the system's behaviour and efficiency.

One of the objectives of this thesis is to develop two implementations of each pattern: one using the SP paradigm and the other using CEP. A comparative performance analysis of both implementations is then conducted to evaluate their effectiveness in real-time anomaly detection.

1.1 Background

Electricity is a crucial form of energy, making it critical for any nation. As economies grow, the demand for industrial electricity rises rapidly. At the same time, residential electricity consumption also surges due to improving living standards. Green energy sources distribute power generation while reducing the burden on centralized power generation houses. Traditional power grids operate with centralized power generation, but green energy sources have resulted in decentralized power generation. The distributed nature of the power grid has resulted in embracing the SG paradigm, which makes efficiently controlling the supply of high-quality and reliable electric power important [5].

Furthermore, objectives by the European Union (EU) such as *Fit for 55*, to reduce greenhouse gas emissions to 55% by 2030 and *REPowerEU*, to end European power reliance on Russia before 2030 [6] have made EU member states adopt SGs and SMs to achieve these goals. The implementation of the SG has benefits for energy companies, producers, and customers. SM data enables advanced monitoring and efficient use of energy resources on the power grid.

SMs play a transformative role in modern power grids by enabling the precise monitoring and management of electricity usage. With the adoption of Advanced Metering Infrastructure (AMI), SMs can track information such as power consumption, detect faults, and perform other smart functionality continuously. The resulting data is transmitted securely to a central management system, allowing utility companies to enhance power reliability, improve operational planning, and respond more swiftly to outages. An SG is an advanced cyber-physical system integrating communication networks with the traditional power infrastructure to enable intelligent and automated control. It facilitates the exchange of real-time data, in the form of

events, along with the flow of electricity. A specialized database system known as a Meter Data Management System is used to efficiently handle the vast amounts of information generated in SGs to store, process, and manage the collected data effectively [7], [8].

In today's digital landscape, virtually all information systems are fundamentally driven by events [9], where the SG is one such system. A complex event is characterized as an occurrence that arises from the aggregation or correlation of multiple individual events, making it identifiable only when these underlying events are observed collectively.

In event-driven architectures, events are the fundamental building blocks that enable systems to react to changes and trigger actions in real-time. Therefore, an event represents an activity in a system at a given instant, denoting the system state or a deviation. At its core, an event typically comprises a unique identifier, a descriptive message detailing the nature of the event, and a timestamp indicating when the event was generated. Different events can be associated with time, causality, and aggregation. If event A causes event B, the two events are considered correlated, implying that A must occur before B [10], [11].

1.2 Motivation

SGs with their SMs have led to a great increase in power usage data that is accessible to utility companies and power system researchers. This shift represents a significant advancement, moving from a single manual meter reading per month to multiple readings captured over an hour. Applying a data-driven approach to decision-making is necessary, as SM adoption has led to a continuous inflow of data. This increasing influx of data offers the potential for higher-quality analytics and improvements to policies and decision-making. While much of this data is benign, indicating normal functioning as intended, the area of interest lies in the subset of data that signals unusual activity requiring attention. The expansion of automated detection and notification of anomalous events is increasingly necessary to address faults or malicious attacks in these complex, distributed networks. This thesis aims to evaluate the effectiveness of various techniques for identifying patterns within SM data streams.

Motivated by this objective, we investigate pattern detection techniques, specifically SP and CEP, in the context of stream data. To assess and compare the performance of these techniques, a comprehensive set of evaluation metrics is employed. The performance evaluation of the patterns is based on four key metrics: throughput, latency, CPU utilization, and memory usage. Together, these metrics provide a well-rounded evaluation framework for assessing the efficiency and responsiveness of the system. Throughput and latency are particularly important, as they indicate the systems ability to handle high volumes of events and the time required to process them. In contrast, CPU utilization and memory usage offer insights into the overall resource consumption of the application, highlighting its computational efficiency under different workloads.

Both SP and CEP have been used in other studies either to target the SG domain or to analyse data from SGs as part of their evaluations [1], [12]–[14]. These techniques can operate on data as it is accumulated over time without the need to wait for it to be available in its entirety, allowing them to operate directly on the theoretically unbounded data streams [15]. While SP allows for versatile analysis and manipulation of data streams, CEP focuses on pattern detection with a higher level of abstraction in these same streams. By using CEP, a user can define patterns that consist of multiple events. These events are expected to occur together, often within a certain time frame or in a specific sequence. When this happens, they may signal a complex event, such as the earlier example, device tampering. The same functionality can be achieved by the use of SP with a greater performance potential. However, since SP operates at a lower level of abstraction, replicating CEP semantics requires the construction of more complex operator pipelines. In contrast, CEP can express the same logic in a more concise and declarative manner.

1.3 Problem description

The difference between SP and CEP sets ease-of-use against performance, both of which are important qualities and may influence their implementation capability in a real-world setting. The current framework for monitoring abnormalities in the data at GE is a combination of manual inspection as a result of customer complaints, together with some automatic detection based on thresholds. As the adoption of SGs spreads and data volume rises, this kind of approach scales poorly, and the possibility of errors in manual inspection increases.

The technologies of SP and CEP provide effective approaches for automating the detection of predefined anomalies in SGs. One of the primary challenges in adopting these technologies lies in managing the resource requirements of their implementations and understanding how the choice of tools impacts system scalability. SP and CEP differ in terms of their programming abstractions and runtime behaviour, which in turn affect their performance and resource utilization. These differences are critical when deciding which approach to adopt and how to architect the solution. An important consideration in this context is how the system will scale under increasing data loads. Scalability strategies can generally be classified into two categories: scaling up and scaling out [16]. The scaling-up scenario implicates a single-job evaluation, where one processing job handles increasing volumes of data from multiple sources. This approach offers a perspective on executing a streaming application as a single job and helps assess the feasibility of deploying it on a single node. In contrast, the scaling-out scenario implicates a multi-job evaluation, in which multiple independent processing jobs are deployed, each consuming data from a fixed source. This setup provides a view of running a streaming application as a distributed set of jobs, offering insights into deployment across multiple nodes. Together, these evaluation strategies offer valuable guidance on infrastructure selection and resource allocation, depending on whether SP or CEP is used, and on how each technique is best deployed. The performance evaluation in this thesis is limited to four key metrics: throughput, latency, CPU utilization, and memory usage.

Another key challenge spans multiple stages of the development lifecycle. It begins with domain experts identifying relevant data characteristics needed to model meaningful patterns, followed by the accurate and efficient implementation of these patterns. Additionally, effective coordination and communication among the various stakeholders are essential to ensure that the resulting system meets both technical and operational requirements.

Research Questions

1. What are the performance differences between SP and CEP implementations for a set of metrics and patterns related to the SG?
    - This question compares the performance of SP and CEP in terms of throughput, latency, and CPU & Memory utilization.
2. Can generalized SP and CEP rule templates be constructed to capture a subset of event patterns for data from the SG?
    - This question investigates how pattern templates could be formed based on performance and reusability insights from the implemented patterns.

1.4 Scope

The thesis is limited to the evaluation of three predefined patterns, selected in collaboration with domain experts at Göteborg Energi (GE). These patterns are chosen based on previously identified anomalies and do not involve the discovery of new patterns or the analysis of emerging trends in historical data. The evaluation is restricted to a dataset representing seven consecutive days of SM measurements, supplemented with artificially generated events to support specific testing scenarios. The data stream is strictly chronological, with no consideration of out-of-order events. Although there are many available Stream Processing Engines (SPEs), the assessment of SP and CEP is confined to Apache Flink (Flink) with its CEP extension *FlinkCEP*, reflecting its current use within GE. Such systems can be deployed on single or multiple nodes, this project's scope is limited to a single-node setup.

Patterns 1 and 2 are implemented using both the SP and CEP paradigms. Pattern 3 is designed to build upon the performance insights gained from the first two patterns, with a focus on developing a more reusable implementation. This approach serves as an initial step toward creating generalized, reusable pattern templates tailored for SG. The templates are limited in functionality to Patterns 1 and 2.

1.5 Ethics

The work includes sensitive real-world data collected by Smart Meters owned by GE; if misused, this may pose a potential privacy, security, and financial risk for both GE and its customers. There is a risk of identifying individual households or companies and gaining information connected to their energy consumption based

on the event data. Furthermore, the energy consumption reading can also divulge information about the activities occurring at a location.

Throughout the project, data handling was conducted with strict care and confidentiality, as the data was restricted to remain within systems owned by GE. This also included ensuring that no sensitive data was used in any examples or plots presented in the thesis.

2

Technical Background

This section introduces key definitions and concepts essential for understanding the foundation of this thesis. It begins with an overview of the schema relevant to the scope of the work, followed by a brief introduction to core SG terminology and a general description of SGs. The section then explores fundamental aspects of SP and CEP, with particular attention given to the SP and CEP capabilities provided by Apache Flink. The metrics and system setup are introduced in the Evaluation section.

2.1 Electricity Terminology

This section introduces the fundamental electrical terms, which serve as the basis for constructing and implementing the three patterns presented in this thesis.

Voltage measured in volts, known as potential difference, refers to the difference in electric potential between two points in an electrical circuit [17]. In Sweden, 230 volts is the default operating voltage for households [18]. Current is the rate at which electrons flow through a conductor. Voltage and current determine how electrical energy is transmitted and used in a system.

Active power is the portion of electrical power that performs useful work in an AC circuit. It is measured in watts (W) or kilowatts (kW) and powers devices such as LED lights, appliances, and other electrical loads [19].

Energy represents the total amount of power consumed over time and is measured in kilowatt-hours (kWh). In essence, power is the rate at which work is done, while energy is the total capacity to perform work over a specified duration.

2.2 Smart Grids

SGs are modern electricity networks that leverage digital technology and two-way communication to supply electricity more efficiently, reliably, and sustainably. In a generalized setup, SGs are commonly comprised of advanced sensors, communication networks, and automated control systems to enhance grid functionality. Several key components typically make up an SG: A smart infrastructure system, AMI, and Smart management and protection systems.

Meter concentrators are a component of AMI, which can serve a role in bridging communication across a network of SMs. A group of SMs communicates with a meter concentrator, which aggregates the data and facilitates its transmission to the service provider. These aggregation points often form the backbone of the service provider's monitoring and control infrastructure[20], [21], but are not used by all service providers.

Figure 2.1 presents a simplified overview of an SG and its core components. In this architecture, SMs generate and transmit event data, which is ultimately consumed by the service provider for analysis, monitoring, and control purposes.

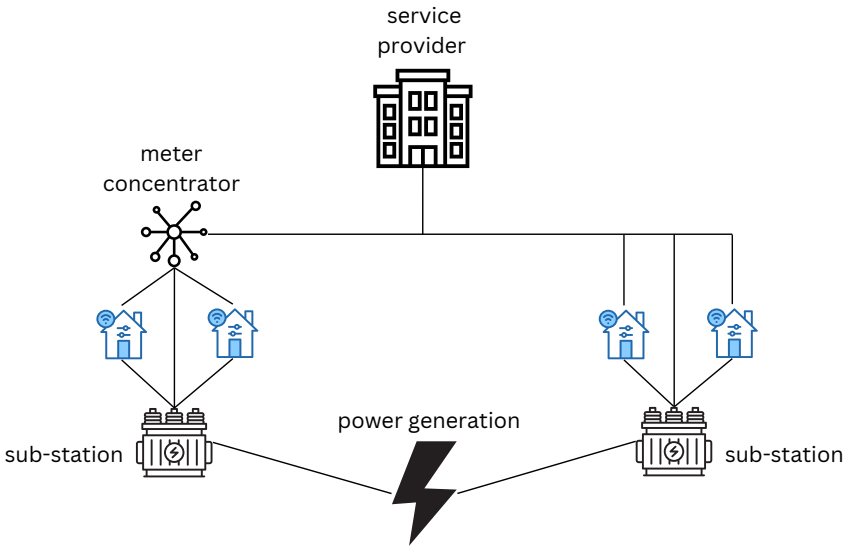


Figure 2.1: A simplified overview of the Smart Grid.

The data arriving at the service provider makes up a data stream of SM events. This stream can then be further processed by the provider to extract information, from the overall grid status to smaller geographical areas and down to that of a single meter.

2.3 Data Stream

Data streams are unbounded, ever-changing, continuous flows of information that can be generated by various sources. Figure 2.2 illustrates multiple SMs generating events that are aggregated into a data stream at the service provider. At the service provider, these events may arrive out-of-order or later than expected, seen from their creation time. This may need to be taken into account when applying functionality to a data stream.

2.3.1 Smart Grid Data

The dataset used in this thesis consists of two separate data streams referred to as SM-values and SM-events; their respective structures are shown in Figure 2.3 (A)

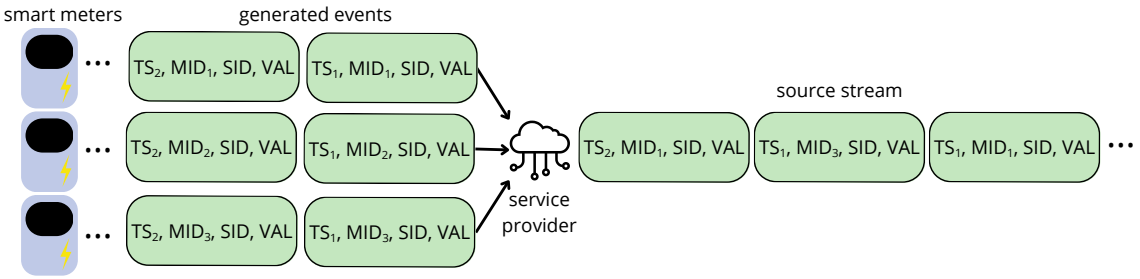


Figure 2.2: Example of a data stream: Event data generated by three Smart Meters is combined into a single data stream at the service provider.

and (B), and a dataset providing metadata about the SMs' phase counts.

Figure 2.3 (A) refers to the SM-values data stream represents the meter readings recorded periodically every 15 minutes. The events from this stream have the format of *TS, MID, SID, VAL*. In addition to a timestamp (TS) and a meter ID (MID), the events of this stream each contain a specific measurement, such as voltage, current, energy, or power, which is distinguished by a corresponding series ID (SID). At any given TS, the SM-values stream for a unique MID has multiple tuples with varying SIDs. The data stream reports measurements of a SID with an accompanying reading value (VAL). For simplicity, "voltage" refers to average voltage, and both "voltage" and "active power" refer to these measurements for all three phases. The events from this stream are represented collectively as *TS, MID, SID, VAL*. For granularity, the individual voltages per phase are V^1, V^2, and V^3. Similarly active power per phase is denoted P^1, P^2, and P^3.

Figure 2.3 (B) illustrates the SM-events data stream, which captures the operational status of an SM. This stream contains aperiodically generated information, referred to as SM-events, with detected anomalies such as power outages, over-current, or high harmonic distortion reported as SID. The events from this stream have the format of *TS, MID, SID*.

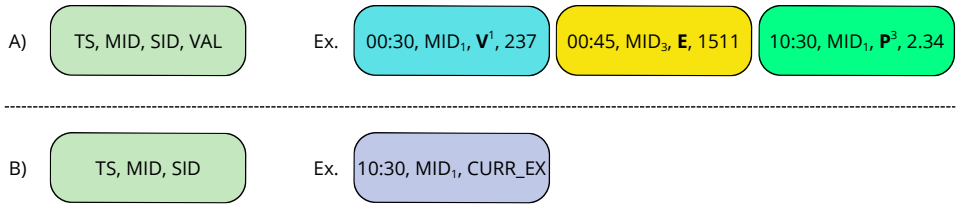


Figure 2.3: Overview of the data stream schemas.

A) SM-values Event Format.
B) SM-events Event Format.

2.4 Stream Processing

Unlike traditional database systems, where data is first persisted to disk before processing, SP enables real-time analysis of data streams as they arrive, without the need for intermediate storage in memory or on disk. While a single source may produce data at a fixed rate, streaming systems typically receive input from multiple sources with data rates that fluctuate over time. Consequently, streaming systems must be capable of dynamically scaling to accommodate these variations and maintain consistent performance.

Flink is the SPE used in this work and is an open-source framework and distributed processing engine designed for streaming over both bounded and unbounded data streams. Flink supports execution in various cluster environments, operates at in-memory speeds, and is highly scalable [22]. To correctly manage timing in stream processing, Flink distinguishes between two time concepts: event time and processing time. Event time refers to the timestamp embedded in the event itself, while processing time is the system time when the event is processed. A key concept in Flink is the use of watermarks to track progress within a data stream that uses event time, where they are essential in generating valid results [23]. A watermark for time t indicates that the system believes it has received all events up to time t. As data streams may include late or out-of-order events, watermarks are essential for determining when windows can be closed and results emitted [24]. Since event time determines the watermarks to ensure accurate and complete computation, any delayed event can lead to delayed output results.

2.4.1 Stream Processing Operators

The underlying SPE provides streaming operators used in the execution of a streaming application. Each SPE defines and implements its own set of core streaming operators, which serve as the building blocks for stream processing. In this work, a subset of commonly used operators is employed, including filter, keyBy, map, flatMap, and various windowing operations [25], [26]. These operators are used throughout the pattern implementations presented in this thesis and are explained in detail below.

The filter operator allows for defining a condition such that only events satisfying the condition are passed to downstream operators, while those that do not match are discarded. The keyBy operator is used to partition a stream by a specified key, such as a unique identifier (e.g., MID). This operation groups related events, enabling further computations to be applied independently to each key group. Figure 2.4 illustrates an example of a keyBy operation applied to partition a data stream by MID.

The map and flatMap operators are used to transform elements in the stream. Both operators apply a user-defined transformation function to each incoming event. The map operator establishes a one-to-one relationship, producing exactly one output event for each input event. In contrast, the flatMap operator supports a one-to-many or one-to-zero transformation, allowing a single input event to produce zero

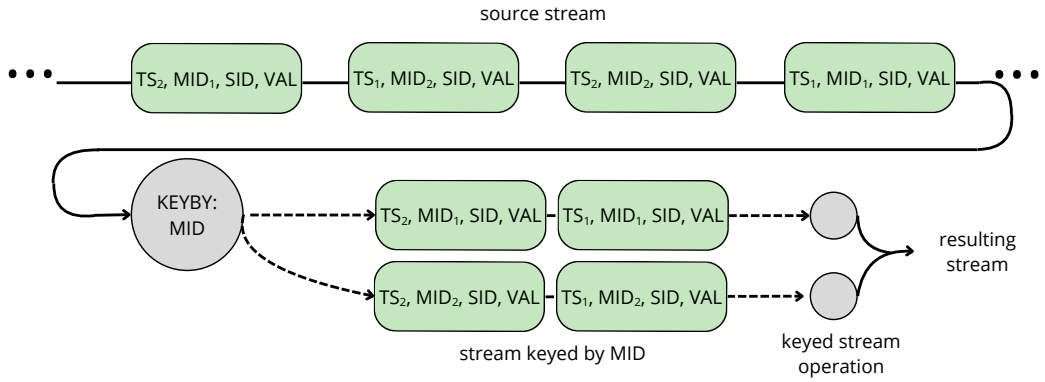


Figure 2.4: A visualization of the *keyBy* operation on a data stream based on MID: The keyBy operation splits the source stream based on the chosen key. This allows for stream operations to be applied on each key in isolation.

or more output events. These transformation operators are particularly useful for enriching, filtering, or restructuring events in real time.

The operators discussed above are considered stateless, as they do not retain any information between processed events. In contrast, windowing operations are stateful, as they maintain an intermediate state over a defined time. Common types of windows include tumbling and sliding windows. Tumbling windows divide the stream into fixed-size, non-overlapping intervals and process data within each discrete window [27]. Sliding windows, on the other hand, allow for overlapping intervals, enabling an event to appear in multiple windows. The tumbling window is defined by a window size, while a sliding window is defined by both a window size and a slide interval [28]. Figure 2.5 illustrates both tumbling and sliding window behaviour.

2.5 Complex Event Processing

CEP is part of the event-driven architecture that evaluates a confluence of events based on user-defined patterns and produces complex events, which are then consumed by downstream operators or processes [11]. Compared to SP implementations, CEP queries are a high-level abstraction of the SP API. Although using the low-level stream processing API gives more flexibility over CEP rules, CEP abstracts this complexity to capture event patterns using a high-level CEP query language. To illustrate CEP, an example from Pattern 3 of the thesis, which detects overcurrent scenarios, will be presented below. The SM infrastructure belonging to GE provides *SM-values* and *SM-events* data streams, which include meter readings and descriptive event messages such as under-voltage and current-limit-exceed.

The over-current pattern involves analysing both the SM-values and SM-events data

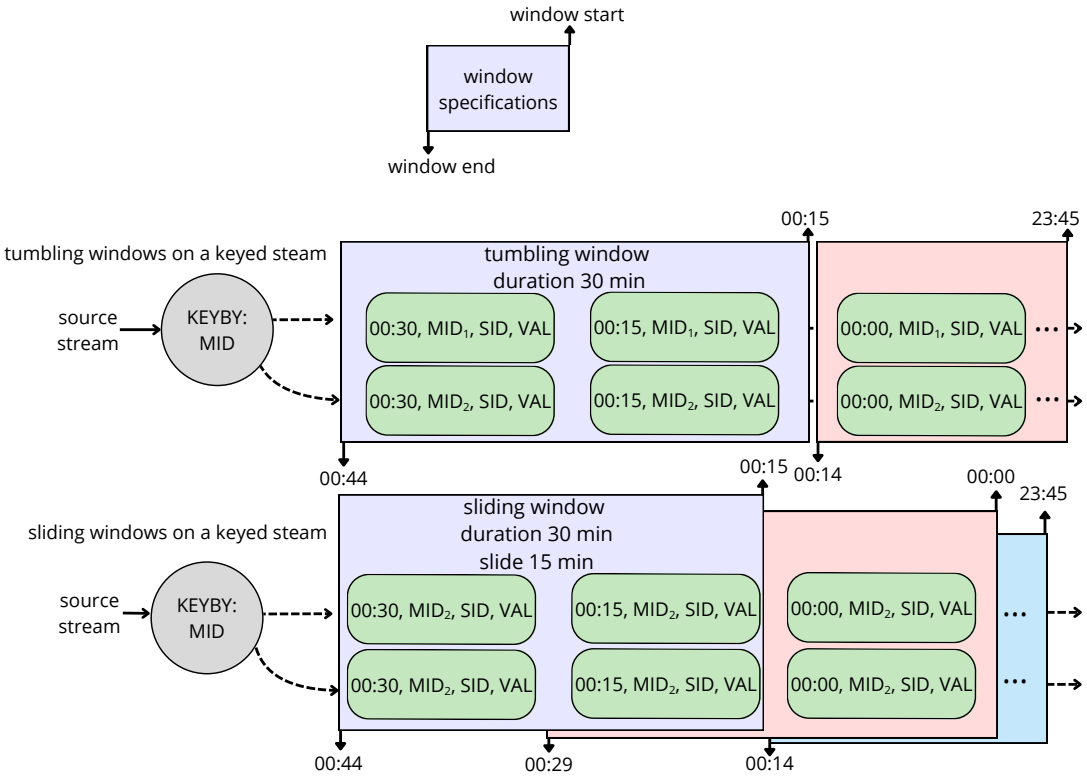


Figure 2.5: A visualization of sliding and tumbling windows on a keyed data stream: The tumbling windows follow each other in strict succession, where one window starts as the previous ends without overlap. The sliding window has an overlap that depends on the duration and the slide of the window.

streams. Under normal operating conditions, an SM reports regular operational values, that is, non-zero voltage and active power readings, in the SM-values stream. At a certain time t, the SM logs a current-limit-exceed event in the SM-events stream. Following this, at two subsequent timestamps t_1 and t_2 (where $t_2 > t_1 > t$), the SM-values stream reports zero voltage and zero active power. This sequence, a current limit exceed followed by two consecutive zero readings, represents a confluence of events indicative of an overcurrent scenario. Figure 2.6 illustrates this pattern, which suggests a meter fault triggered by exceeding current thresholds. Detecting such scenarios involves correlating events across different data streams and identifying temporal patterns. This is well-suited to a CEP framework, which offers a higher-level abstraction for event pattern detection compared to implementations using low-level stream processing operators.

In this work, CEP patterns are implemented using the FlinkCEP library, which operates on top of Flink [29]. The library provides a feature known as the skip strategy, designed to manage situations where the same event appears in multiple matches. The library offers contiguity variation, defining how to match event sequences to a pattern, such as "next" and "followedBy". By specifying a contiguity

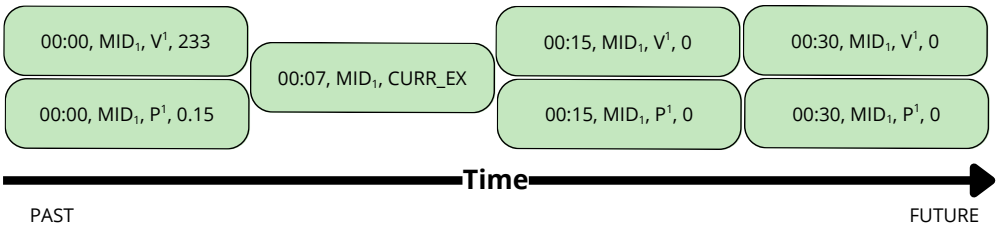


Figure 2.6: Example of events depicting a complex event match: The ordered events matching Pattern 3.

condition for the pattern, the sequences of events it matches can consist of events in direct sequence using the pattern operator "next". Alternatively, it can be defined as a more relaxed sequence using the "followedBy" operator, where non-matching or even matching events may be ignored in between.

The skip strategy *"skip past last"* is the one employed in our pattern implementations. This strategy discards any partial matches that include events already part of a completed match. In other words, once a match is detected, the search for the next match resumes immediately after the last event of the previous match.

3

Methods

This chapter presents an introduction to the patterns and their implementations. It also introduces the corresponding templates derived from the first two patterns. As introduced in Chapter 2, the data stream of SM-values consists of four attributes: timestamp (*TS*), meter ID (*MID*), series ID (*SID*), and value (*VAL*). Its events are represented collectively as *TS, MID, SID, VAL*. The data stream of SM-events, containing anomalies reported by the SM, is represented as *TS, MID, SID*.

3.1 Patterns

The definitions of the three patterns are further elaborated in this section and contextualized within both CEP and SP query implementations. Patterns 1 and 2 include implementations using both SP and CEP, enabling a comparative analysis of the two approaches. Pattern 3, in contrast, incorporates the best-performing techniques identified in Patterns 1 and 2 and unifies them into a single, optimized implementation. Furthermore, the implementation of Pattern 3 serves as a foundation for the development of generalized query templates.

Each query implementation diagram includes indicators labelled (1), (2), and (3), which denote the measurement points for throughput and latency. Throughput at the source is measured at point (1), while point (2) captures throughput after the application of a filter operation. Latency is measured between points (1) and (3). The throughput readings at the source and the filter illustrate that, at the source, all incoming events are read, whereas only relevant events are passed downstream to the operators. Consequently, the throughput measured at the filter reflects the rate at which pattern related events are processed. More details on the evaluation methodology and evaluation setup can be found in Section 4.1.

For the implementation of each pattern, we assume missing data do not contribute to a match of the pattern and expect the SM-values stream data to adhere to the following prerequisites:

- The stream data does not contain any duplicates.
- Each MID emits the data every 15 minutes.

3.1.1 Pattern 1: Voltage Deviation

This pattern captures the voltage deviation across phases for the same timestamp t. Voltage deviations are important to monitor as they can lead to undesirable consequences to electronic equipment. This section presents the details required to construct the voltage deviation pattern with SP and CEP.

From the SM-values data stream tuple *TS, MID, SID, VAL*, the *SID* uniquely identifies voltages by phase. For each timestamp (*TS*), the difference in value (*VAL*) between the corresponding phases determines the deviation. The voltage deviation is thus defined as the absolute difference between any phase voltages exceeding a threshold of *23* volts, corresponding to *10%* of the nominal *230* volts. Zero voltage values are excluded from the pattern, as they indicate other conditions, such as a blown fuse. Figure 3.1 illustrates an example of the pattern, highlighting the complex event matches. For MID_1, the voltage difference between the first and second events is *118*, and between the second and third events is *130*, as indicated by the annotation "MATCH". The remaining events do not satisfy the pattern conditions and are marked as "NO MATCH". In the implementation, an alert is triggered whenever a match occurs between any two phases at the same timestamp. The specific alert generated depends on the order in which the matching event pair arrives.

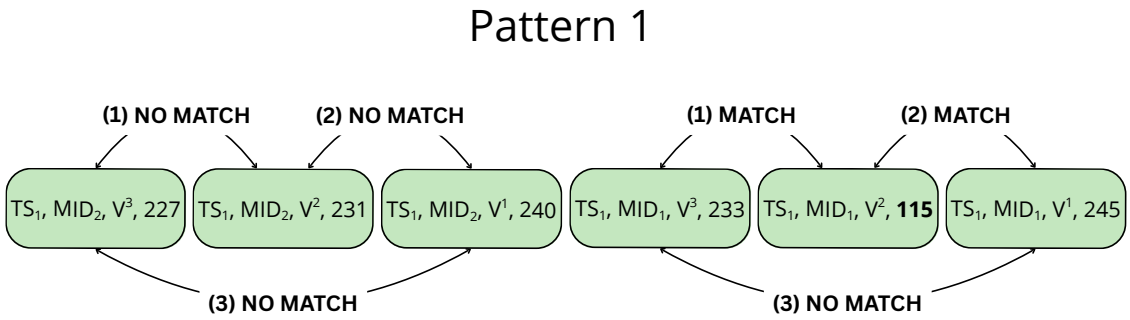


Figure 3.1: Event sequence illustrating Pattern 1: The events associated with MID_1 contain two complex event matches, indicated by "MATCH", while the events of MID_2 do not satisfy the pattern conditions and are labelled "NO MATCH".

Figure 3.2 illustrates the sequence of operators used in the SP query for Pattern 1. An initial filter ensures that the data stream contains only voltage readings for the downstream operators. The stream is then partitioned by MID and evaluated within a tumbling window. Events within each window are examined to determine whether they satisfy the specified voltage deviation condition. Since the SM-values stream emits data every 15 minutes, a 15-minute tumbling window guarantees that readings from different $SIDs$, representing the phase voltages of the same meter, are available within the same window. As a result, the window includes all necessary data to compute the voltage deviation in the process function. Finally, events that exceed the defined voltage deviation threshold are emitted as complex events to a sink.

Figure 3.3 shows an overview of the implementation of the CEP query for Pattern

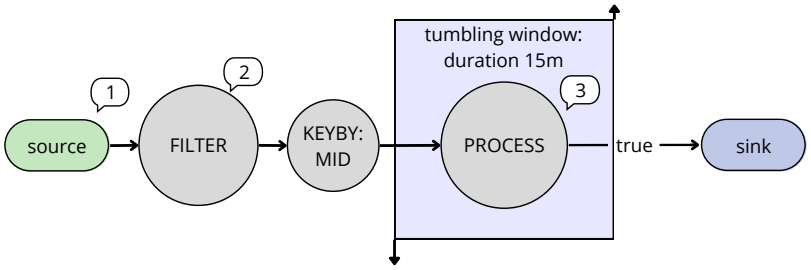


Figure 3.2: Overview of the SP query pipeline for Pattern 1: The key operators include filter, keyBy, and a tumbling window.

1. As with the SP query, the filter removes all values except the voltages and then applies a keyBy function on the MID to apply the pattern on each SM separately. The CEP operator of the pipeline looks for two non-zero voltages within 15 minutes, which are then compared in the process function to detect pairs that have a larger difference than the set threshold. For this pattern, the skip strategy is set to "skip past last". A presentation of the skip strategy is provided in Section 2.5.

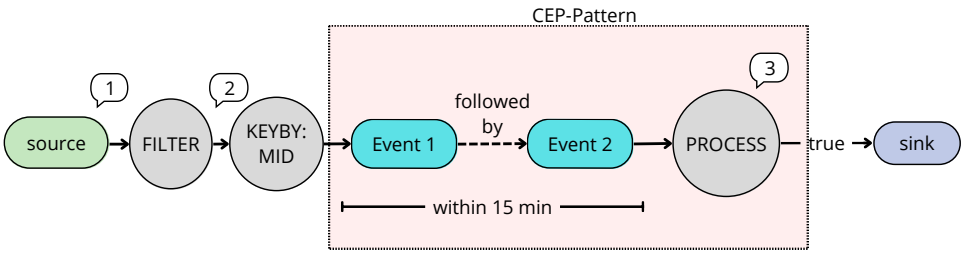


Figure 3.3: Overview of the CEP query pipeline for Pattern 1: The key operators include filter, keyBy, and a CEP pattern.

3.1.2 Pattern 2: Constant Active Energy with Zero Voltage Consumption

The second pattern detects SMs that continuously transmit readings that indicate the meter has no power. Such continuous transmissions should not be possible and imply a fault at the meter. To identify meters exhibiting persistent faults, the pattern requires the sequence of events to occur across three consecutive 15-minute interval timestamps before classifying the meter as faulty. More precisely, the pattern looks for a sequence of active energy and voltage tuples where the active energy value (VAL) should remain constant while the phase voltages all remain zero across the timestamps. Figure 3.4 and Figure 3.5 depict the event sequences for 3-phase and 1-phase complex event matches, respectively. Events that satisfy the pattern conditions are labelled "MATCH", while those that do not are labelled "NO MATCH". Pattern Two is defined using SID values, where E denotes active energy (highlighted in yellow), and the voltages for each of the three phases (highlighted in blue). In the illustrated example, MID_1 corresponds to a complex event match,

with all relevant events enclosed within a dashed black line. In contrast, the non-matching events of MID_2 are indicated by a red dashed line.

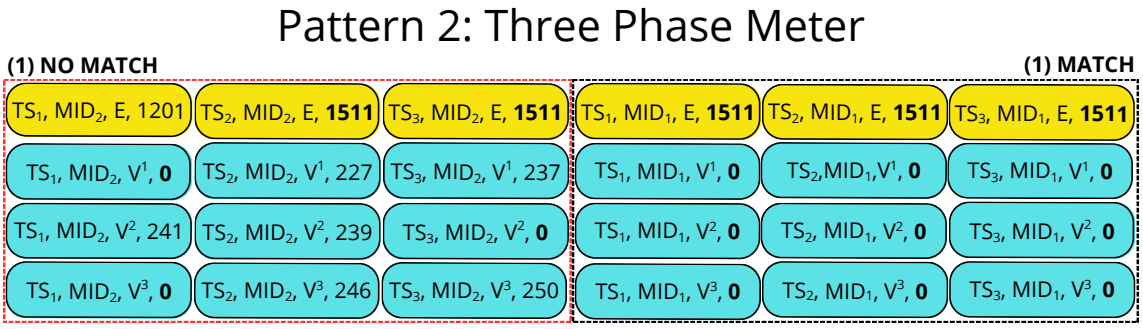


Figure 3.4: Event sequence illustrating Pattern 2 for a 3-phase meter: Complex event matches are outlined in black, while non-matching events are outlined in red.

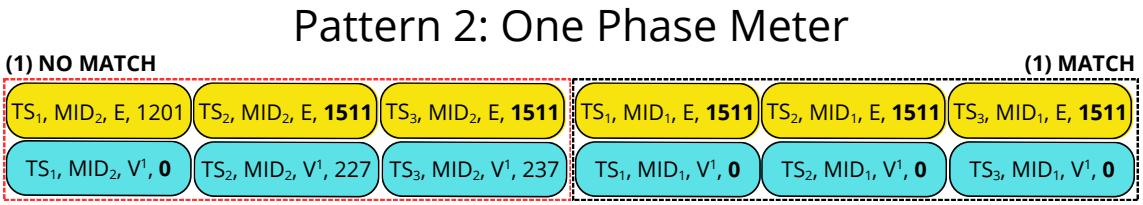


Figure 3.5: Event sequence illustrating Pattern 2 for a 1-phase meter: Complex event matches are outlined in black, while non-matching events are outlined in red.

Figure 3.6 provides a simplified overview of the SP implementation to detect Pattern 2. The filter removes all readings except those reporting a 0 value for voltages and all active energy readings. The stream is then split between 1-phase and 3-phase meters by checking the registered SM type in a database. This phase split also contains a keyBy function and utilizes a state variable to perform the lookup only once for each meter. By using an output tag, the 1-phase meters can then be collected to one stream, and the 3-phase meters collected as a side output, forming a second separate stream. This is done as the number of phases affects the expected number of readings from the SM, as each phase gets its own reading. After applying keyBy on the MID, the number of voltage readings for each MID is counted by an aggregate function in the 15-minute tumbling window, and the active energy reading is saved to be forwarded together with the count by process function 1 to the next step. The next step uses a sliding window of 45 minutes to process three consecutive tumbling window outputs at a time, shown as three windows on top of each other in Figure 3.6. Process function 2 looks for SMs that report a constant value for all active energy readings and have a zero voltage reading count equal to the number of phases of the meter for all three tumbling window outputs. This count is the only difference between the one-phase and three-phase paths.

The CEP query for Pattern 2 is structured into two distinct patterns: one targeting

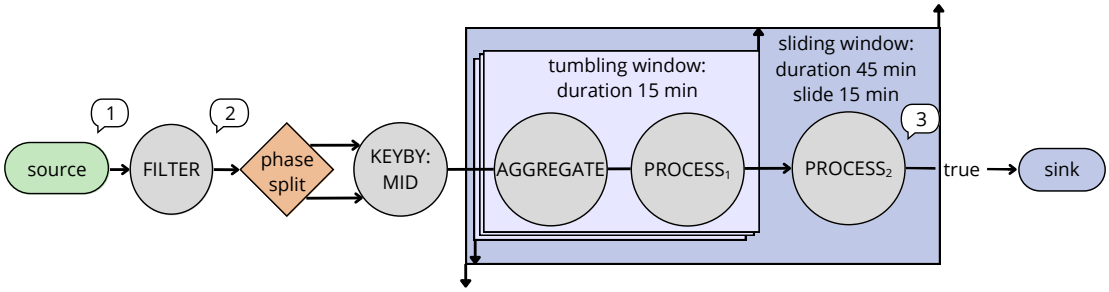


Figure 3.6: Overview of the SP query pipeline for Pattern 2: The key operators include filter, keyBy, a tumbling window, a sliding window, and an aggregate function.

3-phase meters and the other designed for 1-phase meters. The primary distinction lies in whether the pattern matches all voltage phase events or only a single phase event, as illustrated in the earlier example.

Both patterns begin with an initial filter that permits only energy and voltage-related events to pass through to the downstream operators. The 3-phase pattern includes an additional constraint that allows only voltage events with zero values, whereas the 1-phase pattern has no such restriction. This distinction ensures that 3-phase meters are not mistakenly classified as 1-phase meters. In particular, some 3-phase meters may report a zero voltage on phase one while reporting non-zero values on phases two and three, which could otherwise lead to false positives if not properly filtered.

Two filter operators are applied, one for 1-phase and one for 3-phase, allowing the stream to divide into separate branches for their respective pattern checks. In contrast to the SP query, where the phase-split function partitions the stream by meter phases, for the CEP query, the filtering ensures that downstream operators capture only the relevant meter events for validation. The stream is then partitioned by MID using the keyBy operation. After partitioning, the CEP pattern check is applied. The CEP operator seeks three consecutive timestamps in which the voltage values remain zero. These voltage events may arrive in any order within a single timestamp. Additionally, the energy value must remain constant across all three timestamps. The *next()* operator is used to define the sequential temporal constraints between matched events. To avoid overlapping alerts due to interleaved matches, the *"skip past last"* strategy is employed.

Figure 3.7 visualizes the generalized Pattern 2 CEP query for 3-phase and 1-phase meters. The active energy and three-phase voltage events are grouped and represented as *"Events TS_1"*, *"Events TS_2"*, and *"Events TS_3"* to indicate the three consecutive timestamps.

3.1.3 Pattern 3: Over-current Fault

Pattern 3 is designed to detect SM faults caused by over-current. This pattern is characterized by a sequence in which the meter initially records non-zero readings for

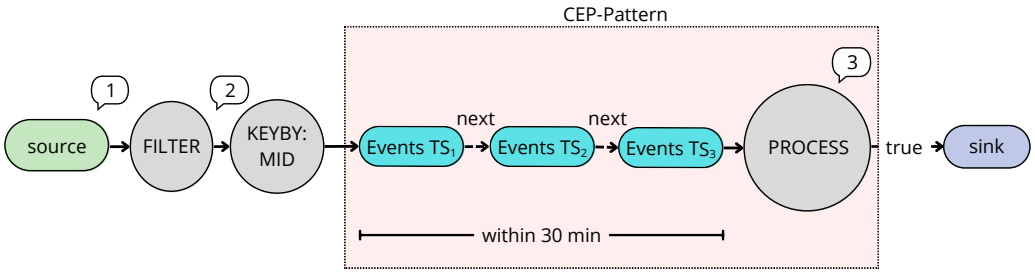


Figure 3.7: Overview of the CEP query Pipeline for Pattern 2: The key operators include filter, keyBy, and a CEP pattern.

voltage and active power, followed by an over-current event that exceeds the meters current limit, and subsequently by a period where those same readings consistently drop to zero.

To construct this pattern, we first group the voltage and active power readings for all phases of a meter by timestamp, as done in Pattern 2. Given that all expected SM-values are present, if the grouped readings contain non-zero values and are followed by a current limit event in a window of 15 minutes, we label the event at that timestamp as 'A'; if they contain only zeros, we label it as 'B'. The target sequence we look for in the data stream is therefore 'A', 'B', 'B'. Including the non-zero values in 'A' in the pattern is crucial, as it indicates that the over-current event likely caused the subsequent zero readings in 'B', rather than being a symptom of a pre-existing issue.

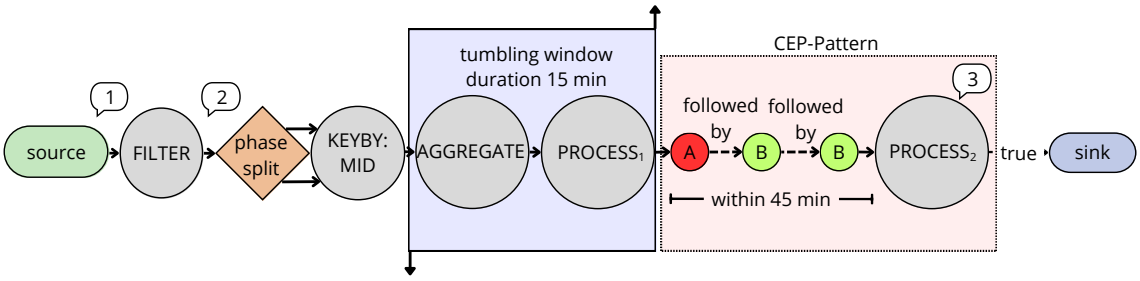


Figure 3.8: Overview of the Pattern 3 query pipeline

In the implementation of the pattern, seen in Figure 3.8, the filter removes all values except voltages, active power, and current overload events. The stream is then split between 1-phase and 3-phase meters as in pattern 2, before using a keyBy to apply the pattern on each SM separately. The aggregate function in the tumbling window then counts the voltages, active power, and current overload events, which reuses functionality from Pattern 2. Process function 1 then uses these counts to emit an output of 'A', 'B', or nothing in accordance with the grouping described above. This is followed by a CEP pattern on the resulting stream from the window outputs to finish up the pattern, where process function 2 issues an alert when 'A' is followed by two outputs of 'B' within 45 minutes. These pipeline choices were made from the observations from earlier patterns that SP more easily handled cases where the

internal ordering of events within a time frame was unimportant, but their relative properties are of interest as in Section 5.1. The choice of using CEP for the second part of the pattern implementation is based on the observations that CEP appeared to have a lower memory load when non-matches could be discarded early, but at the cost of a somewhat higher latency compared to SP, as seen in Section 5.2, as well as the simplicity of the CEP pattern following the initial SP handling.

3.2 Proof of Concept for Templates

To generalize the process of constructing CEP patterns, a simplified approach using templates is proposed. The template mechanism developed in this work serves as a proof of concept. Given that the underlying operations in a CEP query can be chained to create a wide range of combinations, templates help constrain this flexibility by providing reusable structures. Two templates are constructed based on Pattern 1 and Pattern 2 to define the scope of this approach.

Template One corresponds to Pattern 1, where a complex event is triggered when the voltage deviation exceeds a specified threshold. In this template, the quantifying event (*SID*) is defined as a variable, allowing flexibility to accommodate various event types. Its functionality is limited to detecting deviations based on a user-defined threshold, analogous to the logic in Pattern 1.

Template Two follows the structure of Pattern 2 and also uses variables that can be substituted for active energy and phase voltages. This template is designed to evaluate conditions across three consecutive timestamps, allowing substitutions for both energy and voltage events.

4

Evaluation Methodology

This section outlines the evaluation approach used for the three patterns. It begins by defining the evaluation criteria, followed by a description of the experimental setup. The evaluation criteria introduce the key performance metrics used to assess and compare the patterns, including throughput, latency, memory usage, and CPU utilization. The evaluation setup describes the overall system configuration and environment used to conduct the experiments. This includes details on the hardware, software stack, and datasets employed to ensure consistent and reproducible results.

4.1 Evaluation Criteria

This section defines the performance metrics used to evaluate the system and outlines the methodology for comparing the SP and CEP implementations. To ensure consistency and statistical significance in the performance measurements, each query implementation is executed five times.

For Patterns 1 and 2, which include both SP and CEP query implementations, a comparison is conducted to assess their functional equivalence. Rather than simply comparing the total number of alerts generated, the evaluation also verifies that both implementations produce identical alert frequencies per meter ID and compares alert start times to confirm they match, ensuring the correctness and consistency of their outputs.

Following this functional comparison, the system is evaluated using the following four key performance metrics:

- Throughput
- Latency
- CPU Utilization and Memory Usage

The impact of the defined criteria is evaluated under varying levels of parallelism at the data source. This variation in source parallelism is further categorized into two evaluation types: single-job and multi-job evaluation. Detailed discussions of each category are provided in the subsequent sections. This methodology forms the basis for comparing the SP and CEP queries. These criteria are explained in detail in the subsequent sections.

4.1.1 Throughput

Throughput measures the rate at which the system processes incoming data, expressed in events per second (events/s). It is measured at two key points during query execution: at the source and after the filter operation. These measurement points are indicated as (1) and (2), respectively, in the query implementation diagrams referenced in Chapter 3. The average value from the five runs is reported as the final result.

4.1.2 Latency

The latency measurement, often referred to as "system latency", quantifies the time taken for an arriving event to produce a corresponding output. In simple cases, this is the time between ingestion and result generation. However, in complex event scenarios, where a match results from a combination of multiple events, this latency is defined as the time from the arrival of the last contributing event to the generation of the output event. As this measurement is taken with the data already collected on the machine, the largest delay in receiving new data in the running job is related to the reading speed of the source. This does not necessarily correspond to the full latency of a system receiving live data. The system would then also need to wait for both SMs to produce the data and for this data to be delivered. A live system may also require watermark strategies that introduce additional delays to accommodate late and out-of-order arrival of events. The delay incurred as the system waits for input can be referred to as "information latency" and is not measured in this thesis.

The latency measurements are taken using local system time and are expressed in milliseconds (ms). Since all queries use the same system clock, the negligible overhead of reading system time is uniformly applied and thus not considered in the comparison. Measurement points for latency are indicated as (1) and (3) in the diagrams referenced in Chapter 3. Specifically, at point (1), a processing timestamp is inserted into the stream. At point (3), latency is calculated as the difference between the current timestamp and the timestamp of the last arriving event recorded at point (1).

The average latency is calculated across all five runs and presented accordingly. The latency distribution violin plot visualizes the combined latency values from all five runs, providing a comprehensive view of the distribution. Similarly, the latency distribution trends are based on samples from the raw latency data collected from all runs.

4.1.3 CPU Utilization and Memory Usage

CPU utilization reflects the percentage of total available processing power used during query execution. Memory usage indicates the portion of the JVM heap memory consumed, expressed as a percentage of the maximum heap size allocated to the stream processing framework. Both metrics provide insights into the system's resource efficiency under different workloads and configurations.

4.2 Evaluation Setup

The experiments were conducted on a virtual server running Red Hat Enterprise Linux with 9 Intel Xeon Gold 6134M CPU cores, each clocked at 3.20 GHz and 35 GiB of memory. Within the Flink cluster, the JobManager and TaskManager were allocated *1 GiB* and *25 GiB* of memory, respectively, resulting in approximately *20.3 GiB* of available heap memory for task execution. The Flink cluster maximum parallelism is set to 9.

The system leverages the Flink framework as its core SPE. Flink is deployed in session mode as a stand-alone cluster to enable efficient and scalable event processing. The Apache FreeMarker template engine is used to generate the template source code through FTL (FreeMarker Template Language) files.

To evaluate system performance, key metrics, specifically throughput and latency, are reported to Prometheus and visualized using Grafana. These metrics provide a more application-specific insight compared to the default Flink metrics. Figure 4.1 provides a conceptual overview of the system architecture, including the Flink cluster, source, sink, and monitoring tools. The data source can be either a Kafka stream or a bounded file. A match on the evaluating pattern is treated as a complex event, which is captured and emitted as a new event to the sink.

To test patterns with comparable results, the data stream was recorded and stored as a fixed dataset, ensuring uniform input across all pattern implementations and allowing for consistent benchmarking.

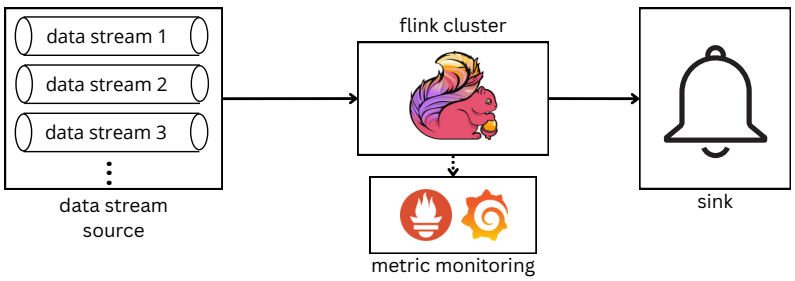


Figure 4.1: Overview of the environment setup: The Flink cluster accesses data streams from either Kafka or file sources, exposes metrics via a Prometheus-Grafana reporter, and outputs alerts to a designated sink.

To accurately assess the quality and performance of the three defined patterns, a consistent and controlled dataset was constructed and used through "replayable" files. This ensures that all pattern implementations process an identical set of input events, enabling fair and reproducible comparisons across different query types. The dataset, shown in Table 4.1, comprises three primary components: *SM-values*, *SM-events*, and *SM-information*. The table details which dataset is used for each pattern. Notably, *SM-values* are utilized across all three patterns. The SP query for Pattern 2 exclusively uses the *SM-information* dataset, while Pattern 3 relies on generated datasets. The tuple structures for *SM-values* and *SM-events* are illustrated in Figure 2.3.

Dataset	Number of records	Used in		
		Pattern 1	Pattern 2	Pattern 3
SM-values	3,230,179,197	X	X	X
Generated-values	7,392,000			X
SM-events	327,302			X
Generated-events	349,986			X
SM-information	275,406		X*	X

Table 4.1: Evaluation dataset overview: The total number of events in each dataset is presented, along with their usage in the respective patterns. *X** indicates that the dataset is used only in the SP query for Pattern 2.

SM-values dataset: This dataset contains periodic event readings from real-world smart meters. The data spans seven days, totalling over 3.2 billion events. To evaluate the effect of source parallelism, the dataset is partitioned into subsets corresponding to parallelism levels of 1, 3, 6, and 9. Each subset includes files equal to its respective parallelism level (e.g., the subset for parallelism 9 includes 9 files), with even distribution of events across the files. After partitioning, each subset is sorted by event timestamp, and duplicate events are removed. This dataset is used in all three patterns.

SM-events dataset: This dataset captures descriptive event messages from SMs, such as over-current, under-voltage, or over-voltage events. These messages vary widely and provide important contextual signals for pattern detection. The SM-events dataset is used specifically in Pattern 3, where such messages are essential for identifying complex event sequences.

SM-information dataset: This dataset includes metadata about SMs, such as their phase type (i.e., 1-phase or 3-phase). This information is useful in queries where distinguishing between meter types affects the interpretation of events. The SM-information dataset is used in Pattern 2 by the SP query and in Pattern 3.

During the evaluation, no valid matches for Pattern 3 were identified within the original dataset, which consisted of seven days of data. To enable meaningful testing, synthetic data, referred to as Generated-values and Generated-events, was created and merged into the corresponding SM-values and SM-events datasets. This generated data, developed under the supervision and approval of a domain expert at GE, spans the same period as the original dataset and includes both 1-phase and 3-phase meters exhibiting anomalous behaviour. These fabricated meters were intentionally designed to deviate from normal operational patterns to simulate realistic, complex event scenarios for validation purposes.

As stated earlier, the evaluation is divided into two categories: single-job and multi-job evaluation. Events carry identifiers, such as MIDs, that can be used for logical grouping. In cases where data streams are already isolated (e.g., specific subsets of MIDs appear only in certain streams) and the output results are also required to be isolated by these identifiers, a multi-job evaluation setup can be more appropriate. This allows the pattern evaluation closer to the SMs, saving bandwidth and

enabling execution on lower resource machines. In contrast, single-job evaluation is necessary when data streams are not pre-isolated by identifiers or when the output must aggregate data across multiple identifiers.

4.3 Single-Job Evaluation

In the single-job evaluations, the implementation is deployed as a single job. In this setup, source parallelism is varied across four levels: 1, 3, 6, and 9. Each job submission to the Flink cluster constitutes a single execution run. The Flink cluster in single-job evaluation is configured with 9 task slots. Additionally, when the job accesses multiple data sources (at parallelism levels 3, 6, and 9), the watermark alignment strategy provided by the Flink framework is employed. This mechanism ensures synchronized event-time progression across all sources, maintaining a consistent processing rate by aligning the watermarks of the parallel source tasks. This synchronization is essential because, without it, downstream operators must wait for all sources to reach the same watermark before producing output, which consumes large amounts of memory.

4.4 Multi-Job Evaluation

In a multi-job evaluation, the implementation is deployed as multiple independent jobs, where each job reads from a single data source. While a source parallelism level of 6 in the single-job evaluation (Section 4.3) corresponds to a single-job reading from six sources, in the multi-job evaluation, it translates to six separate jobs running in parallel, each reading from one distinct source. In this setup, source parallelism is varied across four levels: 3, 6, and 9. Source parallelism level 1 is excluded from the multi-job evaluation, as it is equivalent to the single-job scenario with source parallelism set to 1. The Flink cluster in multi-job evaluation uses a configuration with 81 task slots. As each job runs independently and reads from a single source, watermark alignment is a lesser concern when compared to single-job evaluation.

5

Results and Discussion

The results for the three patterns introduced in Chapter 3 are presented in this section, along with the performance metrics used for evaluation, as outlined in Section 4.1. As previously discussed, the evaluation is divided into single-job and multi-job scenarios. Throughout the experiments, source parallelism is denoted as P**X**, where **X** indicates the degree of parallelism. For each parallelism setting, the job is executed *five* times, and the aggregated results are presented in the corresponding tables for each pattern.

The single-job evaluations compare the performance of both SP and CEP queries across source parallelism levels of P1, P3, P6, and P9. In contrast, the multi-job evaluations are conducted for P3, P6, and P9 only. In the single-job evaluation, the number of input sources is varied, whereas in the multi-job evaluation, the number of concurrently running jobs with one source each is varied. The number of sources is adjusted to facilitate parallel input reading, thereby accommodating variable data rates at the source.

As discussed in previous sections, the dataset used in this evaluation is sorted by event time, and out-of-order events are not considered in the results presented here. Handling out-of-order data typically requires increased memory capacity to accommodate delayed watermark progression. Therefore, to maintain consistency and reduce resource overhead, such events are excluded from this analysis. For each pattern, results are reported separately for both cases, with a discussion after each section. The memory usage plots represent memory consumption over time, where each point on the x-axis corresponds to a tumbling window average over 5% intervals of the total runtime. The latency trend plots are generated using a sample of the overall data points to avoid visual clutter. The samples are evenly distributed and selected using the *DataFrame.sample()* method from the Python pandas library.

5.1 Pattern 1: Voltage Deviation

The following sections present the results of the single-job and multi-job executions for Pattern 1, as previously described in Section 3.1.

The tests were conducted using the SM-values dataset, as detailed in Table 4.1. To ensure consistency, the outputs of the CEP and SP queries were compared. The queries took between 25-140 minutes to execute over the data set and generated

close to forty-five thousand alerts.

Throughout the tests, the SP query is run with parallelism 1 for the operators. For the CEP query, the operator parallelism is set to the same as the source parallelism.

5.1.1 Single-Job Results for Pattern 1

This section focuses on the results of Pattern 1 under single-job execution.

Table 5.1: Pattern 1: Single-job summary of source and filter throughput, per-thread throughput, and average latency of SP and CEP queries across parallelism levels.

	SP				CEP			
Source Parallelism	1	3	6	9	1	3	6	9
Source Throughput per Thread (events/s)	423,347	190,344	151,345	117,732	393,686	170,381	151,776	110,937
Total Source Throughput (events/s)	423,347	571,033	908,068	1,059,589	393,686	511,141	910,657	998,433
Filter Throughput per Thread (events/s)	60,001	26,838	21,600	16,791	55,834	24,108	21,727	15,853
Total Filter Throughput (events/s)	60,001	80,513	129,601	151,121	55,834	72,324	130,359	142,676
Average Latency (ms)	624	842	961	1,007	4,379	2,383	1,411	1,339

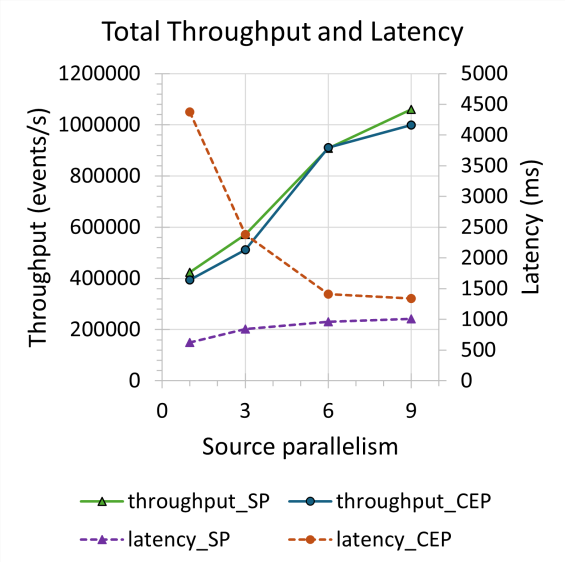


Figure 5.1: Pattern 1: Single-job throughput and average latency of SP and CEP queries.

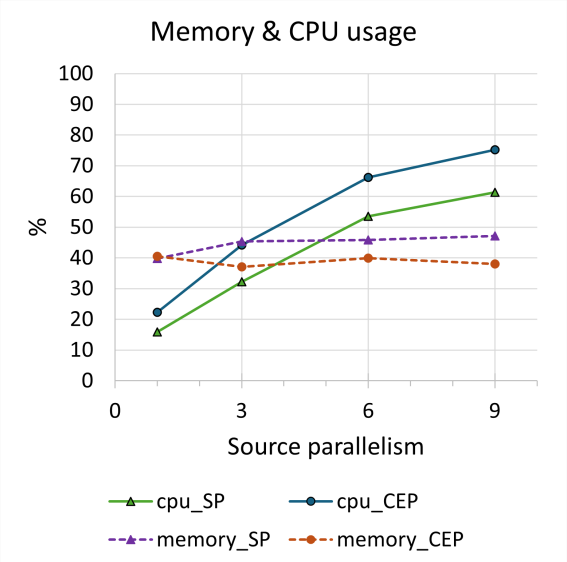
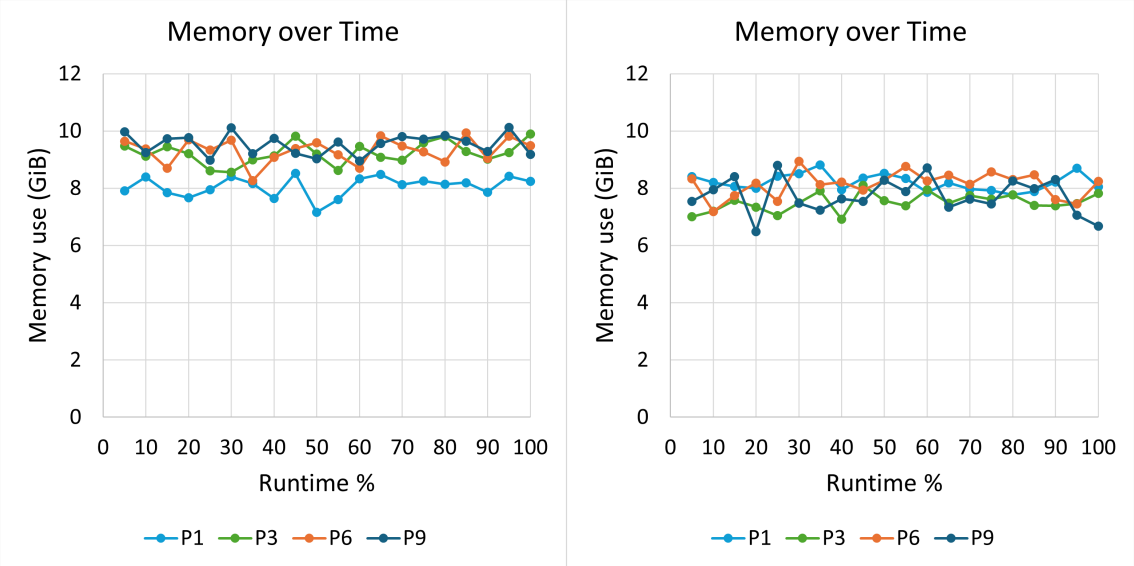


Figure 5.2: Pattern 1: Single-job memory and CPU utilization of SP and CEP queries.



(A) SP (B) CEP

Figure 5.3: Pattern 1: Single-job memory usage over time for varying levels of source parallelism. Each point on the x-axis represents the average memory usage over a consecutive 5% segment of the job's total runtime. (A) shows the SP query; (B) shows the CEP query.

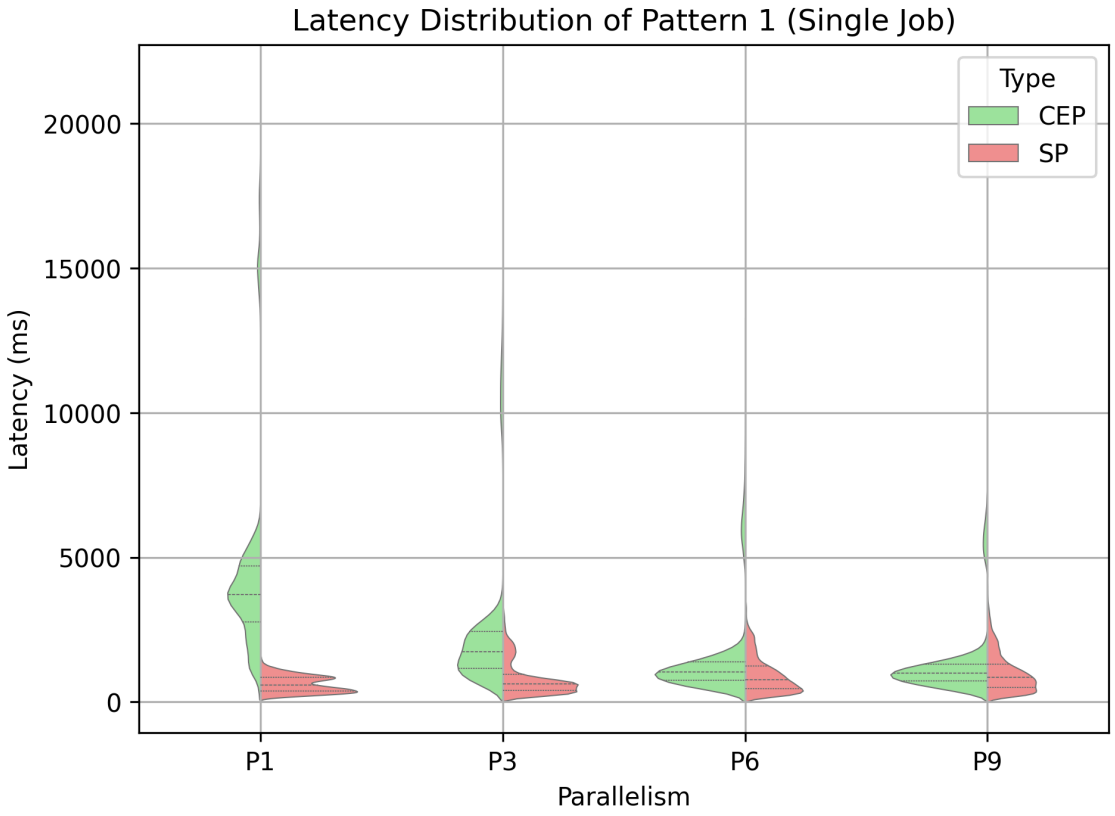


Figure 5.4: Pattern 1: Single-job latency distribution across parallelism levels for CEP and SP query.

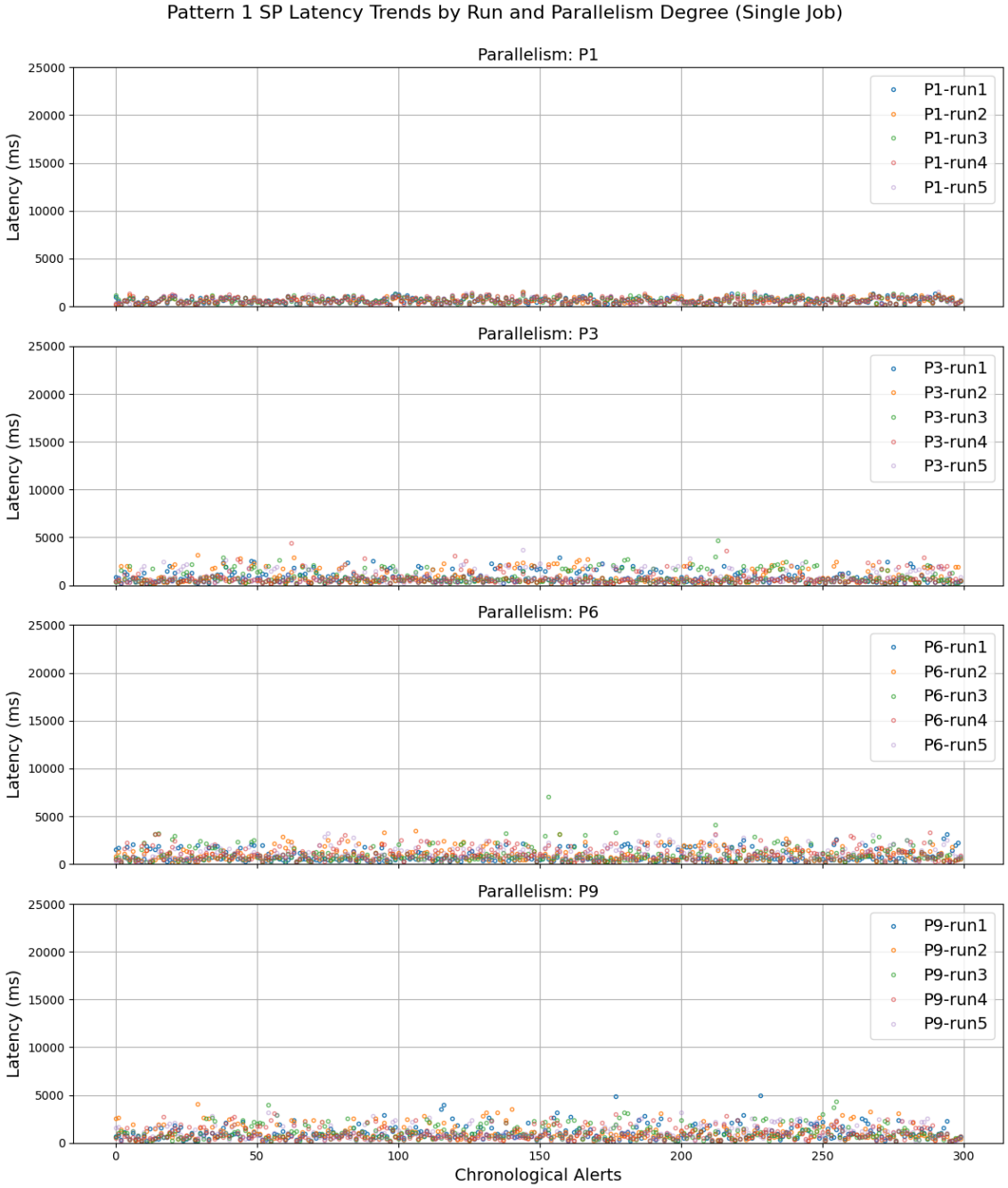


Figure 5.5: Pattern 1: Single-job SP query latency trend as a function of run number and degree of parallelism for the SP query of Pattern 1. The plot displays a representative sample of 300 points.

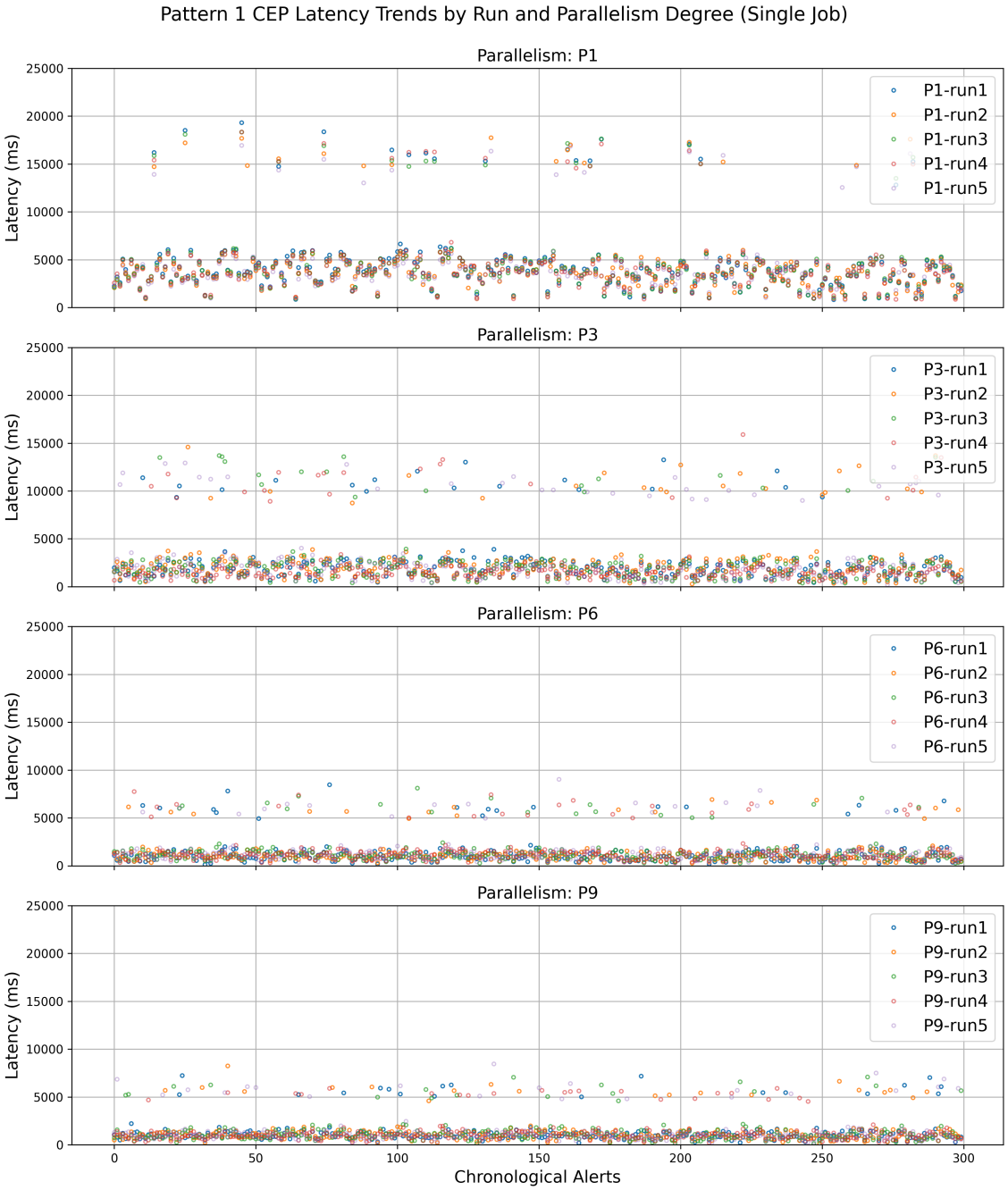


Figure 5.6: Pattern 1: Single-job SP query latency trend as a function of run number and degree of parallelism for the CEP query. The plot displays a representative sample of 300 data points.

The aggregated results for each source parallelism level in the single-job execution are presented in Table 5.1. This data is further visualized in Figure 5.1 for the throughput and latency of both queries. The throughput is seen to increase with higher source parallelism, with P3 and P6 achieving notably higher throughput. For the SP query, average latency shows a slight increase with higher parallelism. In contrast, the CEP query exhibits significantly higher latency at P1, approximately six times that of the SP query. However, as source parallelism increases, the average latency of the CEP query decreases. By P9, the CEP latency has reduced by a factor of three compared to P1 and becomes comparable to the latency of the SP query.

Figure 5.2 presents the average CPU utilization and memory usage. Across all levels of parallelism, the CEP query consistently exhibits higher CPU utilization compared to the SP query. Conversely, the SP query demonstrates higher memory usage across different parallelism levels. In summary, the CEP query consumes more CPU but less memory, while the SP query shows the opposite trend.

Figure 5.3 panels (A) and (B) illustrate memory usage over the total execution time. The CEP query maintains a slightly lower memory footprint than the SP query. While memory usage for the CEP query remains relatively stable across all parallelism levels, the SP query shows a noticeable increase in memory usage at higher parallelism, despite having similar usage to CEP at P1.

Figure 5.4 illustrates the latency distribution for both SP and CEP queries using violin plots. Examining the SP query, the latency is tightly confined and does not exceed 1 second. The CEP query, on the other hand, shows a broader latency distribution, although the majority of values remain within the range of 2 to 3 seconds; it also includes latencies reaching up to 20 seconds. For a more detailed view of the density, particularly at the higher end of the latency range, we refer readers to Section A.1, which includes an overlaid scatter plot on the violin plot.

Figure 5.5 and Figure 5.6 depict a sample of 300 latency values to highlight the general latency trends over time. Notably, the CEP query exhibits a bimodal distribution across different levels of parallelism, forming two distinct latency bands. While the cause of this behaviour is unknown, a few possibilities have been ruled out. There appears to be no connection to the parallelism degree of the job. Any connection to the SM type has also been ruled out, as the pattern only targets the 3-phase meters. For a complete visualization including all latency points, please refer to Section A.1.

5.1.1.1 Discussion of Single-Job Results for Pattern 1

The single-job evaluation results indicate that the throughput of both SP and CEP queries increases with higher levels of parallelism. While SP achieves slightly higher throughput than CEP, the difference is not significant. Memory usage for both implementations remains relatively stable across parallelism levels, with CEP showing marginally lower memory consumption and higher CPU utilization. This may be attributed to CEPs ability to discard non-matching events early in the pipeline, whereas SP continues processing all incoming data regardless of relevance.

A more notable distinction between the two implementations emerges in latency trends. SP maintains relatively consistent latency across different levels of parallelism, whereas CEP shows a significant reduction in latency as parallelism increases.

Further analysis reveals that CEP exhibits a wider distribution in latency, characterized by distinct gaps, in contrast to the more uniform latency observed in SP. However, this variability in CEP diminishes with increased parallelism.

5.1.2 Multi-Job Results for Pattern 1

This section focuses on the results of Pattern 1 under multi-job execution.

Table 5.2: Pattern 1: Multi-job summary of source and filter throughput, per-thread throughput, and average latency of SP and CEP queries across parallelism levels.

	SP			CEP		
Source Parallelism	3	6	9	3	6	9
Source Throughput per Job (events/s)	368,071	329,134	229,900	316,362	210,363	139,612
Total Source Throughput (events/s)	1,104,213	1,974,806	2,069,100	949,085	1,262,177	1,256,505
Filter Throughput per Job (events/s)	52,401	46,908	32,579	45,167	29,926	19857
Total Filter Throughput (events/s)	157,203	281,446	293,208	135,501	179,558	178,716
Average Latency (ms)	336	271	292	1,693	1,379	1,504

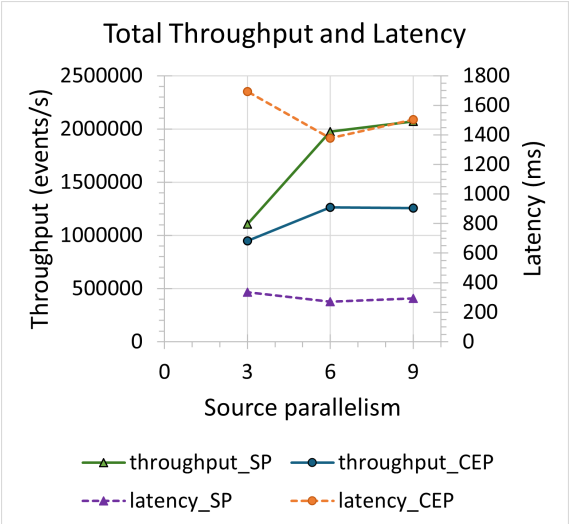


Figure 5.7: Pattern 1: Multi-job throughput and average latency of SP and CEP queries.

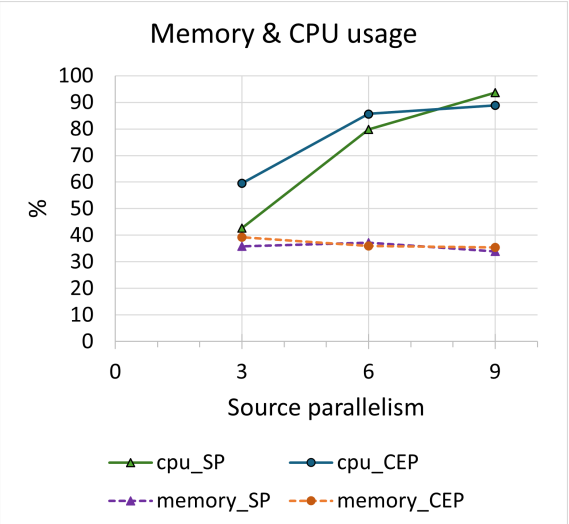
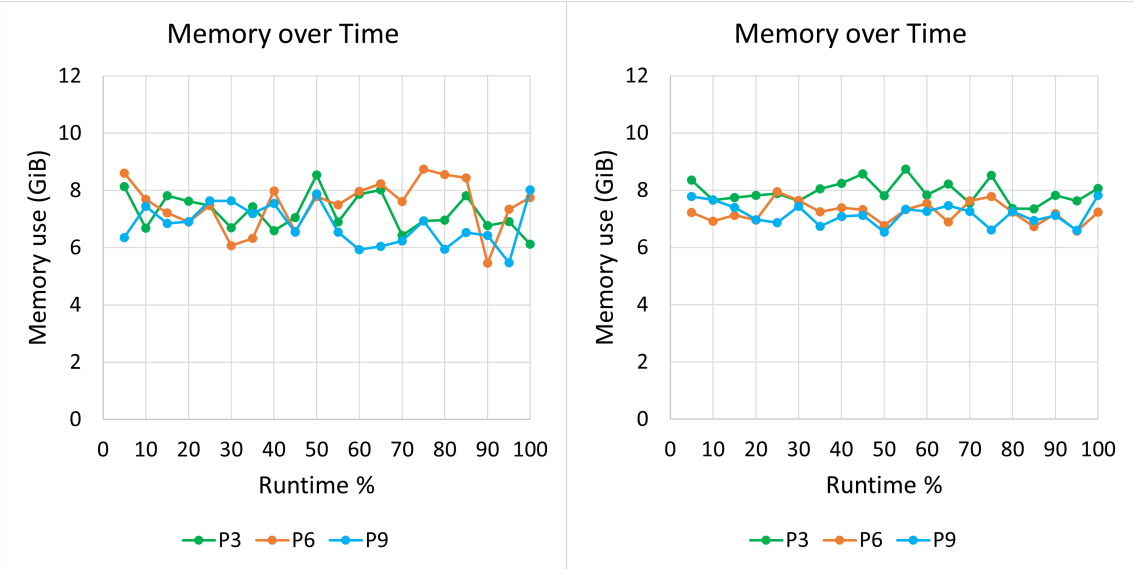


Figure 5.8: Pattern 1: Multi-job memory and CPU utilization of SP and CEP queries.



(A) SP (B) CEP

Figure 5.9: Pattern 1: Multi-job memory usage over time for varying levels of source parallelism. Each point on the x-axis represents the average memory usage over a consecutive 5% segment of the job's total runtime. (A) shows the SP query; (B) shows the CEP query.

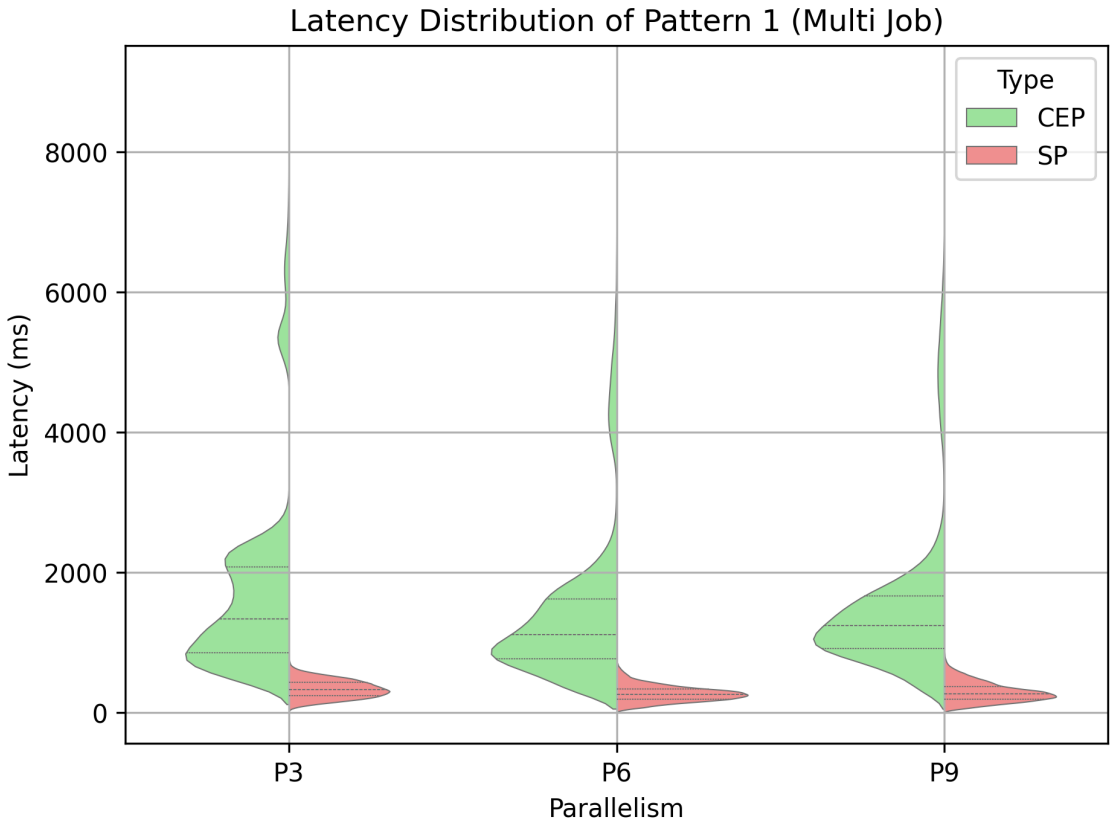


Figure 5.10: Pattern 1: Multi-job latency distribution across parallelism levels for CEP and SP query.

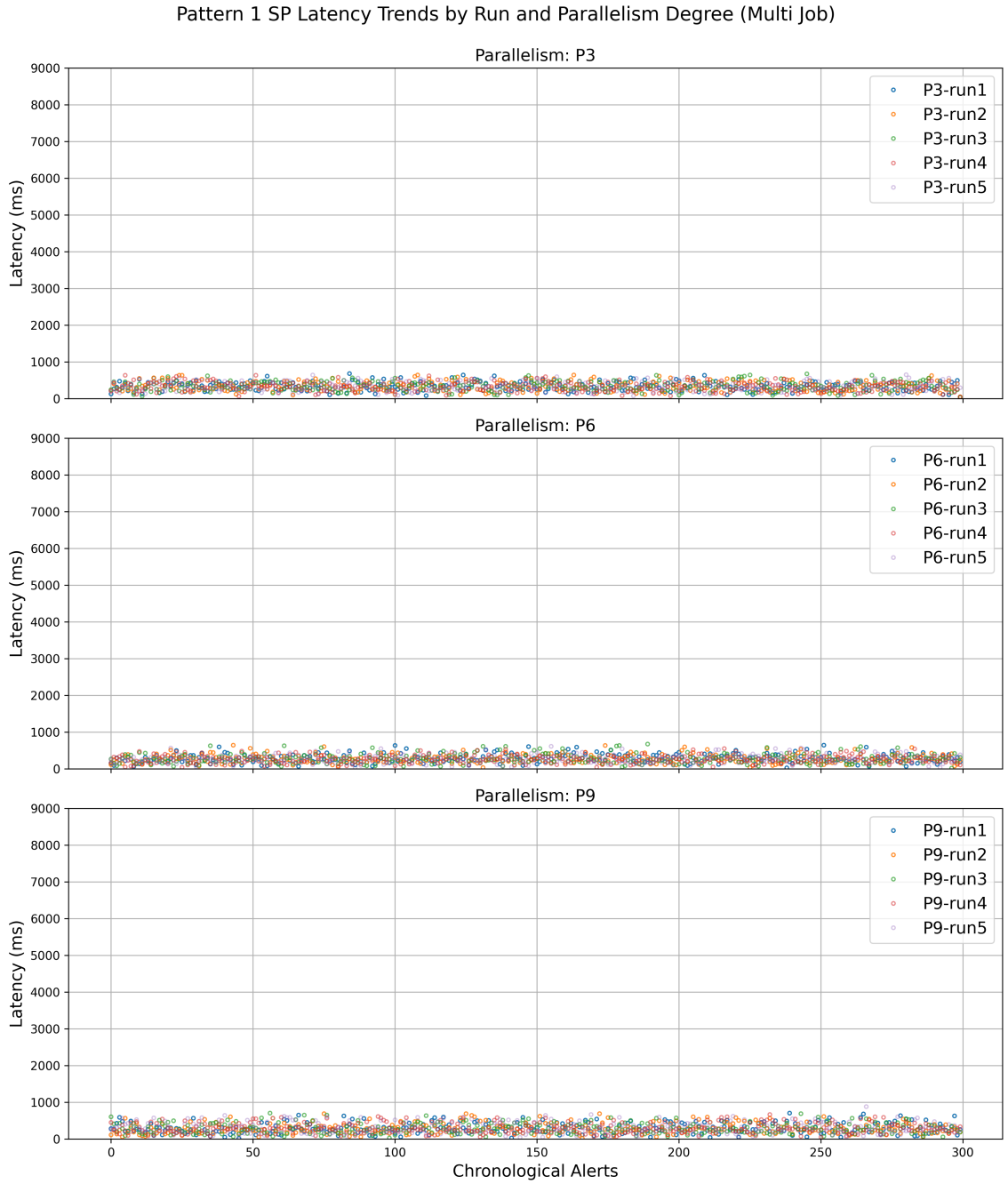


Figure 5.11: Pattern 1: Multi-job SP query latency trend as a function of run number and degree of parallelism for the SP query of Pattern 1. The plot displays a representative sample of 300 data points.

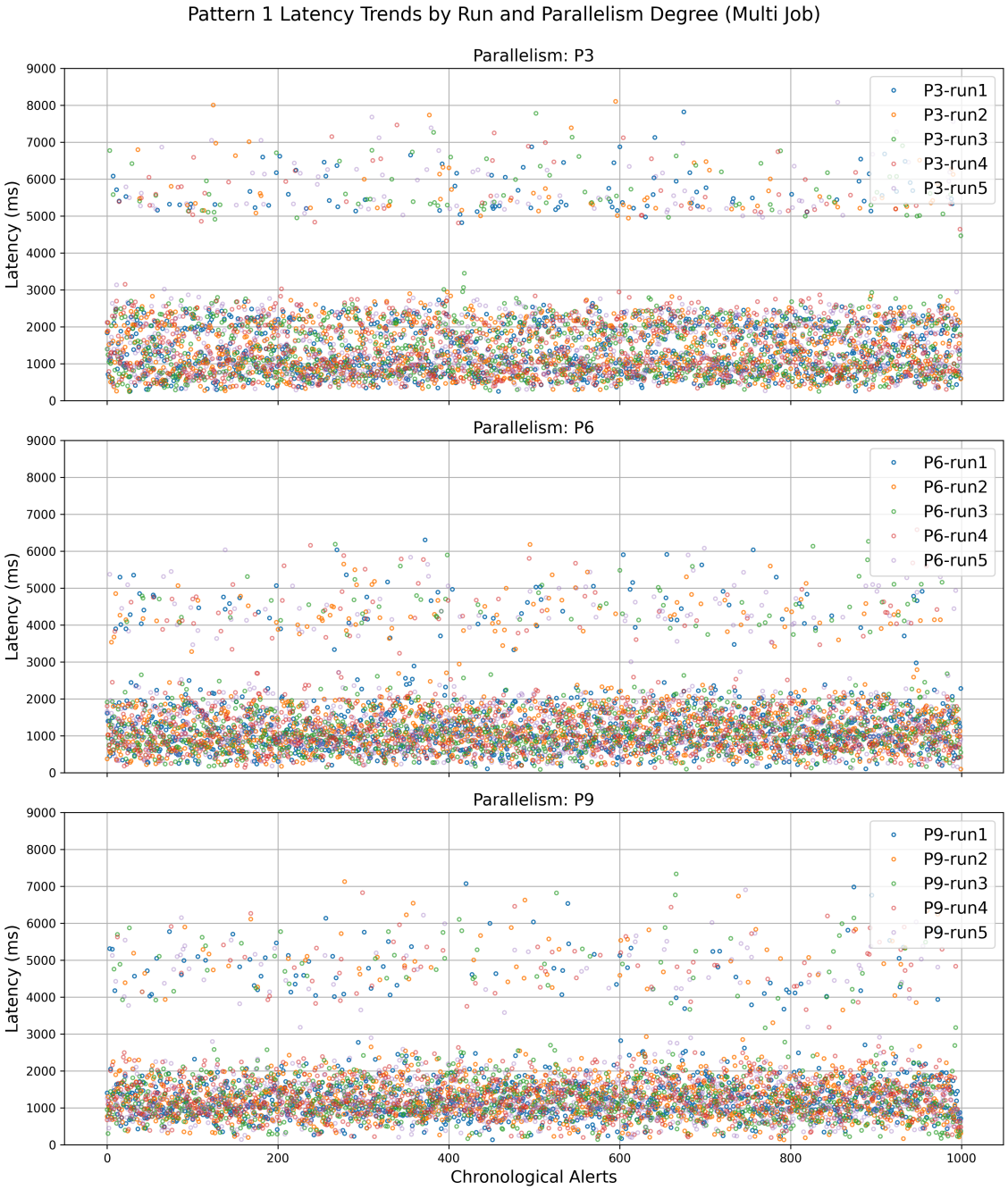


Figure 5.12: Pattern 1: Multi-job SP query latency trend as a function of run number and degree of parallelism for the CEP query. The plot displays a representative sample of 1000 data points.

The aggregated results at each source parallelism level are presented in Table 5.2. These results, including throughput, average latency, CPU utilization, and memory usage, are visualized in Figure 5.7 and Figure 5.8.

Figure 5.7 illustrates that throughput increases for both CEP and SP queries as parallelism increases, stabilizing around P6 and P9. Overall, SP achieves higher throughput than CEP. In terms of average latency, the SP query consistently outperforms the CEP query, with approximately four times lower latency. For the CEP query, latency is notably higher at P1 but decreases significantly at P6 and P9.

Figure 5.8 shows an expected trend in CPU utilization, which increases with higher parallelism. At P9, SP demonstrates higher CPU utilization compared to CEP. However, at other parallelism levels, SP generally uses less CPU than CEP. Memory usage remains stable and nearly identical for both SP and CEP across all configurations. Figure 5.9 illustrates memory usage over time for both queries, which remains consistent within a range of 6 to 9 GiB throughout the evaluation.

Figure 5.10 presents the latency distributions for CEP and SP queries. The CEP query exhibits a broader distribution, while SP latency values are tightly clustered. For a more detailed view, we refer the reader to Section A.2, which overlays individual latency points on the violin plot.

To highlight latency trends over time, Figure 5.11 and Figure 5.12 display a sample of 300 and 1000 latency points, respectively. The SP query maintains a tightly clustered latency pattern, while the CEP query reveals two distinct latency clusters.

5.1.2.1 Discussion of Multi-Job Results for Pattern 1

Contrasting the multi-job evaluation results with the single-job results presented in Section 5.1.1 and Section 5.1.2, a notable difference is observed in throughput. In the multi-job evaluation, the throughput of SP and CEP queries diverges significantly. For example, at P9, the SP query achieves a throughput 64% higher than that of the CEP query. In contrast, during the single-job evaluation, both queries deliver throughput within a similar range. Overall, both SP and CEP demonstrate substantially better throughput performance in the multi-job evaluation compared to the single-job setting. Another key improvement observed in the multi-job evaluation is in average latency. The SP query shows a significant reduction in latency, while the CEP query displays a modest improvement at P6 and P9 relative to the single-job evaluation.

CPU utilization across both single-job and multi-job evaluations reveals an interesting trend. In the single-job evaluation, CPU utilization remains relatively low, with a maximum observed utilization of 75%. In contrast, the multi-job evaluation exhibits significantly higher CPU usage, reaching up to 95% at P9. One possible explanation for this increase is the reduced idle time for operators due to continuous watermark progression. Without waiting periods, operators remain actively engaged in processing events, contributing to higher throughput. Additionally, in Flink, each job runs as a separate JVM process with multiple tasks. Therefore, executing multiple jobs results in more JVM processes, which in turn leads to increased CPU

utilization. Interestingly, memory usage in the multi-job evaluation is slightly lower than in the single-job evaluation, which might also be attributed to the reduced waiting periods for watermark progression. Overall, memory consumption remains relatively consistent across both scenarios.

The latency trends in the multi-job evaluation exhibit a similar pattern to the single-job results: CEP queries display two distinct latency clusters, while SP queries show a single, tightly clustered distribution.

5.2 Pattern 2: Constant Active Energy with Zero Voltage Consumption

The Pattern 2 CEP and its corresponding SP query follow the same methodology as in Pattern 1. The SP query leverages both *SM-values* and *SM-info*, whereas the CEP query requires only the *SM-values* dataset. Details of the datasets used are provided in Table 4.1. Pattern detection for both one-phase and three-phase meters is performed within the same job for both the SP and CEP queries, in both the single-job and multi-job evaluations.

As discussed in Section 3.1, the CEP patterns differ between 1-phase and 3-phase meters, necessitating extensive tuning of parallelism settings within the CEP operator across the implementation pipelines. The CEP operator was evaluated at parallelism levels of 6 and 9. The SP implementation had parallelism equal to the source parallelism throughout the pipeline. Based on performance observations, a parallelism level of 6 was identified as the optimal configuration for both one-phase and three-phase CEP patterns.

To avoid generating overlapping alerts, the CEP query employs the *"skip past last"* strategy presented in Section 2.5. In contrast, the SP query does not use this strategy, resulting in a different frequency of output alerts between the two queries. To enable a meaningful comparison and evaluate the equivalence of the outputs, a transformation function is defined as follows:

$$\text{cep\_alert} = \left\lfloor \frac{\text{sp\_alert} + 2}{3} \right\rfloor$$

The transformation function maps time-wise continuous SP alerts to their corresponding expected CEP alerts. This function is constructed based on the assumption that the alerts produced by the SP query serve as the reference against which CEP alerts are evaluated. The number of expected CEP alerts produced by this function is then compared against the actual number of CEP alerts. The queries took between 27-258 minutes to execute over the data set and converted to CEP alerts, generating close to two thousand alerts.

5.2.1 Single-Job Results for Pattern 2

This section focuses on the results of Pattern 2 under single-job execution.

Table 5.3: Pattern 2: Single-job summary of source and filter throughput, per-thread throughput, and average latency of SP and CEP queries across parallelism levels.

	SP				CEP			
Source Parallelism	1	3	6	9	1	3	6	9
Source Throughput per Thread (events/s)	391,001	174,840	144,127	104,828	214,358	179,189	133,751	88,487
Total Source Throughput (events/s)	391,001	524,519	864,764	943,453	214,358	537,566	802,507	796,385
Filter Throughput per Thread (events/s)	22,152	9,855	8,145	5,938	42,666	35,357	26,645	17,633
Total Filter Throughput (events/s)	22,152	29,565	48,872	53,440	42,666	106,071	159,870	158,693
Average Latency (ms)	1,596	1,738	1,365	1,487	3,032	1,533	1,369	1,644

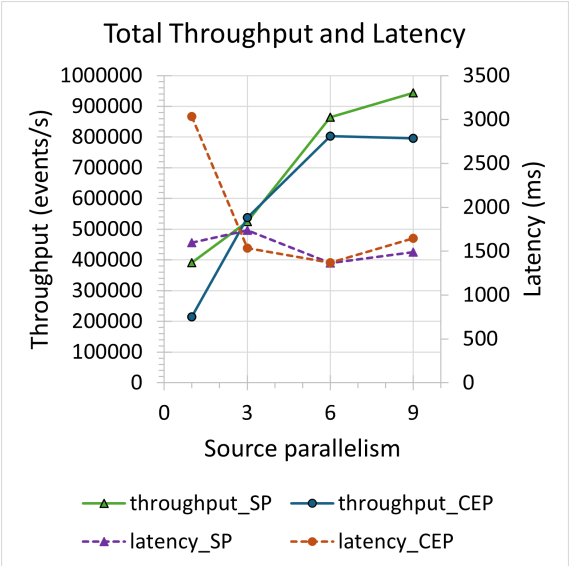


Figure 5.13: Pattern 2: Single-job throughput and average latency of SP and CEP queries.

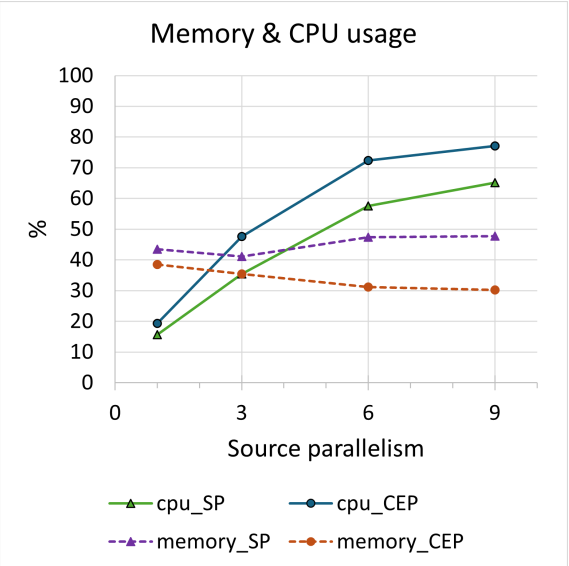
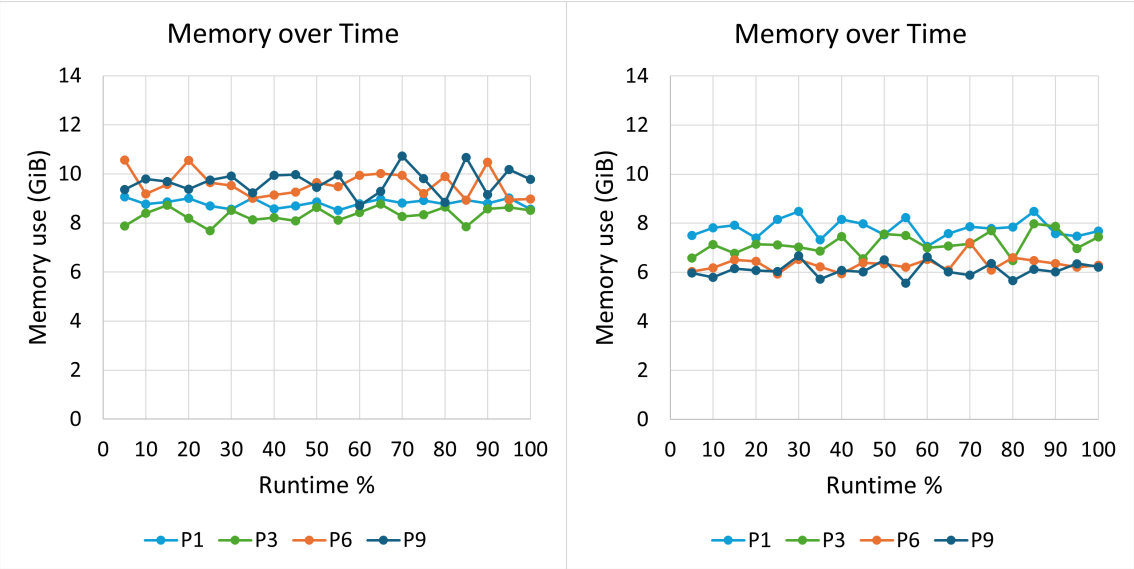


Figure 5.14: Pattern 2: Single-job memory and CPU utilization of SP and CEP queries.



(A) SP (B) CEP

Figure 5.15: Pattern 2: Single-job memory usage over time for varying levels of source parallelism. Each point on the x-axis represents the average memory usage over a consecutive 5% segment of the job's total runtime. (A) shows the SP query; (B) shows the CEP query.

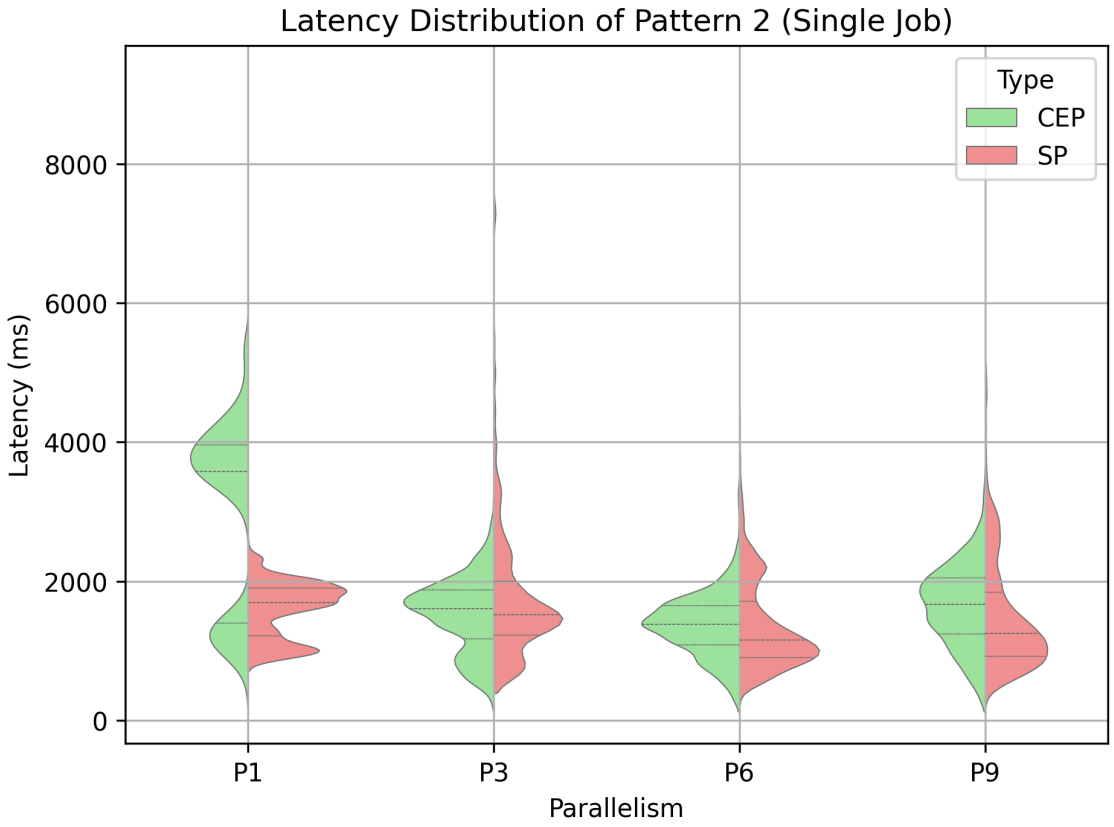


Figure 5.16: Pattern 2: Single-job latency distribution across parallelism levels for CEP and SP query.

Figure 5.17: Pattern 2: Single-job SP query latency trend as a function of run number and degree of parallelism for the SP query of Pattern 2. The plot displays a representative sample of 1000 data points.

Figure 5.18: Pattern 2: Single-job SP query latency trend as a function of run number and degree of parallelism for the CEP query. The plot displays a representative sample of 400 data points.

Table 5.3 presents the total throughput, per-thread throughput at the source and filter operators, and average latency for both CEP and SP implementations under varying levels of source parallelism.

Figure 5.13 illustrates the total throughput and average latency for the CEP and SP implementations. At a P1, the SP implementation achieves nearly double the throughput of CEP, while the latency for CEP is approximately twice as high as that of SP. At P3, the throughput becomes comparable between the two implementations, with CEP latency decreasing significantly, falling below that of SP. For P6 and P9, throughput continues to increase for both CEP and SP, and their latencies converge to similar values.

Figure 5.14 presents the memory and CPU resource utilization for both implementations. Across all levels of parallelism, CPU utilization is slightly higher for CEP compared to SP. In contrast, SP consistently exhibits higher memory usage than CEP.

Figure 5.15 illustrates memory usage over time for varying levels of source parallelism. Both plots, (A) CEP and (B) SP, demonstrate stable memory usage throughout execution, with no significant spikes. As also shown in Figure 5.14 and Figure 5.15, the CEP implementation consistently exhibits slightly lower memory consumption compared to SP. Across all levels of parallelism, CEP memory usage ranges from 6 to 8 GiB, while SP ranges from 8 to 10 GiB.

Figure 5.16 illustrates the latency distribution of all generated alerts for both CEP and SP across varying levels of parallelism. At P1, CEP latency values are more widely distributed, reaching up to 6 seconds, while SP latency reaches a maximum of around 2 seconds. At P3, the latency distributions of CEP and SP appear similar, although SP includes a few outliers extending up to 8 seconds. At P6 and P9, both implementations exhibit similarly clustered latency distributions, with a few outliers in SP reaching up to 6 seconds. For a clearer view of individual data points overlaid on the violin plots, refer to Appendix Section A.3.

Figure 5.18 illustrates a sampled latency trend of the CEP pattern based on alerts issued at each level of source parallelism. Latency values are sorted by alert creation timestamps, revealing the temporal progression of latency throughout the execution. At P1, latencies appear to cluster into two distinct bands, either between 1-2 seconds or 35 seconds. At higher parallelism levels, latency values are more consistently distributed within the 1 to 3-second range, without any noticeable spikes. A notable pattern emerges between alerts 750 and 1100, where latency remains nearly constant, forming distinct horizontal lines that may indicate stable system performance during this interval.

Figure 5.17 presents the corresponding sampled latency trends for the SP implementation. At higher parallelism levels, specifically P3 and P9, certain runs display elevated latency not observed in other runs. Overall, for P3, P6, and P9, most latencies remain within 4 seconds, though occasional outliers contribute to higher latency spikes in specific runs. Similar to CEP, the SP exhibits an interesting latency pattern around 2000 and 3000 alerts. This behaviour is further examined in

the discussion section.

5.2.1.1 Discussion of Single-Job Results for Pattern 2

The single-job evaluation results demonstrate that the throughput of both SP and CEP queries increases with higher levels of parallelism. While SP achieves slightly higher throughput overall, the throughput values match at P3. Consistent with the observations from Pattern 1, the average latency of the CEP query begins higher at P1 and decreases with increasing parallelism. At P3, CEP exhibits lower average latency than SP, though this slightly reverses at P9, where SP marginally outperforms CEP. Nonetheless, from P3 to P9, the average latencies of both queries remain within a comparable range.

CPU utilization and memory usage trends in Pattern 2 also mirror those observed in Pattern 1. The CEP query consistently shows higher CPU utilization and lower memory consumption, while SP demonstrates the opposite behaviour, lower CPU usage but higher memory consumption.

As mentioned above, the latency trend plots reveal a visual artifact in the form of horizontal lines or patches for a period of the alerts. This indicates periods where multiple consecutive alerts exhibit similar latency values. These lines might be caused by batches of events being emitted simultaneously, potentially due to watermark progression or from events arriving in close succession. Since alerts are plotted in chronological order, the appearance of these bands suggests that several alerts were generated at the same moment as the watermark advanced. The consistent presence of these bands across different levels of source parallelism further supports the likelihood that alerts associated with certain event batches are output collectively, resulting in this visual artifact. Moreover, as the source parallelism seems to vary the length of the individual lines and to some extent dampen the effect, it may be caused by a period containing a larger number of meters that exhibit Pattern 2. Increasing the source parallelism then likely introduces distortion and reduces the number of afflicted meters placed in the same source.

5.2.2 Multi-Job Results for Pattern 2

This section focuses on the results of Pattern 2 under multi-job execution.

Table 5.4: Pattern 2: Multi-job summary of source and filter throughput, per-thread throughput, and average latency of SP and CEP queries across parallelism levels.

	SP			CEP		
Source Parallelism	3	6	9	3	6	9
Source Throughput per Thread (events/s)	365,898	328,829	204,061	271,790	179,609	115,168
Total Source Throughput (events/s)	1,097,694	1,972,972	1,836,550	815,369	1,077,654	1,036,516
Filter Throughput per Thread (events/s)	20,781	18,688	11,567	53,988	35,575	22,842
Total Filter Throughput (events/s)	62,344	112,126	104,103	161,964	213,449	205,575
Average Latency (ms)	664	486	546	1,111	998	1,103

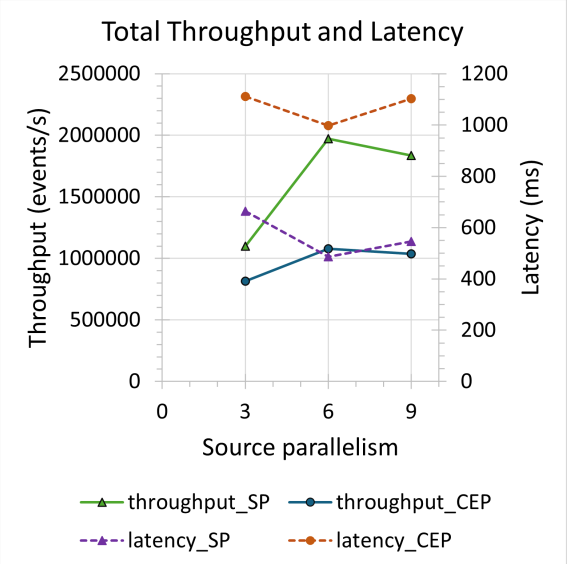


Figure 5.19: Pattern 2: Multi-job throughput and average latency of SP and CEP queries.

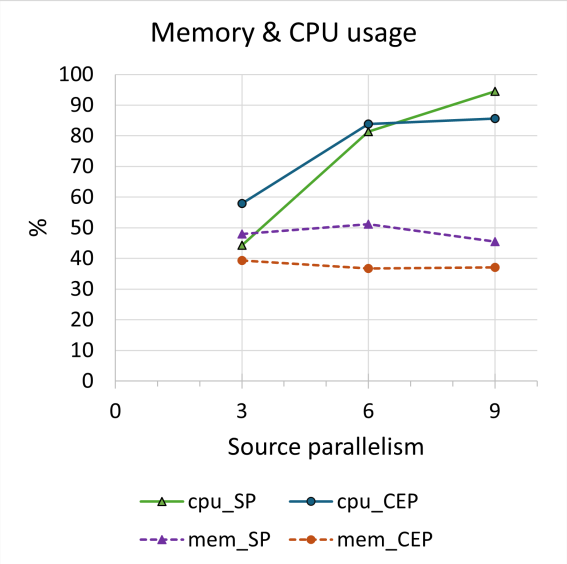
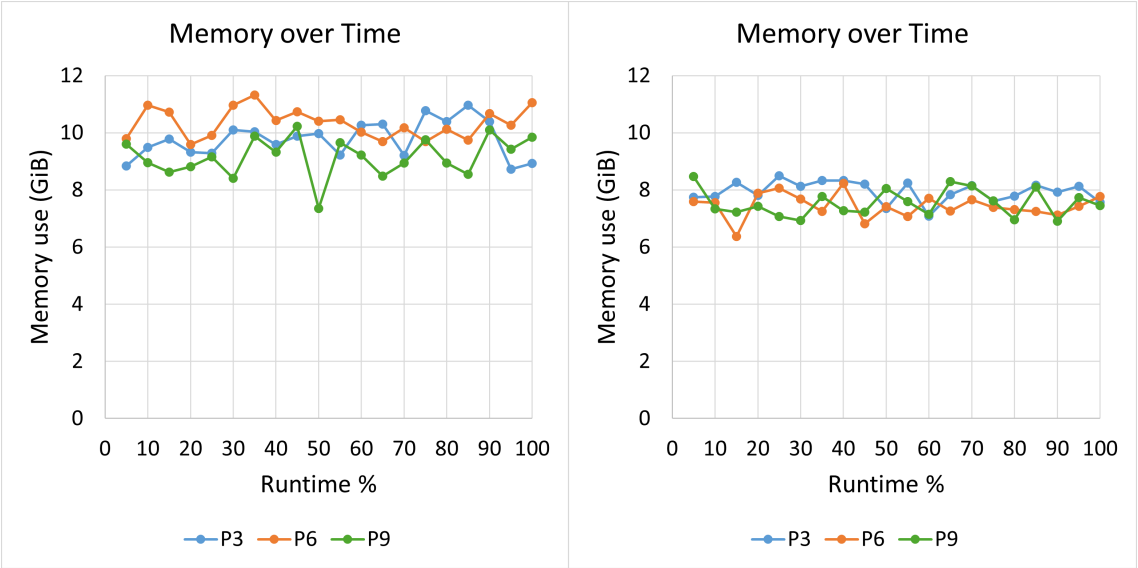


Figure 5.20: Pattern 2: Multi-job memory and CPU utilization of SP and CEP queries.



(A) SP (B) CEP

Figure 5.21: Pattern 2: Multi-job memory usage over time for varying levels of source parallelism. Each point on the x-axis represents the average memory usage over a consecutive 5% segment of the job's total runtime. (A) shows the SP query; (B) shows the CEP query.

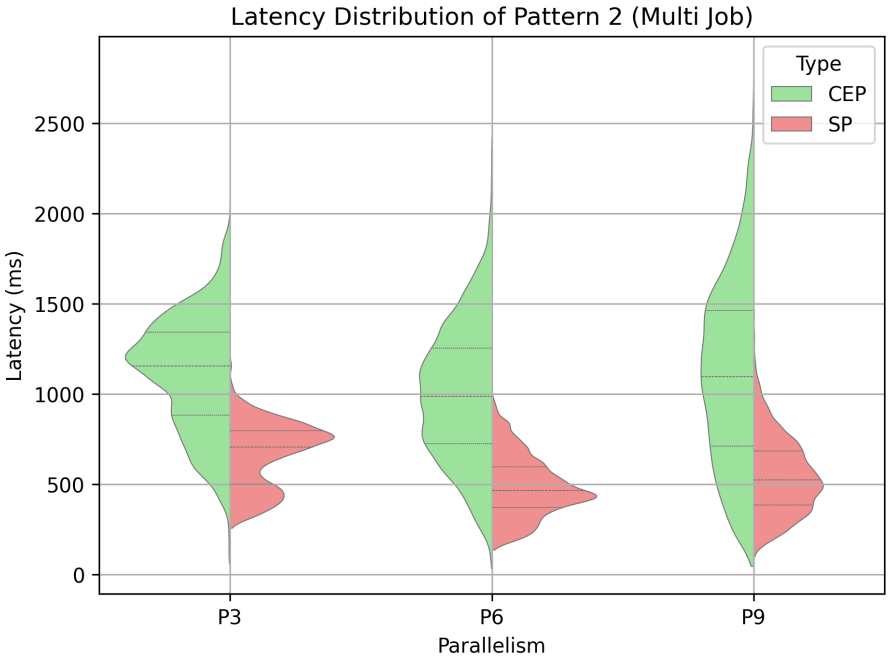


Figure 5.22: Pattern 2: Multi-job latency distribution across parallelism levels for CEP and SP query.

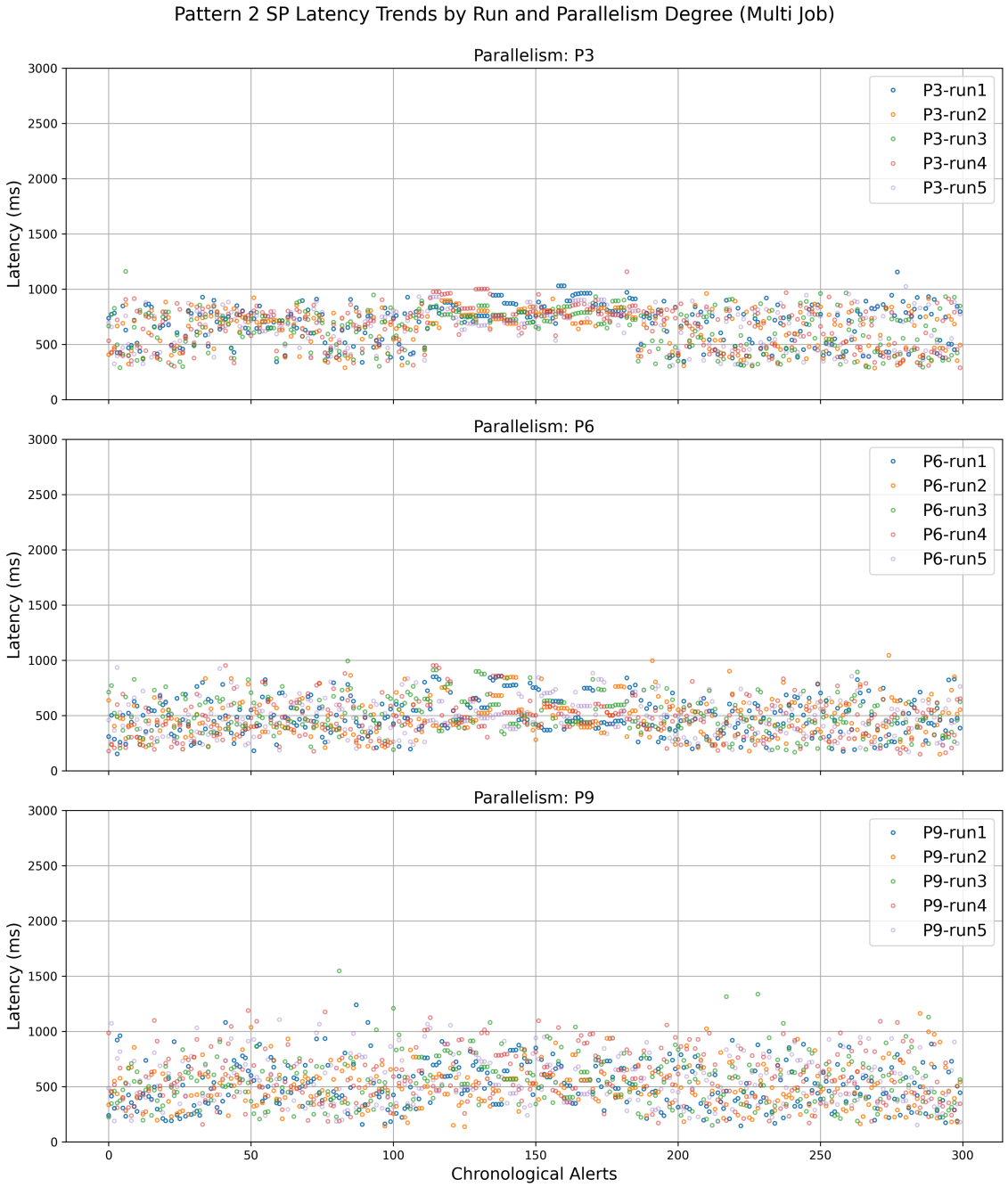


Figure 5.23: Pattern 2: Multi-job SP query latency trend as a function of run number and degree of parallelism for the SP query of Pattern 2. The plot displays a representative sample of 300 data points.

Figure 5.24: Pattern 2: Multi-job SP query latency trend as a function of run number and degree of parallelism for the CEP query. The plot displays a representative sample of 300 data points.

Table 5.4 presents the total throughput, per-thread throughput at the source and filter operators, and average latency for both CEP and SP implementations across varying levels of source parallelism. Figure 5.19 visualizes the throughput and average latency data from the table. The SP query outperforms the CEP query in throughput, delivering approximately twice the throughput at P6 and P9. Similarly, the SP query maintains an average latency that is roughly half of the CEP query's across all levels of parallelism.

Figure 5.20 illustrates CPU utilization and memory usage across different parallelism levels. CPU utilization for both queries is comparable and follows a similar trend. However, CEP consistently exhibits lower memory usage than SP. This trend is further confirmed by Figure 5.21, which shows that SPs memory usage remains higher throughout execution, regardless of parallelism.

Figure 5.22 presents the latency distribution for both queries. The CEP query demonstrates a broader distribution, indicating more variability, whereas the SP query shows a tighter, more clustered distribution.

Figure 5.23 and Figure 5.24 display the chronological trends of alert latencies for SP and CEP queries, respectively, based on sampled data points. Notably, both plots reveal a visually distinct mid-period, between alerts 2,000 and 3,000 for SP, and between 750 and 1,250 for CEP, which is discussed in more detail in the subsequent discussion section.

5.2.2.1 Discussion of Multi-Job Results for Pattern 2

Comparing the multi-job evaluation results with those of the single-job evaluation for Pattern 2 reveals a clear distinction in throughput. In both evaluation modes, the CEP query consistently underperforms compared to the SP query. Notably, in the single-job evaluation, the average latencies for both queries are nearly equivalent at P3, P6, and P9. However, under the multi-job evaluation, the SP query demonstrates a significant reduction in latency. This reduction is most likely due to the watermarks of different jobs progressing independently without having to wait for all sources to reach the same timestamps. In contrast, the CEP query maintains latency values comparable to those observed in the single-job setting.

A broader comparison between the multi-job evaluations of Pattern 1 and Pattern 2 reveals consistent trends across both patterns. Specifically, SP queries exhibit higher throughput and lower average latency, while CEP queries show lower throughput and higher latency. This pattern also extends to CPU utilization: in both Pattern 1 and Pattern 2, SP exceeds CEP in CPU utilization at P9, whereas CEP maintains higher utilization at P3 and P6.

Comparing the latency distributions between the multi-job and single-job evaluations, a general reduction in overall latency is observed under the multi-job setup. This trend holds across both Pattern 1 and Pattern 2, where average latency consistently decreases in the multi-job evaluation. Figure 5.22 presents the combined latency distribution for both one-phase and three-phase meters. For a detailed breakdown of latency distributions by meter type, readers are referred to Section A.4.

As observed in the single-job evaluation latency trend plots, the multi-job evaluation also exhibits distinct horizontal lines in the mid-region, indicating periods of uniform latency. The division between several parallel jobs appears to only increase the distortion in the effect without otherwise altering it. As the lines remain present and in the same section of alerts, but less pronounced, it seems to support the previously discussed causes from Section 5.2.1.

5.3 Pattern 3: Over-current Fault

Pattern 3 utilizes all datasets listed in Table 4.1. As mentioned in Section 4.2, synthetic data specifically designed to match Pattern 3 was generated and integrated with the original dataset. Unlike previous patterns, Pattern 3 does not demonstrate a distinction between SP and CEP queries. Instead, it leverages the strengths of both SP and CEP approaches, as demonstrated in earlier patterns. The parallelism of the CEP operator is configured to match the source parallelism. The query took between 37-144 minutes to execute over the data set and generated just over six thousand alerts.

5.3.1 Single-Job Results for Pattern 3

This section presents the results of Pattern 3 under single-job execution.

Table 5.5: Pattern 3: Single-job summary of source and filter throughput, per-thread throughput, and average latency across parallelism levels.

Source Parallelism	1	3	6	9
Source Throughput per Thread (events/s)	390,019	178,228	145,868	100,874
Total Source Throughput (events/s)	390,019	534,685	875,211	907,869
Filter Throughput per Thread (events/s)	112,417	51,186	41,930	29,005
Total Filter Throughput (events/s)	112,417	153,558	251,581	261,043
Average Latency (ms)	596	868	1,122	1,375

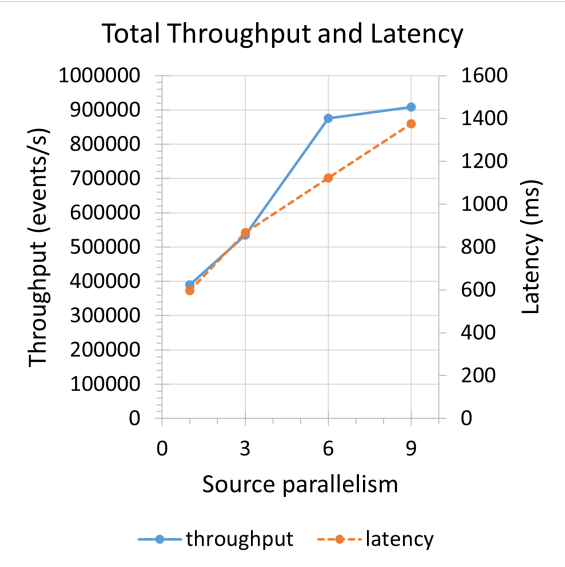


Figure 5.25: Pattern 3: Single-job throughput and average latency.

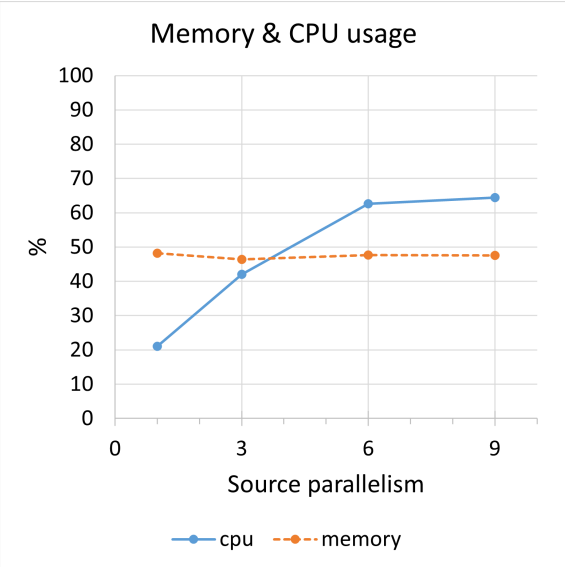


Figure 5.26: Pattern 3: Single-job memory and CPU utilization.

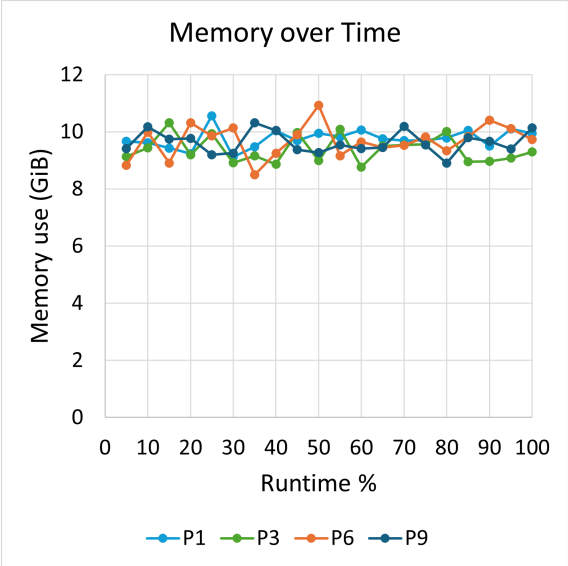


Figure 5.27: Pattern 3: Single-job memory usage over time for varying levels of source parallelism. Each point on the x-axis represents the average memory usage over a consecutive 5% segment of the job's total runtime.

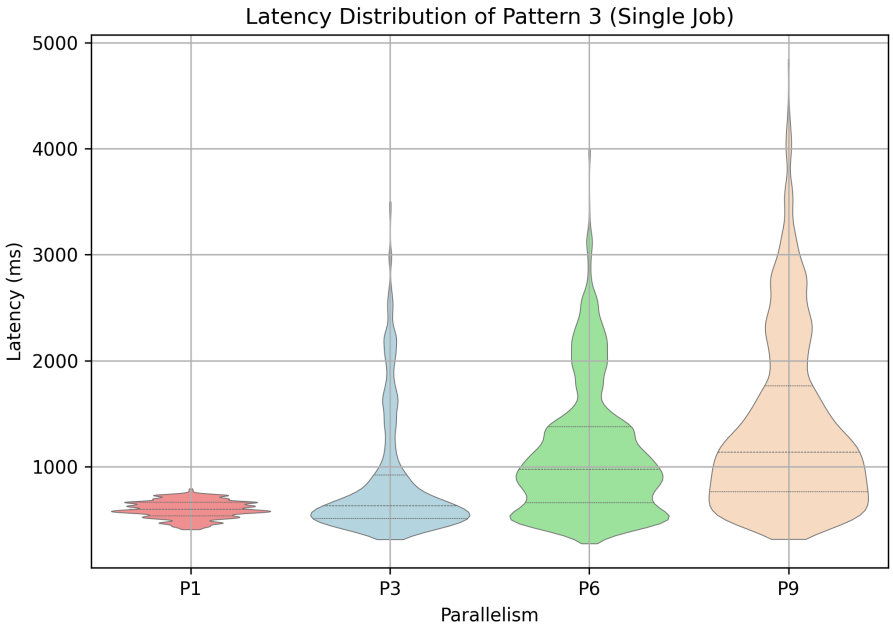


Figure 5.28: Pattern 3: Single-job latency distribution across parallelism levels.

Figure 5.29: Pattern 3: Single-job latency as a function of run number and degree of parallelism. The plot displays a representative sample of 400 data points.

An overview of the results is provided in Table 5.5, with supporting visualizations in Figure 5.25 and Figure 5.26, which illustrate the average throughput, latency, memory usage, and CPU utilization across different levels of source parallelism.

As shown in Figure 5.25, throughput increases linearly with higher parallelism levels. However, between P6 and P9, the increase becomes marginal, and throughput effectively stabilizes at P6. In contrast, average latency also increases linearly with rising source parallelism, suggesting a trade-off between throughput and latency at higher parallelism levels.

Figure 5.26 indicates that CPU utilization trends similarly to throughput, increasing with parallelism but with only a slight change between P6 and P9. Memory usage remains largely consistent across all levels of source parallelism. This stability is further supported by Figure 5.27, which shows that memory consumption over time remains steady regardless of parallelism level.

Finally, Figure 5.28 displays the latency distribution across parallelism settings. At P9, the maximum observed latency reaches 5 seconds. However, the majority of latency values remain within 2 seconds, indicating relatively consistent performance despite the higher peak.

5.3.1.1 Discussion of Single-Job Results for Pattern 3

As observed in the single-job evaluation results presented in Section 5.3.1, throughput increases with higher levels of parallelism, although it stabilizes at P6 and P9. Interestingly, this significant jump from P3 to P6 mirrors the trend seen in the previously discussed Pattern 1 and Pattern 2. The average latency behaves similar to that of SP in Pattern 1 where average latency increases linearly with higher parallelism. CPU utilization and memory usage follow a similar pattern to those in earlier evaluations: CPU utilization increases with parallelism, while memory usage remains relatively stable across all parallelism levels.

The latency distribution reveals particularly interesting behaviour. Unlike Pattern 2, where short horizontal lines appeared only in the middle region of the plot, Pattern 3 displays such lines throughout the entire alert range. This phenomenon might be attributed to the use of SP techniques in Pattern 3, which preprocess the data stream before it is processed by the CEP engine. The SP component likely emits batches of events in sync with the progression of watermarks. As the watermark advances in a consistent manner, the CEP engine generates corresponding results in batches, resulting in clusters of alerts with nearly identical latency values, visualized as horizontal line strips in the plot.

5.3.2 Multi-Job Results for Pattern 3

This section focuses on the results of Pattern 3 under multi-job execution.

This section presents the results of Pattern 3 under multi-job evaluation. Table 5.6 summarizes the key metrics, including throughput, average latency, memory usage,

Table 5.6: Pattern 3: Multi-job summary of source and filter throughput, per-thread throughput, and average latency across parallelism levels.

Source Parallelism	3	6	9
Source Throughput per Thread (events/s)	348,481	238,810	160,223
Total Source Throughput (events/s)	1,045,444	1,432,860	1,442,003
Filter Throughput per Thread (events/s)	100,882	70,189	47,099
Total Filter Throughput (events/s)	302,646	421,136	423,888
Average Latency (ms)	376	428	416

and CPU utilization. These results are further visualized in Figure 5.30 and Figure 5.31.

As depicted in Figure 5.30, throughput increases with higher levels of parallelism, reaching a plateau at P6 and remaining constant through P9. Average latency follows a similar trend, increasing up to P6 and then slightly decreasing at P9.

Figure 5.31 illustrates that CPU utilization also rises with increasing parallelism, with a significant jump observed from P3 to P6 and a marginal increase from P6 to P9. Memory usage, on the other hand, remains relatively stable across all parallelism levels. Interestingly, at P9, memory usage drops below that observed at P3. This trend is corroborated by Figure 5.32, which shows stable memory consumption over time, with the lowest usage occurring at P9.

Finally, Figure 5.33 presents the latency distribution across different parallelism levels using a violin plot. While P9 exhibits a few outliers that reach up to 1 second, the majority of latency values remain within 600 milliseconds, indicating generally consistent performance with some isolated spikes at higher parallelism levels.

5.3.2.1 Discussion of Multi-Job Results for Pattern 3

The most significant deviation observed in the multi-job evaluation compared to the single-job evaluation lies in the average latency. In the multi-job evaluation, the average latency remains relatively stable across different parallelism levels, with a slight increase at P6. Notably, this contrasts with the behaviour seen in previous patterns, where the average latency at P6 during multi-job execution was consistently the lowest. In the case of Pattern 3, however, P6 exhibits the highest average latency relative to P3 and P9 under the same evaluation mode.

The latency distribution of the multi-job exhibits a similar behaviour to that of Pattern 2 in Section 5.2.2. Compared to the single-job evaluation, the multi-job evaluation produces less pronounced and shorter horizontal lines in the latency plot.

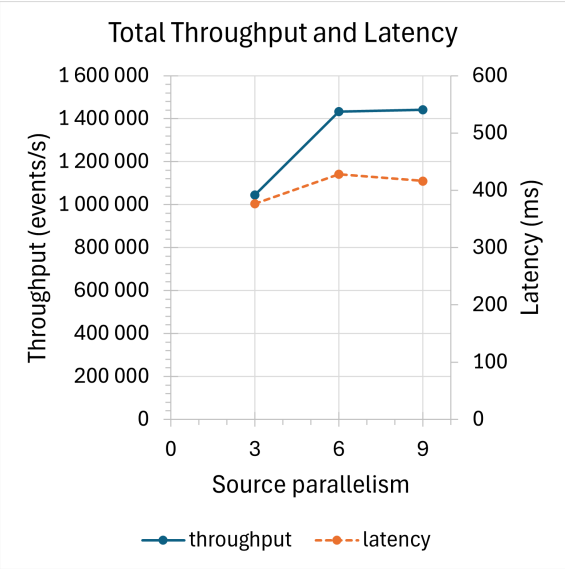


Figure 5.30: Pattern 3: Multi-job throughput and average latency.

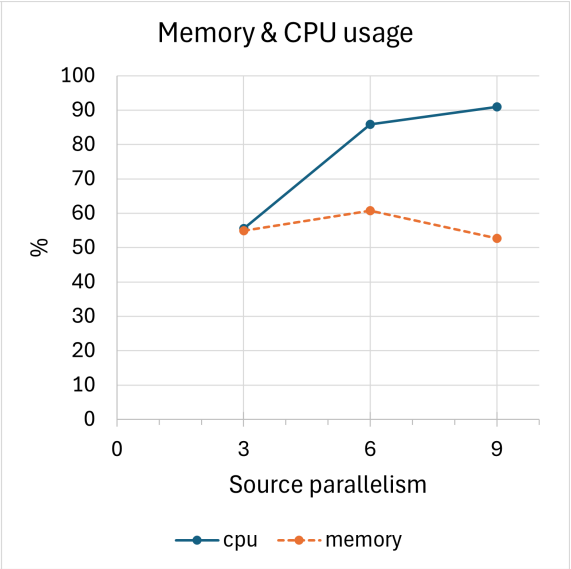


Figure 5.31: Pattern 3: Multi-job memory and CPU utilization.

This suggests that the bursts of identical latency values are less apparent. As multiple jobs run in parallel, the processing is more distributed, and the synchronization of alert generation becomes less tightly grouped, distorting the effect. Additionally, these lines of near constant latency become smaller as parallelism increases, an effect that was also observed in Pattern 2.

Another notable observation is the throughput behaviour, where a plateau is observed at parallelism levels 6 and 9 across all three patterns. A plausible explanation is that, under the current configuration, queries achieve optimal performance at a parallelism level of 6. After a certain point, adding more parallelism creates resource overhead that outweighs the benefits.

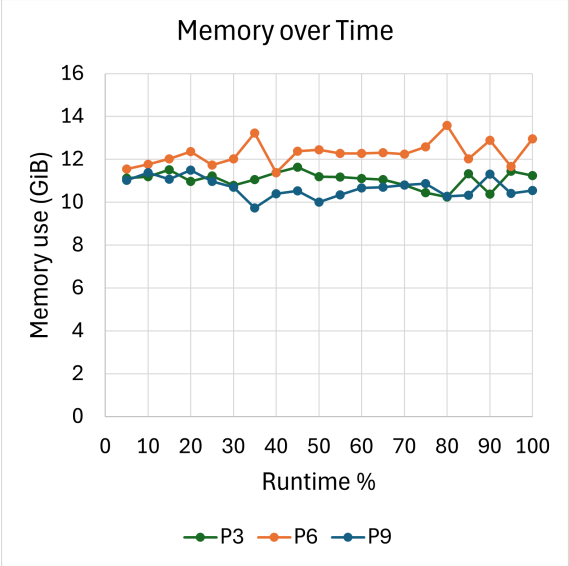


Figure 5.32: Pattern 3: Multi-job memory usage over time for varying levels of source parallelism. Each point on the x-axis represents the average memory usage over a consecutive 5% segment of the job's total runtime.

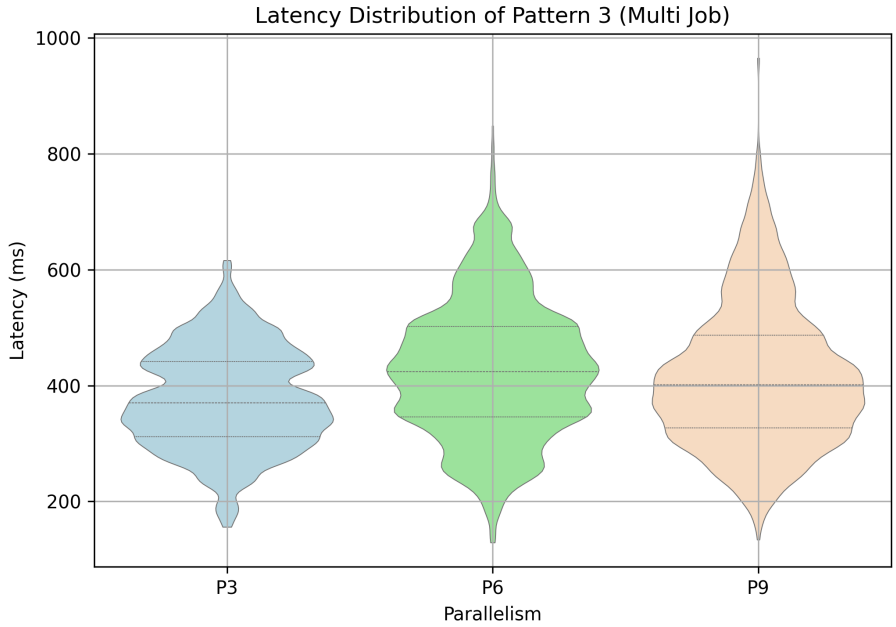


Figure 5.33: Pattern 3: Multi-job latency distribution across parallelism levels.

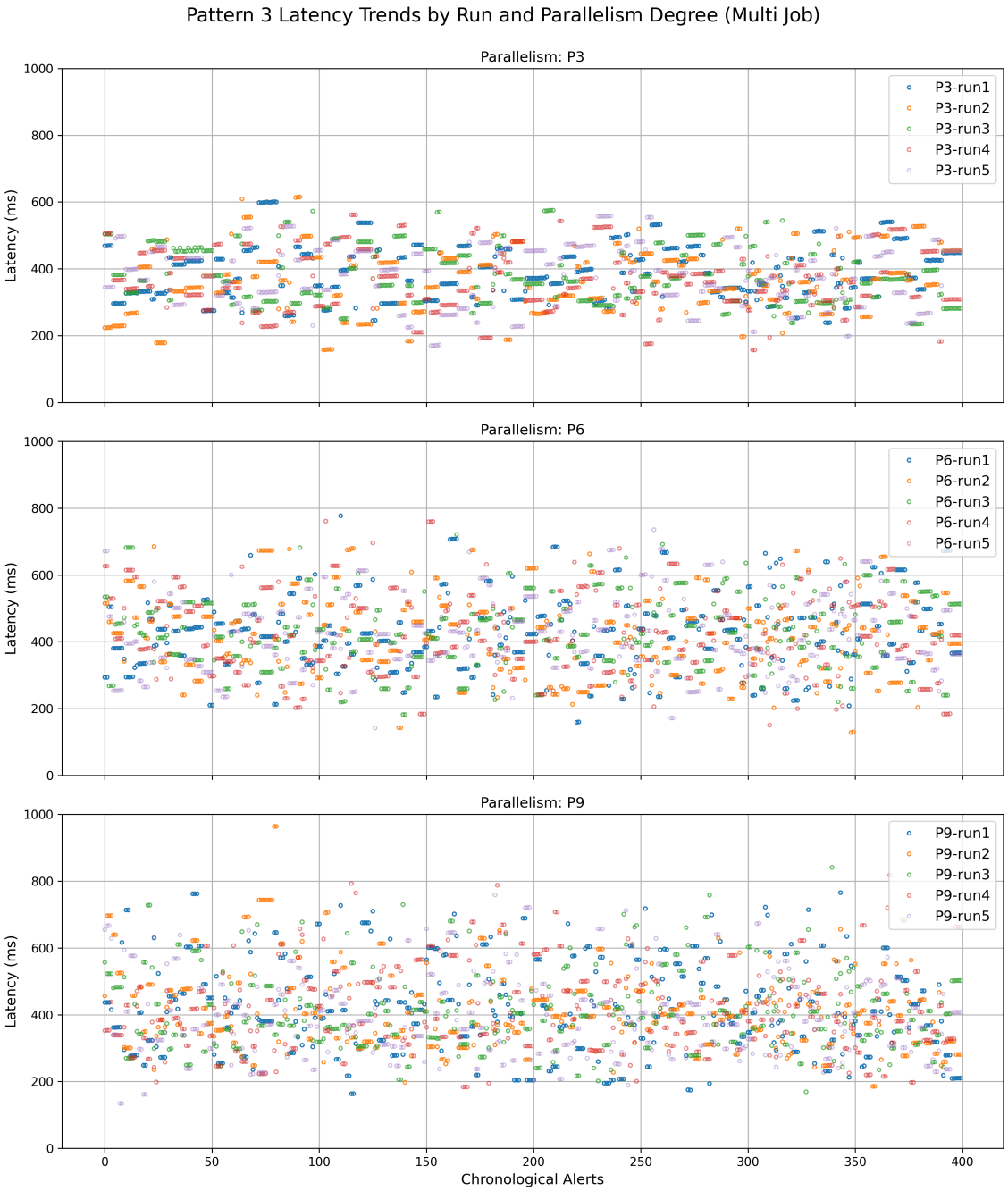


Figure 5.34: Pattern 3: Multi-job latency as a function of run number and degree of parallelism. The plot displays a representative sample of 400 data points.

6

Related work

This section presents related work on the use of SP and CEP across domains. There is growing interest in the advantages of data processing within digitalized systems, particularly in the context of promoting sustainable urban development, where electricity networks play a vital role in infrastructure.

A study comparable to ours, which evaluates the performance of SP and CEP queries, is that of Ziehn *et al.* [30]. They aim to bridge the gap between CEP and SP by decomposing the CEP pattern and mapping it to SP operations. Through this, the authors translate CEP patterns to execute their equivalence as an SP query with the aim of making use of the performance benefits SP queries offer. Like our work, their implementations are done in Flink and FlinkCEP for SP and CEP, respectively. The authors focus on evaluating the performance of translated operators and how the performance changes based on criteria such as pattern length and output selectivity (#matches/#events). In the pattern construction, their choice of CEP operators does not appear to be aimed at detecting the pattern as efficiently as possible. Instead, they limit the construction to a specific set of operators, which they use to create patterns of a certain length and contiguity. In contrast, our work implements patterns that are of interest to the service provider gathering the data. Both our SP and CEP implementations are manually constructed and aim to achieve the same sought-after behaviour with high performance, without necessarily being direct translations of each other. Their performance metrics are similar to ours, where they measure the maximum sustainable throughput, the same type of latency as us, and the CPU & memory usage in different experiments. They conclude that their SP mapping on average achieves 60% higher throughput, which is a similar difference to what we observe in some of our tests. They also further point to FlinkCEP exhibiting large memory consumption and greatly decreased throughput with increased workloads. In contrast, our experiments show that the average memory consumption for CEP is generally equivalent to, or lower than, that of the SP implementations. Their experiments also appear to be highly memory-intensive, with queries using between 100-200 GB of memory at a given time but only 20-70% of the CPU. Our queries, on the other hand, use 6-10 GB of memory on average and between 15-95% of the CPU, with the CPU usage increasing with higher parallelism settings. These diverging results might stem from differences in implementation, data properties and data handling.

Many studies investigate the performance of either SP or CEP under different set-

tings. Similar to our thesis, they often employ comparable metrics and data sources or address the same problem from a different perspective. Some of the following research also explores the applicability of either SP or CEP in SGs, demonstrating its use and value in this domain. These also highlight how similar metrics are commonly employed across related studies.

Grabs *et al.* [31] present a study on measuring the latency performance of CEP systems, distinguishing between "system latency"and "information latency". System latency refers to delays introduced by the processing of events, that is, the time it takes for a CEP system to complete its computations. In contrast, information latency captures the delays incurred while the system waits for additional input data. For systems that permit out-of-order event arrival, information latency is identified as a critical factor influencing overall system performance. The paper primarily focuses on measuring and comparing information latency across different CEP systems. The authors propose a framework for assessing CEP system performance with respect to both data rate and latency. While these insights are relevant to the context of this thesis, our evaluation of latency focuses specifically on system latency and compares the performance between SP and CEP queries for the selected patterns rather than between different CEP systems.

Gulisano *et al.* [32] introduce StreamCloud, a scalable SPE designed to handle growing data volumes by aggregating the processing capacity of multiple nodes. Scalability is achieved through query parallelization and the introduction of specialized operators that encapsulate the logic for parallel execution. A key contribution is a novel parallelization strategy that reduces distribution overhead, thereby enhancing performance. Additionally, StreamCloud incorporates mechanisms for elasticity and dynamic load balancing, which enable the system to adapt to fluctuating workloads while minimizing resource usage. While their focus is on scalability strategies, our work complements this by conducting two types of evaluations to better understand infrastructure choices, using real data and incorporating metrics that support resource selection decisions.

A group of works makes use of data provenance, which is the tracing of events through an application. Data provenance is said to be valuable for debugging and testing as well as tracing and investigating the causes behind system behaviour. A type of provenance called forward provenance finds the possible outcomes that can occur stemming from some data in the data stream, which in a live setting can enable identification of potential critical outcomes ahead of time. A second type, called backward provenance, traces backwards to identify the events that caused a certain outcome after it has occurred. These functionalities could be valuable tools for testing and debugging patterns and could complement an implementation using pattern templates. If the criticality of the pattern could be set through the templates, it might also be applicable together with forward provenance. This would then allow for identification of potentially critical outcomes ahead of time in a live setting, enabling faster responses.

In the research summary of Palyvos-Giannas *et al.* [33], the authors discuss SP and edge computing as a way to handle the increasing data volumes as well as the oppor-

tunities and challenges that come with combining the two approaches. Their work focuses on data provenance and optimization in the form of scheduling. The scheduling part of the research summary addresses how the execution of concurrent applications on limited resources can be optimized while adhering to different performance requirements. The scheduling can be done with either custom user- or OS-level scheduling, and both approaches of scheduling are said to achieve higher performance than that of default OS-level scheduling, but the custom OS-level scheduling may suffer compatibility issues. Our work similarly explores SP as a way to handle the increasing data volumes, but then by comparing its performance against that of CEP for pattern recognition.

Palyvos-Giannas *et al.* [13] present a framework, referred to as Ananke, for forward provenance in a live setting. The work compares performance in the form of throughput, latency, memory, and CPU utilization between different Ananke implementations, as well as with on-demand implementations for provenance on data from different sources. One of these datasets originates from the SG domain, a source also used in our work. However, similar to their performance comparisons, we evaluate differences between SP and CEP techniques, leveraging real-time data processing and metrics that guide resource selection decisions.

In their paper, Palyvos-Giannas *et al.* [34] present a so-called why-provenance framework, which aims to explain why the results of queries on a data stream are missing parts of the output expected by the user. This is done by identifying events within a set time interval that could have led to the expected output but were removed by some operator in the SP pipeline. Part of the query evaluation of their framework is on real-world data from the SG, similar to the data used for pattern performance evaluation between SP and CEP in our thesis.

In the work of Palyvos-Giannas *et al.* [14] GeneaLog, a framework and technique for data provenance in deterministic streaming applications using fixed-size metadata attached to each event, is proposed. The attached metadata is used to link an operators output events with its input events, and through this, an output event can be traced back to the source input, forming a contribution graph. In large part, this is done by including the identification of events that, in the most recent operation, created the currently addressed event. This allows the framework to trace multiple input tuples as well as longer input chains to their source, while still keeping the size of the metadata attached to any one event constant. Among other sources, part of the evaluation of the framework was conducted on data from the SG. This work provided additional context for designing our latency-capturing mechanism, particularly through incorporating timestamps within events to measure latency effectively.

Another work concerning SP in the SG is that of Van Rooij *et al.* [12]. They propose a middleware, referred to as TinTiN, to allow SP operations not affected by time or ordering to progress while allowing out-of-order handling to operations that do by using a D-bound eventual determinism. This includes the ability to reevaluate results affected by late-arriving events by replaying a portion of already processed input together with the newly arrived. As a result, it does not guarantee only

once delivery of results that are reevaluated, but instead provides timely processing of both complete and time-insensitive data. In our work we have refrained from handling out-of-order data, where this work could greatly complement ours. In an environment where the expected number of events is known, as for the SM readings, this middleware could be especially beneficial as both the memory burden of handling out-of-order events and the latency of complete data can be reduced.

Van Rooij *et al.* [25] explore the application of SP techniques to SG devices and presents *LoCoVolt*, a system designed to detect SMs that report faulty power quality readings. Developed using Apache Flink, LoCoVolt prioritizes performance under hardware constraints. The system operates by comparing voltage measurements from geographically proximate SMs, based on the premise that nearby meters should exhibit similar voltage trends. LoCoVolt functions in a distributed manner: each SM can "accuse"another if its readings deviate significantly from those of its neighbours. The weight of each accusation is influenced by the number of allegations previously received by the accusing meter, introducing a form of weighted consensus. The system's effectiveness is evaluated by analysing the number of accusations required to detect malfunctioning meters reliably. Similar to our approach, Van Rooij *et al.* [25] show that SP can be effectively applied to detect anomalies in SM data. In our work, we extend this idea by employing CEP queries to identify anomalous patterns.

Van Rooij *et al.* [35] also highlight the growing importance of data analysis in SGs, particularly in light of increasing regulatory pressures driving their adoption. The authors propose a hybrid system called eChIDNA, which leverages both the streaming paradigm and the traditional store-then-process approach. Data validation rules are implemented using Apache Storm, and their evaluation shows that this hybrid model achieves high throughput with low latency. Their findings suggest that such validation rules could serve as a foundation for developing real-time error correction mechanisms within streaming frameworks. Notably, the system was able to validate 270,000 hourly readings using only 5% of available resource capacity, demonstrating that SP applications can scale efficiently and are well-suited for SG environments. While their work demonstrates the scalability of streaming-based validation, our approach builds on this by leveraging CEP-based detection for anomaly identification in SM data, using real-world data streams and incorporating performance metrics that support resource selection decisions.

Gulisano *et al.* [36] emphasize the critical role of data structures in streaming applications. Given that such applications operate on unbounded data streams in real time, efficient data structures are essential for maintaining state and enabling high-performance processing. Specifically, the authors examine how data structures function as articulation points, key connectors that maintain operator state, facilitate high parallelism. The study focuses on data structures used during streaming aggregation, particularly in scenarios involving multiple input streams. The authors propose two novel objects, T-Gate and W-Gate, along with their algorithmic implementations. These structures serve as the foundation for three enhanced multiway aggregate operators. Evaluations using datasets from SoundCloud and SG networks demonstrate that their approach outperforms existing implementations in terms of both throughput and latency, for both order-sensitive and order-insensitive

aggregation functions. Both their work and ours show the applicability of SP in an SG context, but their work addresses fundamental operator-level optimizations, our study takes a broader view by comparing SP and CEP systems using similar performance metrics, including throughput and latency.

Gulisano *et al.* [37] investigate the integration of SP with differential privacy, addressing the critical challenge of privacy in cyber-physical systems such as SGs and AMIs. Differential privacy offers strong guarantees against powerful adversaries, making it a suitable approach for protecting sensitive data in these contexts. The authors introduce a streaming-based framework called BES, designed for AMIs. The framework proposes techniques to minimize the noise typically introduced by differential privacy, thereby improving data utility while preserving strong privacy guarantees. Performance is evaluated by varying the event injection rate, using real-world AMI data. The results demonstrate that BES can process thousands of events per second with low latency, while maintaining error rates below 10%, showcasing its practicality for privacy-preserving SP in SGs.

Gulisano *et al.* [38] address the challenge of data validation in stream data, a critical requirement in the transition from traditional power grids to SGs with AMIs. The advent of AMIs enables two-way communication, supporting real-time applications such as intrusion detection and demand-response systems. However, these applications rely on accurate and timely data, necessitating validation to filter out noise before it reaches utility providers. The authors propose a data validation approach based on the SP paradigm, utilizing the Apache Storm SPE. The evaluation, conducted using real-world AMI data, measures performance in terms of throughput and latency across three different sets of validation rules and four batch sizes. Results show that while both throughput and latency increase with batch size, latency remains within a few milliseconds, demonstrating the systems ability to scale while preserving real-time responsiveness efficiently. Building on the approaches by Gulisano *et al.* [37], [38], our work similarly applies SP to SG data, employing comparable performance metrics such as throughput and latency to evaluate scalability and responsiveness.

Gulisano *et al.* [4] present the application of SP in an intrusion detection system for SGs. The authors introduce a streaming-based, two-tier intrusion detection framework called METIS, which leverages probabilistic methods to detect cyberattacks. The adversary model targets the theft of energy consumption information from AMI users, assuming that malicious activity occurs either by compromising a Meter Concentrator Unit (MCU) or by deploying a rogue MCU. The first tier of METIS, referred to as the Interaction Modeler, processes communication between AMI devices and uses anomaly-based detection to identify suspicious messages. Given the possibility of message loss in the network, legitimate messages may be misclassified as suspicious. To address this, the second tier applies expert-defined pattern-matching rules to verify suspicious messages identified by the first tier. The METIS framework is implemented using Apache Storm. The results demonstrate that METIS can achieve high detection accuracy with a low false alarm rate, making it an effective solution for real-time intrusion detection in AMI environments. While their work demonstrates the scalability of streaming-based validation, our approach builds on

this by leveraging CEP-based detection for anomaly identification in SM data, using real data streams and incorporating performance metrics that support resource selection decisions. Moreover, their work closely resembles the Pattern 3 query, as it also employs a two-stage structure: the first stage processes data using SP, followed by a CEP-based pattern check.

Itria *et al.* [1] explores anomaly detection in Information and Communication Technology networks, with the main focus on finding known cyber-attacks, by applying CEP patterns on Simple Network Management Protocol data. Their proposed solution is applied and tested on a simulated SG environment of a single node, and they reproduced cyberattacks both to construct and test their CEP patterns. Similar to their approach, our work leverages CEP patterns for anomaly detection in the SG. However, we apply our methods to real-world data, while their evaluation relies on artificial datasets.

Few studies have investigated various approaches to simplifying the creation and management of CEP patterns, aiming to make them more accessible in different application domains. While they share the same goal of making pattern creation easier, their approach to this differs from our approach to work towards pattern templates. These efforts range from graphical interfaces for intuitive rule definition to logical operators that translate high-level expressions into executable CEP queries.

Liang *et al.* [39] propose a graphical interface for dynamic rule definition. Their work seeks to simplify rule creation and management of CEP systems by providing a graphical user interface tailored for threat detection and defence in combat systems. The interface enables users to insert, update, and remove rules without programming expertise, utilizing a structured format IF <conditions> FROM <target object> WINDOW <time frame> (optional) THEN <output>which supports chaining to construct complex CEP patterns. The evaluation of their work focuses on system-level performance based on rule configurations using a dataset of 10,000 simulated events. This is a work that could complement that of template creation. By combining the two approaches, one could potentially simplify template use, allowing template chaining and enhancing management of deployed patterns.

Giatrakos *et al.* [40] present a logical CEP operator that transforms extended regular expressions and rewrites them into the FlinkCEP program, and an accompanying optimizer. In their work, they compare the performance of the resulting FlinkCEP queries, as well as their optimizer, using a dataset with 16M simple events divided evenly over 8 streams. Their performance measure focused on the resulting throughput when varying parallelism and skip policies for strict, relaxed, and non-deterministic queries. Our work similarly varies the source parallelism in our performance comparison of SP and CEP queries, with the throughput being one of the performance measurements.

A further range of research efforts has explored the integration of machine learning and rule-based methods to facilitate the generation of CEP patterns. Through their work, the authors explore a different approach to the same problem of simplifying pattern creation. These works present alternatives to our efforts in template creation and may work better with the availability of historical data.

Kumar *et al.* [41] propose a framework making use of machine learning classifiers to extract CEP rules. This is proposed as a complement to the input from domain experts to efficiently create error-free CEP rules. Their analysis, performed on an air quality dataset, shows that rule extraction using Decision Trees yields high accuracy and outperforms traditional rule extraction approaches for data streams.

Naseri *et al.* [42] propose a health monitoring system that makes use of CEP. They employ rule-based machine learning methods for CEP rule extraction from a hospital dataset. This is with the aim of decreasing the need for human involvement and increasing the explainability of the system. They conclude the JRip algorithm to be the most appropriate method among the rule-based methods investigated in their work. This conclusion is based on JRip's accuracy, together with a lower number of resulting CEP rules from applying the method compared to other methods with high accuracy.

Roldán-Gómez *et al.* [43] present a machine learning-driven method using Principal Component Analysis for generating CEP patterns to detect cyberattacks in the Internet of Things (IoT). Their approach utilizes historical data to derive rules for both previously known attacks and unspecified anomalous behaviour. The proposed algorithm caters to the deployment of CEP rules on edge devices with lower performance.

Anicic *et al.* [44] present a logic-based approach to CEP, offering formal semantics and reasoning capabilities over events and their interrelationships. The proposed language is grounded in deductive rules and provides a clear, declarative, and formally defined semantics for expressing complex event patterns. In contrast to traditional logic-based systems, which often compile to Prolog and suffer from runtime inefficiencies, this approach introduces a novel execution model. It compiles complex event patterns directly into logic rules, enabling timely and efficient detection of complex events. Moreover, unlike existing logic-based rule systems, it supports an alternative execution model specifically designed for event-driven scenarios, facilitating real-time, rule-based detection.

Simsek *et al.* [45] introduce a novel, generalized framework for automatic CEP rule extraction using deep learning techniques. The proposed framework, named ARECEP (Automatic Rule Extraction for CEP), is designed to extract meaningful and accurate CEP rules from unlabelled IoT time-series data. ARECEP operates in two main phases: data labelling and automatic rule extraction. In the labelling phase, various DL-based methods are applied to the raw time-series data to identify anomalous patterns. The most accurate labelling results, those with the lowest reconstruction error, are then passed to the rule mining phase, where multiple rule extraction methods are applied comparatively to derive CEP rules. The authors evaluate the performance of ARECEP using an air quality dataset collected from a smart city application. The results show that ARECEP can effectively generate accurate and meaningful CEP rules from unlabelled data, demonstrating its potential for anomaly detection and real-time event recognition in IoT environments. This approach could be extended to our SG use case by applying it to the available dataset. In particular, historic data from the main stage could be leveraged to automatically

generate CEP pattern rules. Furthermore, the second stage of our approach could be generalized by integrating this automatic rule extraction process into template rule creation.

Margara *et al.* [46] address the challenge of manual rule specification in CEP by introducing iCEP, a framework that automatically generates CEP rules from historical event traces. The authors formally define the problem of automated rule generation and propose a modular, decomposable approach supported by custom learning algorithms. Their implementation of iCEP is evaluated using both synthetic data and real-world traffic monitoring scenarios, demonstrating its effectiveness in uncovering hidden causal relationships between events and improving the automation of rule creation in CEP systems.

Kumar *et al.* [41], Naseri *et al.* [42], Roldán-Gómez *et al.* [43], Anicic *et al.* [44], Simsek *et al.* [45], and Margara *et al.* [46] explore the use of machine learning techniques for automatic extraction of CEP rules from observed data, aiming to reduce human effort and improve rule accuracy in anomaly detection and event recognition. While these approaches leverage classifiers, deep learning, or logic-based inference to generate rules, our work differs in that we focus on the manual construction of CEP queries. Nonetheless, we share the same fundamental strategy of analyzing observed data streams to detect anomalies by constructing CEP patterns, emphasizing interpretability over the detection logic.

7

Conclusion

This thesis aimed to compare the performance of SP and CEP in the context of pattern recognition in data streams with SM data by comparing patterns implemented with each technique. These findings were then used to combine the two techniques and guide the creation of pattern templates. In our findings, we confirmed the performance advantage of SP but also showed that the combination of the technique can yield comparable performance results when applied to SG data, while opening up for reusable components that can be combined in future patterns. The trade-offs between different queries and their implementations should be evaluated concerning the chosen performance metric and addressed accordingly.

7.1 Future Work

As a continuation of our findings, several paths could be explored. A promising next step is to design templates as small, reusable building blocks that can be combined to construct complete patterns. For example, in Pattern 3, one block could perform stream processing to identify relevant events, followed by another block that applies a CEP pattern check based on the output of the first. This modular approach would simplify the construction of complex patterns by assembling simpler components. An extension of this idea is the development of a framework where users provide example events as input, and the system automatically generates a pattern query. Such a framework would allow users to benefit from complex event detection without needing to manage the underlying programming details.

Another direction could be examining how both system latency and information latency combined may impact the performance of various pattern implementations. It could, for example, be valuable to explore whether certain implementation approaches perform better when information latency is relatively low or high. Exploring these relationships could then guide more effective design choices for different use cases.

A third potential direction could be to explore which approach works best for handling multiple patterns running simultaneously on the same data. Since the patterns in our work have been executed one at a time, it does not cover how well they manage shared resources and intermediate results. Further exploration in this area could provide insights into how a composite system with multiple patterns is best set up.

Bibliography

[1] M. L. Itria, E. Schiavone, and N. Nostro, "Towards anomaly detection in smart grids by combining complex events processing and snmp objects," in *2021 IEEE International Conference on Cyber Security and Resilience (CSR)*, 2021, pp. 212–217. DOI: 10.1109/CSR51186.2021.9527928.

[2] H. Taherdoost, "A systematic review of big data innovations in smart grids," *Results in Engineering*, vol. 22, p. 102 132, 2024, ISSN: 2590-1230. DOI: https://doi.org/10.1016/j.rineng.2024.102132. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2590123024003864.

[3] M. J. B. Kabeyi and O. A. Olanrewaju, "Smart grid technologies and application in the sustainable energy transition: A review," *International Journal of Sustainable Energy*, vol. 42, no. 1, pp. 685–758, 2023. DOI: 10.1080/14786451.2023.2222298. eprint: https://doi.org/10.1080/14786451.2023.2222298. [Online]. Available: https://doi.org/10.1080/14786451.2023.2222298.

[4] V. Gulisano, M. Almgren, and M. Papatriantafilou, "Metis: A two-tier intrusion detection system for advanced metering infrastructures," in *Proceedings of the 5th International Conference on Future Energy Systems*, ser. e-Energy '14, Cambridge, United Kingdom: Association for Computing Machinery, 2014, pp. 211–212, ISBN: 9781450328197. DOI: 10.1145/2602044.2602072. [Online]. Available: https://doi.org/10.1145/2602044.2602072.

[5] L. Wen, K. Zhou, S. Yang, and L. Li, "Compression of smart meter big data: A survey," *Renewable and Sustainable Energy Reviews*, vol. 91, pp. 59–69, 2018, ISSN: 1364-0321. DOI: https://doi.org/10.1016/j.rser.2018.03.088. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1364032118301849.

[6] "Smart grids and meters." (), [Online]. Available: https://energy.ec.europa.eu/topics/markets-and-consumers/smart-grids-and-meters_en.

[7] D. B. Avancini, J. J. Rodrigues, S. G. Martins, R. A. Rabêlo, J. Al-Muhtadi, and P. Solic, "Energy meters evolution in smart grids: A review," *Journal of Cleaner Production*, vol. 217, pp. 702–715, 2019, ISSN: 0959-6526. DOI: https://doi.org/10.1016/j.jclepro.2019.01.229. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0959652619302501.

[8] G. R. Barai, S. Krishnan, and B. Venkatesh, "Smart metering and functionalities of smart meters in smart grid - a review," in *2015 IEEE Electrical Power and Energy Conference (EPEC)*, 2015, pp. 138–145. DOI: 10.1109/EPEC.2015.7379940.

[9] D. C. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. USA: Addison-Wesley Longman Publishing Co., Inc., 2001, ISBN: 0201727897.

[10] D. Robins, "Complex event processing," in *Second International Workshop on Education Technology and Computer Science. Wuhan*, Citeseer, 2010, pp. 1–10.

[11] B. M. Michelson, "Event-driven architecture overview," *Patricia Seybold Group*, vol. 2, no. 12, pp. 10–1571, 2006.

[12] J. Van Rooij, V. Gulisano, and M. Papatriantafilou, "Tintin: Travelling in time (if necessary) to deal with out-of-order data in streaming aggregation," in *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '20, Montreal, Quebec, Canada: Association for Computing Machinery, 2020, pp. 141–152, ISBN: 9781450380287. DOI: 10.1145/3401025.3401769. [Online]. Available: https://doi.org/10.1145/3401025.3401769.

[13] D. Palyvos-Giannas, B. Havers, M. Papatriantafilou, and V. Gulisano, "Ananke: A streaming framework for live forward provenance," *Proc. VLDB Endow.*, vol. 14, no. 3, pp. 391–403, Nov. 2020, ISSN: 2150-8097. DOI: 10.14778/3430915.3430928. [Online]. Available: https://doi.org/10.14778/3430915.3430928.

[14] D. Palyvos-Giannas, V. Gulisano, and M. Papatriantafilou, "Genealog: Fine-grained data streaming provenance in cyber-physical systems," *Parallel Computing*, vol. 89, p. 102 552, 2019, ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2019.102552. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819119301437.

[15] M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos, "A survey on the evolution of stream processing systems," *The VLDB Journal*, vol. 33, no. 2, pp. 507–541, 2024, ISSN: 0949-877X. DOI: 10.1007/s00778-023-00819-8. [Online]. Available: https://doi.org/10.1007/s00778-023-00819-8.

[16] K. Hwang, X. Bai, Y. Shi, M. Li, W.-G. Chen, and Y. Wu, "Cloud performance modeling with benchmark evaluation of elastic scaling strategies," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 1, pp. 130–143, 2016. DOI: 10.1109/TPDS.2015.2398438.

[17] Wikipedia, *Wikipedia voltage*, Accessed: 2025-05-07, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Voltage.

[18] IEC, *International electrotechnical commission*, Accessed: 2025-05-07, 2025. [Online]. Available: https://www.iec.ch/world-plugs.

[19] S. electric, *Power*, Accessed: 2025-05-07, 2025. [Online]. Available: https://eshop.se.com/in/blog/post/difference-between-active-power-reactive-power-and-apparent-power.html#:~:text=Active%20power%2C%20also%20known%20as,being%20utilized%20in%20a%20system..

[20] R. Rashed Mohassel, A. Fung, F. Mohammadi, and K. Raahemifar, "A survey on advanced metering infrastructure," *International Journal of Electrical Power Energy Systems*, vol. 63, pp. 473–484, 2014, ISSN: 0142-0615. DOI: https://doi.org/10.1016/j.ijepes.2014.06.025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0142061514003743.

[21] X. Fang, S. Misra, G. Xue, and D. Yang, "Smart grid the new and improved power grid: A survey," *IEEE Communications Surveys Tutorials*, vol. 14, no. 4, pp. 944–980, 2012. DOI: 10.1109/SURV.2011.101911.00087.

[22] A. Flink. "Flink concepts - overview." (2024), [Online]. Available: https://nightlies.apache.org/flink/flink-docs-release-1.20/docs/concepts/overview/.

[23] A. Flink. "Timely stream processing." (), [Online]. Available: https://nightlies.apache.org/flink/flink-docs-release-2.0/docs/concepts/time/.

[24] Apache Flink Team, *Streaming analytics*, https://nightlies.apache.org/flink/flink-docs-release-2.0/docs/learn-flink/streaming_analytics/, Nightly docs, version 2.0; accessed 20250722, 2025.

[25] J. Van Rooij, V. Gulisano, and M. Papatriantafilou, "Locovolt: Distributed detection of broken meters in smart grids through stream processing," in *Proceedings of the 12th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '18, Hamilton, New Zealand: Association for Computing Machinery, 2018, pp. 171–182, ISBN: 9781450357821. DOI: 10.1145/3210284.3210298. [Online]. Available: https://doi.org/10.1145/3210284.3210298.

[26] A. Flink. "Operators." (), [Online]. Available: https://nightlies.apache.org/flink/flink-docs-release-2.0/docs/dev/datastream/operators/overview/.

[27] RisingWave, *Master tumbling and hopping windows for stream analytics success*, https://risingwave.com/blog/master-tumbling-and-hopping-windows-for-stream-analytics-success/, Accessed: 2025-07-22, May 2024.

[28] A. Gupta, *Stream processingăăwindows*, https://hackernoon.com/stream-processing-windows, Accessed: 2025-07-22, Nov. 2024.

[29] A. Flink. "Flinkcep - complex event processing for flink." (2024), [Online]. Available: https://nightlies.apache.org/flink/flink-docs-master/docs/libs/cep/.

[30] A. Ziehn, P. M. Grulich, S. Zeuch, and V. Markl, "Bridging the gap: Complex event processing on stream processing systems.," in *EDBT*, 2024, pp. 447–460.

[31] T. Grabs and M. Lu, "Measuring performance of complex event processing systems," in *Topics in Performance Evaluation, Measurement and Characterization*, R. Nambiar and M. Poess, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 83–96, ISBN: 978-3-642-32627-1.

[32] V. Gulisano, R. Jiménez-Peris, M. Patiño-Martínez, C. Soriente, and P. Valduriez, "Streamcloud: An elastic and scalable data streaming system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 12, pp. 2351–2365, 2012. DOI: 10.1109/TPDS.2012.24.

[33] D. Palyvos-Giannas, M. Papatriantafilou, and V. Gulisano, "Research summary: Deterministic, explainable and efficient stream processing," in *Proceedings of the 2022 Workshop on Advanced Tools, Programming Languages, and PLatforms for Implementing and Evaluating Algorithms for Distributed Systems*, ser. ApPLIED '22, Salerno, Italy: Association for Computing Machinery,

2022, pp. 65–69, ISBN: 9781450392808. DOI: 10.1145/3524053.3542750. [Online]. Available: https://doi.org/10.1145/3524053.3542750.

[34] D. Palyvos-Giannas, K. Tzompanaki, M. Papatriantafilou, and V. Gulisano, "Erebus: Explaining the outputs of data streaming queries," vol. 16, no. 2, pp. 230–242, Oct. 2022, ISSN: 2150-8097. DOI: 10.14778/3565816.3565825. [Online]. Available: https://doi.org/10.14778/3565816.3565825.

[35] J. Van Rooij, J. Swetzén, V. Gulisano, M. Almgren, and M. Papatriantafilou, "Echidna: Continuous data validation in advanced metering infrastructures," in *2018 IEEE International Energy Conference (ENERGYCON)*, IEEE, 2018, pp. 1–6.

[36] V. Gulisano, Y. Nikolakopoulos, D. Cederman, M. Papatriantafilou, and P. Tsigas, "Efficient data streaming multiway aggregation through concurrent algorithmic designs and new abstract data types," *ACM Trans. Parallel Comput.*, vol. 4, no. 2, Oct. 2017, ISSN: 2329-4949. DOI: 10.1145/3131272. [Online]. Available: https://doi.org/10.1145/3131272.

[37] V. Gulisano, V. Tudor, M. Almgren, and M. Papatriantafilou, "Bes: Differentially private and distributed event aggregation in advanced metering infrastructures," in *Proceedings of the 2nd ACM International Workshop on Cyber-Physical System Security*, ser. CPSS '16, Xi'an, China: Association for Computing Machinery, 2016, pp. 59–69, ISBN: 9781450342889. DOI: 10.1145/2899015.2899021. [Online]. Available: https://doi.org/10.1145/2899015.2899021.

[38] V. Gulisano, M. Almgren, and M. Papatriantafilou, "Online and scalable data validation in advanced metering infrastructures," in *IEEE PES Innovative Smart Grid Technologies, Europe*, 2014, pp. 1–6. DOI: 10.1109/ISGTEurope.2014.7028740.

[39] Y. Liang, J. Lee, B. Hong, and W. Kim, "Design and implementation of rule-based cep for threat detection and defense," in *2019 IEEE International Symposium on INnovations in Intelligent SysTems and Applications (INISTA)*, 2019, pp. 1–6. DOI: 10.1109/INISTA.2019.8778299.

[40] N. Giatrakos, E. Kougioumtzi, A. Kontaxakis, A. Deligiannakis, and Y. Kotidis, "Easyflinkcep: Big event data analytics for everyone," in *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, ser. CIKM '21, Virtual Event, Queensland, Australia: Association for Computing Machinery, 2021, pp. 3029–3033, ISBN: 9781450384469. DOI: 10.1145/3459637.3482094. [Online]. Available: https://doi.org/10.1145/3459637.3482094.

[41] S. S. Kumar, A. Kumar, R. Chandra, S. Agarwal, M. Syafrullah, and K. Adiyarta, "Rule extraction using machine learning classifiers for complex event processing," in *2023 10th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*, 2023, pp. 355–360. DOI: 10.1109/EECSI59885.2023.10295840.

[42] M. M. Naseri, S. Tabibian, and E. Homayounvala, "Intelligent rule extraction in complex event processing platform for health monitoring systems," in *2021 11th International Conference on Computer Engineering and Knowledge (ICCKE)*, 2021, pp. 163–168. DOI: 10.1109/ICCKE54056.2021.9721525.

[43] J. Roldán-Gómez, J. Boubeta-Puig, J. M. Castelo Gómez, J. Carrillo-Mondéjar, and J. L. Martínez Martínez, "Attack pattern recognition in the internet of things using complex event processing and machine learning," in *2021 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2021, pp. 1919–1926. DOI: 10.1109/SMC52423.2021.9658711.

[44] D. Anicic, P. Fodor, S. Rudolph, R. Stühmer, N. Stojanovic, and R. Studer, "A rule-based language for complex event processing and reasoning," in *Web Reasoning and Rule Systems*, P. Hitzler and T. Lukasiewicz, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 42–57, ISBN: 978-3-642-15918-3.

[45] M. U. Simsek, F. Y. Okay, and S. Ozdemir, "A deep learning-based cep rule extraction framework for iot data," *The Journal of Supercomputing*, vol. 77, no. 8, pp. 8563–8592, 2021, ISSN: 1573-0484. DOI: 10.1007/s11227-020-03603-5. [Online]. Available: https://doi.org/10.1007/s11227-020-03603-5.

[46] A. Margara, G. Cugola, and G. Tamburrelli, "Learning from the past: Automated rule generation for complex event processing," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '14, Mumbai, India: Association for Computing Machinery, 2014, pp. 47–58, ISBN: 9781450327374. DOI: 10.1145/2611286.2611289. [Online]. Available: https://doi.org/10.1145/2611286.2611289.

A

Appendix 1

A.1 Single-Job Results for Pattern 1

Figure A.1 presents a violin plot illustrating the latency distribution of CEP and SP for Pattern 1 single job, with individual points overlaid to highlight the outliers.

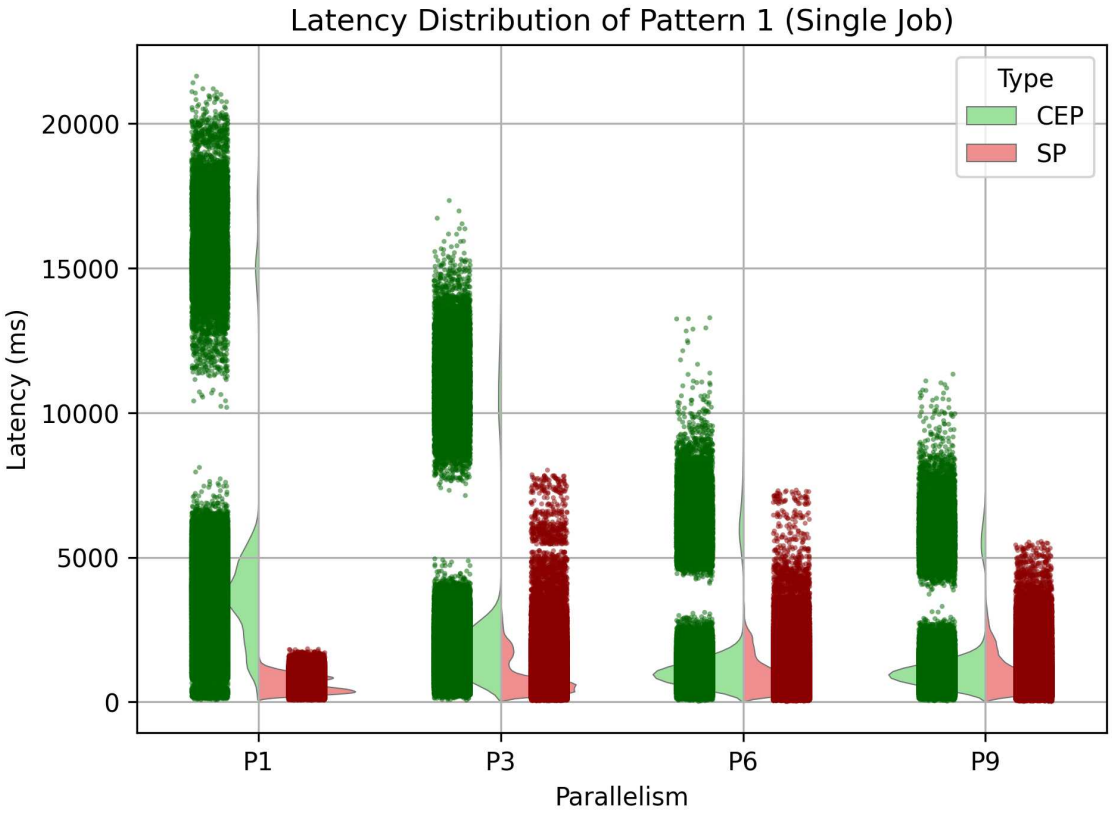


Figure A.1: Pattern 1: Single-job SP and CEP latency distribution

A.2 Multi-Job Results for Pattern 1

Figure A.2 presents a violin plot illustrating the latency distribution of CEP and SP for Pattern 1 multi-job, with individual points overlaid to highlight the outliers.

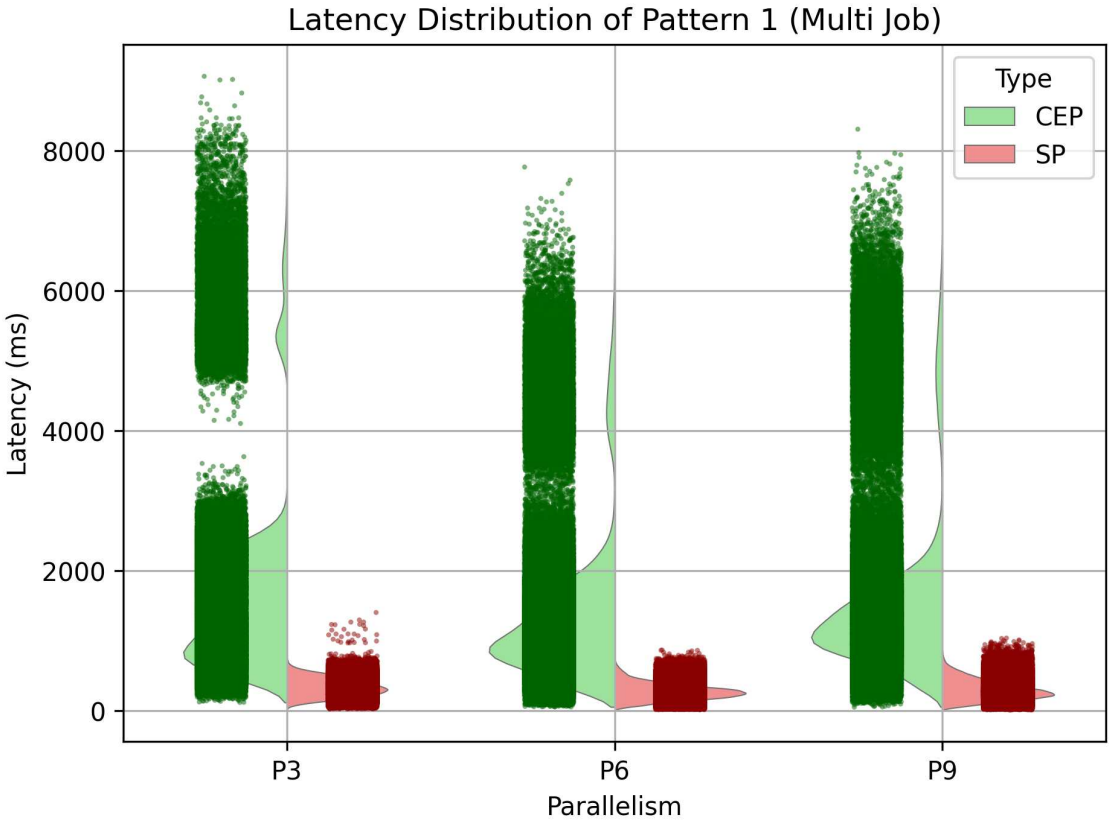


Figure A.2: Pattern 1: Multi-job SP and CEP latency distribution

A.3 Single-Job Results for Pattern 2

Figure A.3 presents a violin plot illustrating the latency distribution of CEP and SP for Pattern 2 single job, with individual points overlaid to highlight the outliers.

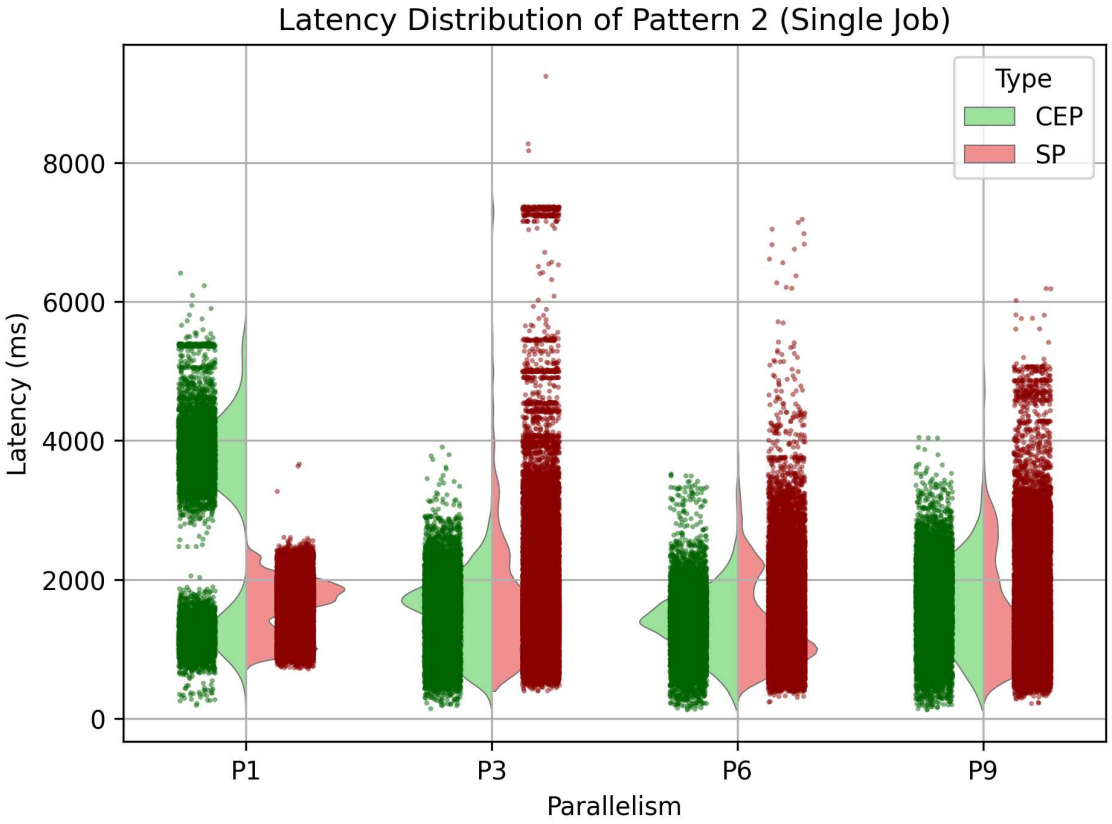


Figure A.3: Pattern 2: Single-job SP and CEP latency distribution

Figure A.4 presents a violin plot illustrating the latency distribution of CEP and SP for a 1-phase meter of Pattern 2 single job, with individual points overlaid to highlight the outliers.

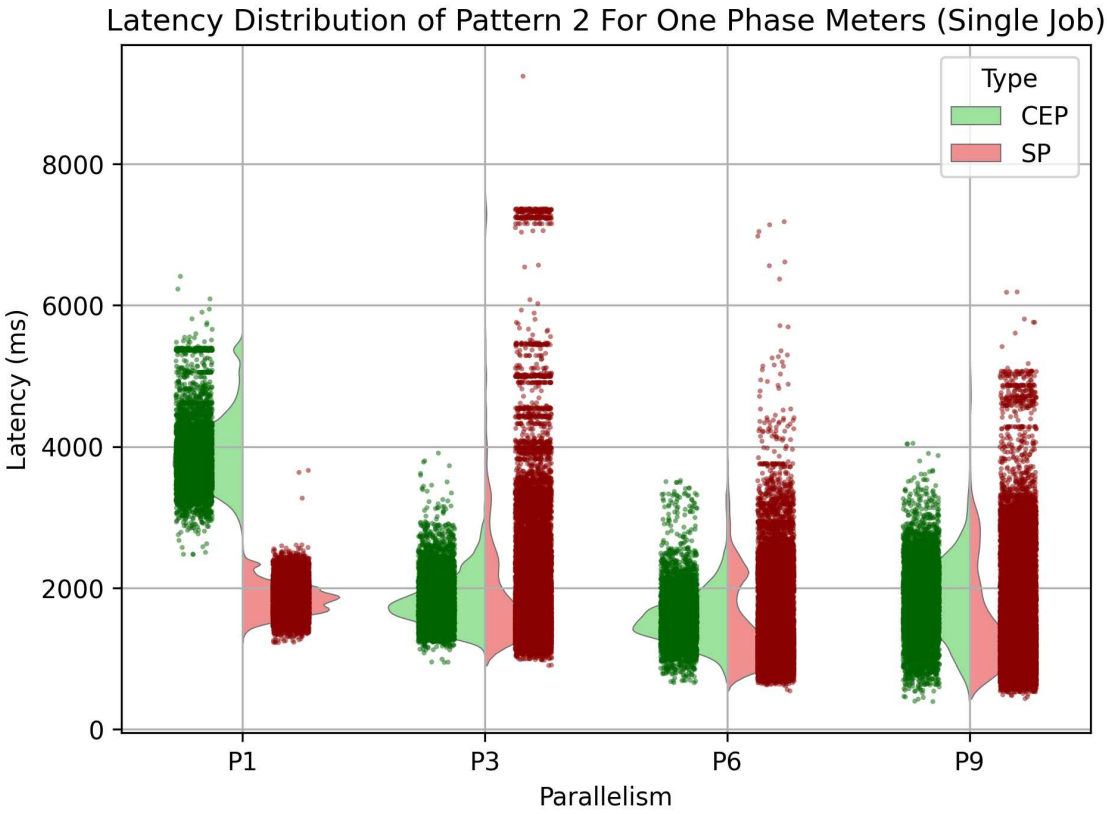


Figure A.4: Pattern 2: Single-job SP and CEP latency distribution for 1-phase meters

Figure A.5 presents a violin plot illustrating the latency distribution of CEP and SP for a 3-phase meter of Pattern 2 single job, with individual points overlaid to highlight the outliers.

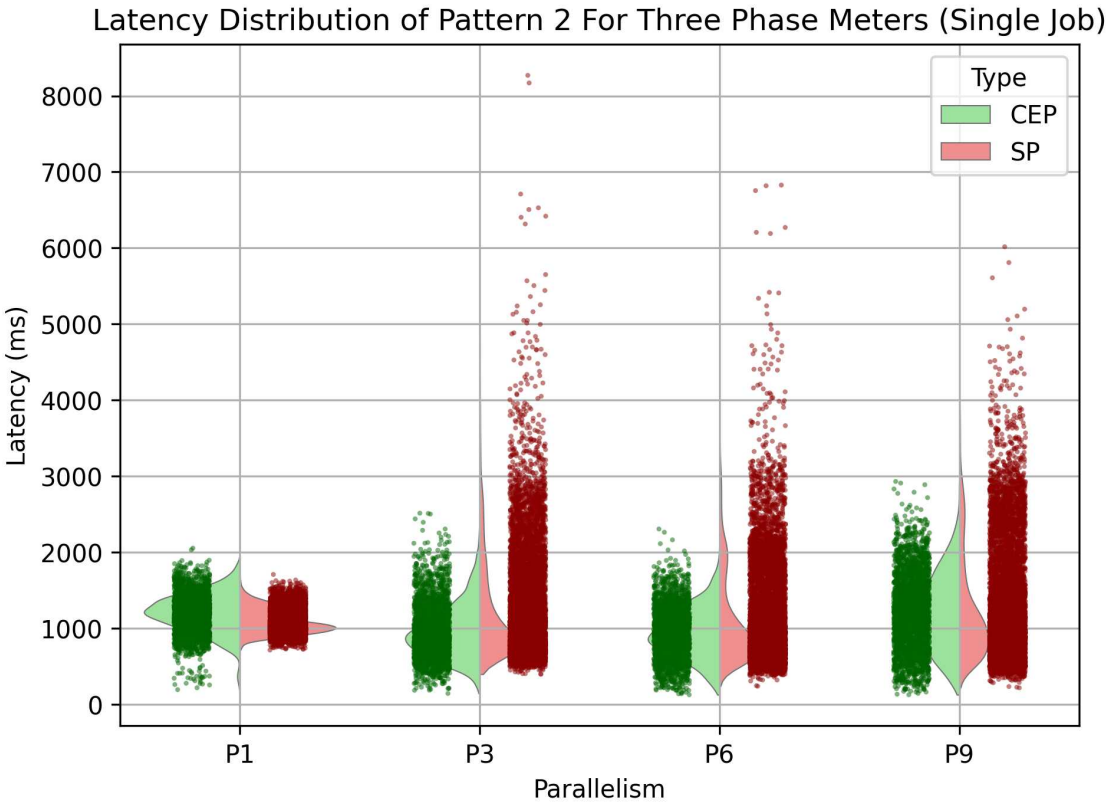


Figure A.5: Pattern 2: Single-job SP and CEP latency distribution for 3-phase meters

A.4 Multi-Job Results for Pattern 2

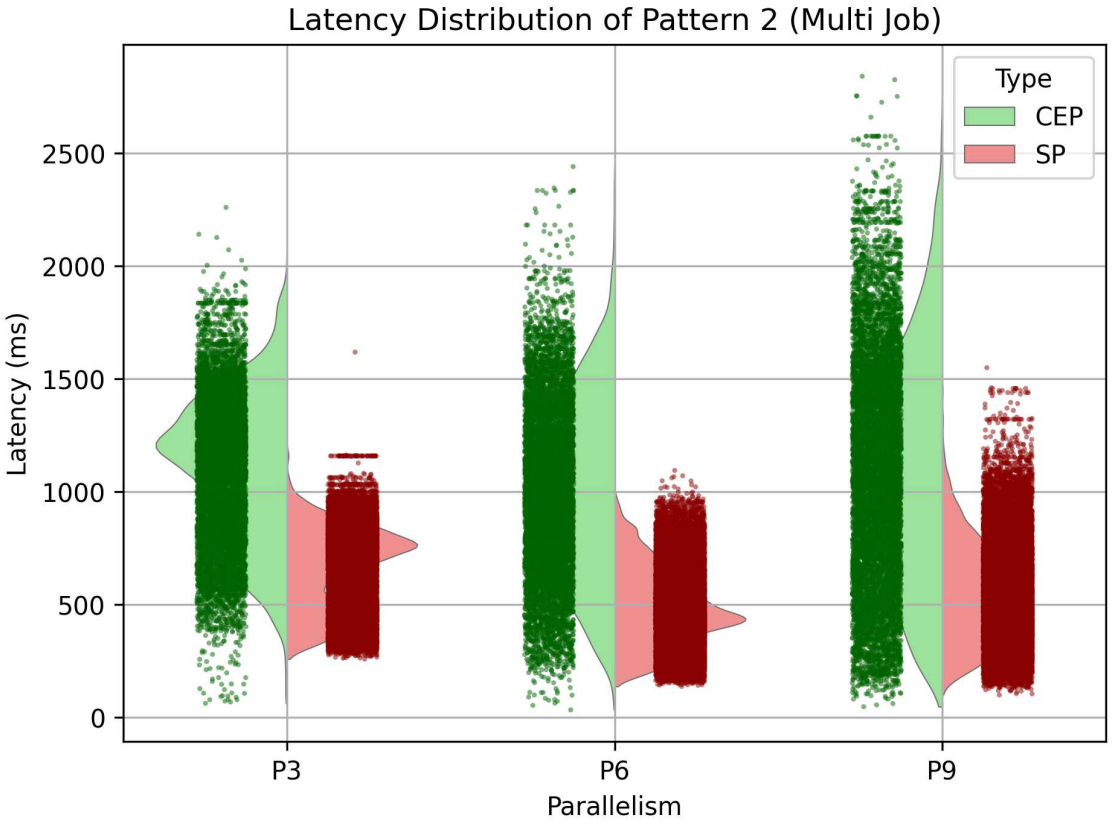


Figure A.6: Latency Distribution For Pattern 2 Multi Job

A.5 Single-Job Results for Pattern 3

Figure A.7 presents a violin plot illustrating the latency distribution for Pattern 3 single job, with individual points overlaid to highlight the outliers.

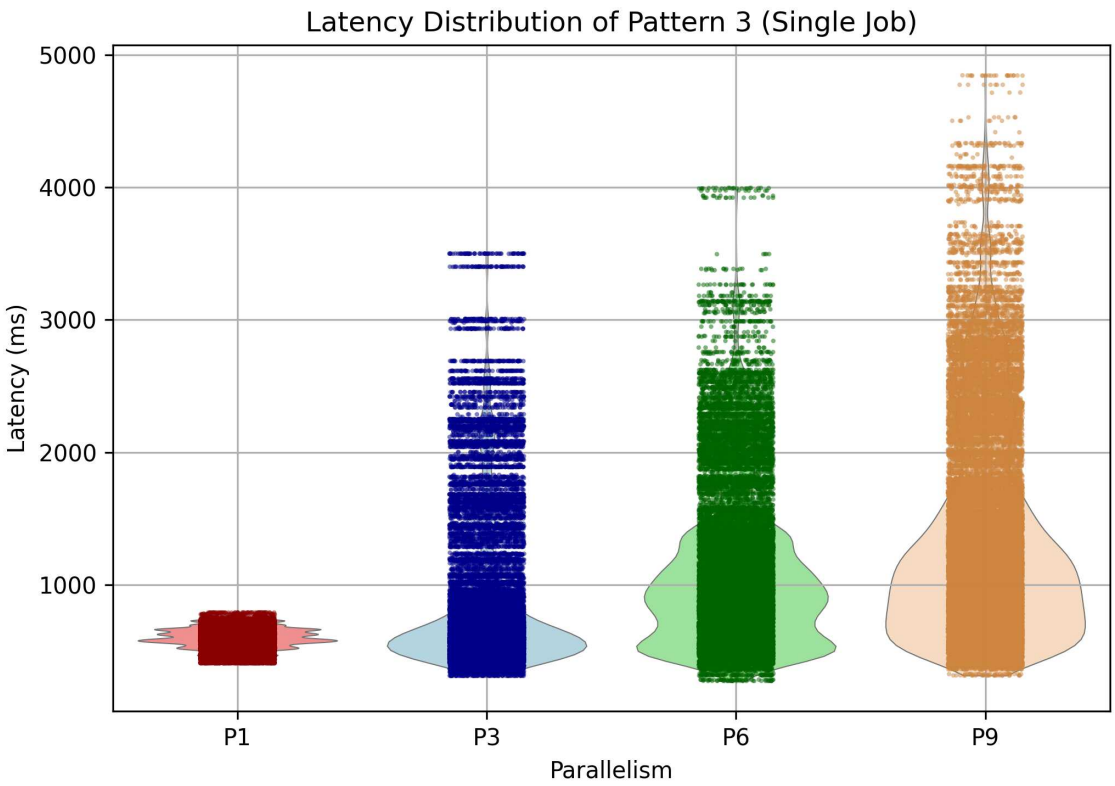


Figure A.7: Pattern 3: Single-job latency distribution

A.6 Multi-Job Results for Pattern 3

Figure A.8 presents a violin plot illustrating the latency distribution for Pattern 3 multi-job, with individual points overlaid to highlight the outliers.

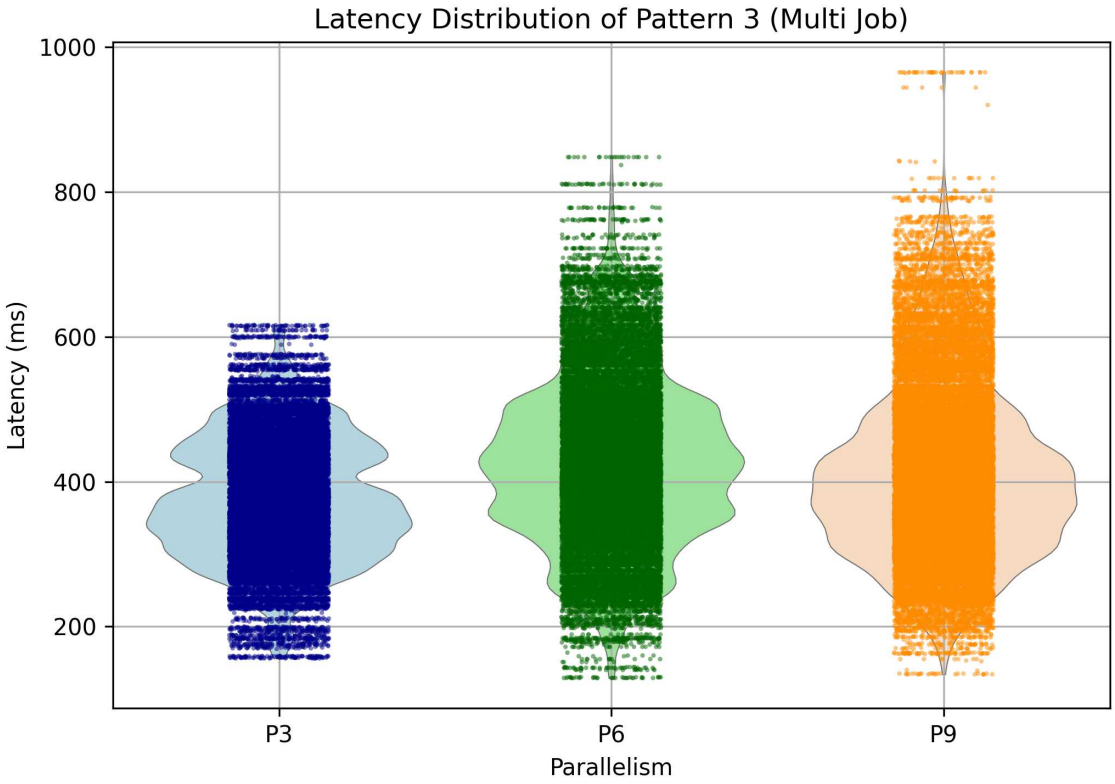


Figure A.8: Pattern 3: Multi-job latency distribution