IoT

MDPI

Article

PatternStudio: A Neuro-Symbolic Framework for Dynamic and High-Throughput Complex Event Processing

Jesús Rosa-Bilbao

UCASE Software Engineering Research Group, Department of Computer Science and Engineering, University of Cadiz, Avda. de la Universidad de Cadiz 10, Puerto Real, 11519 Cadiz, Spain; jesus.rosa@uca.es

Abstract

Complex Event Processing (CEP) is essential for real-time analytics in domains such as industrial IoT, cybersecurity, and financial monitoring, yet CEP adoption is still hindered by the difficulty of authoring temporal rules and by rigid redeployment workflows. This paper presents PatternStudio, a neuro-symbolic CEP framework that translates natural language specifications into validated event-processing patterns and executes them on a deterministic Apache Flink-based runtime without interrupting service. The generative layer is constrained to produce a typed intermediate representation, while the symbolic layer enforces validation and runtime execution guarantees. We evaluate the prototype as a single-node system-characterization study on commodity hardware representative of edge and near-edge gateways rather than microcontroller-class devices. Under this setting, PatternStudio reaches 47,910 events per second at 250 active rules while maintaining a bounded memory footprint between 1.6 GB and 1.9 GB during the reported runs. Beyond 500 active rules, throughput degradation is driven primarily by CPU saturation and alert amplification, which also explains the sharp increase in tail latency. Additional measurements with parallelism 4, a static baseline, and a two-stage NL-to-IR evaluation further show that the architecture remains functional under partitioned execution, incurs moderate dynamic-orchestration overhead, preserves rule structure reliably under natural-language authoring, and supports interchangeable LLM backends at the semantic front end.

Keywords: complex event processing; neuro-symbolic AI; stream processing; Apache Flink; dynamic orchestration

1. Introduction

The exponential growth of streaming data has progressively shifted big data analytics from offline batch processing toward real-time and continuous insight extraction. In this context, Complex Event Processing (CEP) has emerged as a foundational technology for detecting higher-level situations—such as temporal sequences, aggregations, and correlations—over unbounded data streams [1,2]. CEP plays a critical role in mission-critical domains, including industrial automation, cybersecurity, and financial monitoring, where timely and deterministic event detection is essential.

This problem is especially relevant to the Internet of Things (IoT), where large numbers of heterogeneous sensors, gateways, and actuators continuously emit fine-grained telemetry that is only useful if it can be interpreted in context and transformed into timely decisions [3–5]. In IoT deployments, raw events rarely have operational value in isolation. Temperature samples may need to be sustained above a threshold before an alarm is justified; vibration bursts may need to be correlated with machine state changes before

Check for updates

Academic Editor: Amiya Nayak

Received: 16 March 2026
Revised: 16 April 2026
Accepted: 20 April 2026
Published: 22 April 2026

Copyright: © 2026 by the author. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

maintenance is triggered; and security-relevant signals such as failed logins, privilege escalations, and network anomalies may need to be combined across devices and time windows before an incident response workflow starts. For this reason, CEP has long been viewed as a natural processing layer for IoT architectures, enabling event abstraction, near-real-time decision making, and event-driven automation over continuous sensor streams [6–8].

In the architecture considered in this paper, PatternStudio is not executed on tiny sensing devices. Instead, it is intended for the aggregator, gateway, edge server, or cloud tier that receives raw IoT telemetry and must interpret it with low operational friction. This distinction matters because the proposed neuro-symbolic stack assumes enough resources to host Kafka, Flink, and the control API, which is realistic for edge and near-edge gateways but not for microcontroller-class nodes.

Despite its maturity, the practical adoption of CEP systems is hindered by a persistent usability–performance trade-off. High-performance CEP engines offer strong execution guarantees but impose substantial cognitive and operational burdens on users, limiting their accessibility and adaptability in dynamic environments.

Existing CEP solutions largely fall into two categories, each exhibiting structural limitations. White-box CEP engines, such as Apache Flink CEP [9], Siddhi [10], and Esper [11], provide deterministic execution semantics and high throughput. However, they require users to express detection logic using low-level DSLs or host-language APIs. As a result, domain experts—such as shop-floor engineers, IoT operators, or security analysts—are often unable to directly author or modify complex temporal patterns, creating a dependency on software engineers and slowing down operational response.

In addition to usability challenges, the deployment model of stream processing systems remains rigid. Most CEP applications adhere to the Dataflow Model [12], where processing graphs are treated as immutable. Consequently, modifying a detection rule typically entails stopping the job, recompiling the pipeline, and redeploying the system. This stop-the-world approach is incompatible with scenarios requiring continuous operation, such as intrusion detection, industrial monitoring, and patient monitoring systems, where downtime introduces critical blind spots and delayed responses [13,14].

At the opposite end of the spectrum, purely data-driven AI approaches for stream analytics—particularly deep learning models—have gained traction due to their expressive power and adaptability. However, these systems often function as black boxes, offering limited explainability and non-deterministic behavior [15]. In regulated or safety-critical domains, probabilistic predictions are frequently insufficient compared to auditable, rule-based detections that can be formally verified and justified.

The practical limitations motivating this work can therefore be summarized as follows:

- Authoring barrier: deterministic CEP engines remain difficult for non-programmer domain experts to configure.
- Operational rigidity: updating CEP logic often requires redeployment, which is undesirable in always-on IoT and monitoring scenarios.
- Explainability gap: purely neural stream-analysis approaches do not provide the same auditable behavior as symbolic rule execution.
- Deployment tension: IoT deployments need both semantic flexibility and bounded resource usage at gateway or near-edge tiers.

These limitations motivate the need for a new class of CEP systems that reconcile semantic accessibility, dynamic reconfigurability, and deterministic execution. In the IoT domain, this requirement is even stronger because decision latency, deployment flexibility, and rule correctness directly affect monitoring, automation, and edge-resource utilization. To this end, we introduce PatternStudio, a neuro-symbolic CEP architecture [16] that

combines the semantic flexibility of Generative AI with the execution rigor of symbolic stream processing.

The system does not use an autonomous AI agent that directly acts on the runtime. Instead, the generative component is deliberately narrow in scope: it interprets natural-language specifications and proposes a typed intermediate representation (IR), while validation, persistence, and execution remain fully deterministic. This separation is important because it preserves auditability and allows the symbolic layer to reject incomplete, hallucinated, or semantically inconsistent rule definitions before they reach the stream processor.

The neuro-component leverages Large Language Models (LLMs) to interpret natural-language pattern specifications (e.g., "Alert if temperature exceeds 75 °C for 60 s") and translate them into a formally constrained Intermediate Representation (IR) [17]. The symbolic component, built on a customized Apache Flink execution engine, deterministically evaluates these patterns over live data streams. Crucially, PatternStudio introduces a Dynamic Graph Injection mechanism that enables the insertion, modification, and removal of CEP logic at runtime without interrupting the underlying dataflow.

The main contributions of this paper are summarized as follows:

- Neuro-Symbolic Translation Layer: We propose a translation pipeline that grounds LLM-generated specifications into strictly validated, JSON-based CEP pattern definitions, ensuring syntactic correctness and executability while mitigating hallucination risks [18].
- Zero-Downtime CEP Orchestration: We design and implement a runtime reconfiguration mechanism based on Flink's Broadcast State, enabling dynamic adaptation of CEP operator logic without explicit job interruption.
- IoT-Oriented Event Processing Architecture: We position PatternStudio as an event interpretation and orchestration layer for IoT gateway, edge-server, and near-edge scenarios, where raw telemetry must be transformed into actionable events under latency and deployment constraints.
- Single-Node Empirical Characterization: We conduct a descriptive performance evaluation on commodity hardware, identifying the configuration that yields the best observed throughput (250 active rules, 47,910 events/s) and the saturation region beyond 500 active rules, which is dominated by CPU contention and alert amplification.

The remainder of this paper is organized as follows. Section 2 introduces the underlying technologies, and Section 3 reviews prior work in CEP for IoT, dynamic stream processing, and AI-assisted rule engineering. Section 4 presents the materials and methods, including the PatternStudio architecture and the evaluation protocol. Section 5 reports the empirical results, Section 6 discusses their implications and limitations, and Section 7 concludes the paper.

2. Background

To establish the technical foundations of the proposed system, this section reviews the principles of stateful distributed stream processing, the role of CEP in IoT architectures, and the emerging paradigm of Neuro-Symbolic Artificial Intelligence.

2.1. Stateful Stream Processing and Complex Event Processing

CEP extends conventional stream processing beyond simple extract–transform–load (ETL) operations by focusing on the detection of higher-level temporal and logical patterns across multiple event streams [2]. As formalized in the survey by Cugola and Margara [1], CEP enables the derivation of semantic events from low-level observations through constructs such as temporal ordering, aggregation, and causal relationships.

Modern CEP engines, including Apache Flink [9], are built on stateful stream processing abstractions. Rather than treating events in isolation, these systems maintain a persistent operator state—typically scoped per key (e.g., sensor identifier or transaction ID)—and manage it fault-tolerantly via distributed snapshotting mechanisms [19]. This design allows CEP operators to evaluate complex temporal logic, including sliding and tumbling windows [20], interval joins, and non-deterministic event sequences, while preserving strong consistency guarantees.

Despite these capabilities, the dominant execution model of stream processing frameworks remains largely static. Following the Dataflow Model [12], the logical computation graph is compiled into a fixed physical execution plan prior to runtime. While this immutability enables high throughput and Exactly-Once processing semantics [21], it significantly constrains adaptability. In practice, modifying CEP logic requires a savepoint–stop–redeploy cycle, which introduces downtime, operational complexity, and risk. Such disruptions are unacceptable in continuous monitoring scenarios, including industrial control systems and real-time security analytics [13].

2.2. CEP as an IoT Data Interpretation Layer

These CEP capabilities align closely with the operational requirements of IoT systems. IoT platforms continuously ingest large volumes of heterogeneous sensor data, device states, and control signals that must be transformed into actionable situations with bounded latency [3,4]. In this setting, CEP acts as the interpretation layer between raw telemetry and meaningful business events, supporting event abstraction, context-aware filtering, and event-driven automation loops over streaming device data [6–8].

The connection becomes even stronger in edge and fog environments. Pushing CEP operators closer to data sources can reduce round-trip delay, lower core-network traffic, and enable local decision-making in time-sensitive applications such as industrial control, environmental monitoring, and intrusion detection [14,22]. Consequently, CEP should be understood not merely as a generic stream analytics technique, but as one of the enabling mechanisms through which IoT systems derive timely knowledge from raw event streams and trigger automated reactions.

2.3. Neuro-Symbolic Artificial Intelligence

Neuro-Symbolic Artificial Intelligence is a hybrid paradigm that aims to combine the perceptual and learning strengths of neural models with the rigor, interpretability, and formal guarantees of symbolic reasoning systems [16]. This dichotomy is frequently interpreted through Kahneman's cognitive framework of System 1 (fast, intuitive reasoning) and System 2 (slow, deliberate reasoning), as further articulated in contemporary AI research [23].

2.3.1. Neural Component

LLMs, such as GPT-4 [17] and Gemini [24], exhibit remarkable capabilities in natural language understanding and semantic abstraction. These models are particularly effective at translating unstructured human intent into structured representations. However, their probabilistic nature introduces stochasticity and the risk of hallucinations [18,25], leading to outputs that may be syntactically valid yet semantically incorrect. As a result, LLMs cannot be reliably entrusted with the direct execution or control of safety-critical systems.

2.3.2. Symbolic Component

Symbolic reasoning systems—including CEP engines and knowledge-graph-based frameworks [26]—offer deterministic execution, explainability, and verifiability. Their behavior can be formally reasoned about, audited, and reproduced, making them well suited

for regulated and mission-critical environments. However, these systems are traditionally difficult to configure and evolve, requiring expertise in formal specification languages (e.g., EPL or SQL-like pattern languages), which limits accessibility and hampers rapid adaptation [27].

2.4. Neuro-Symbolic Stream Processing

PatternStudio leverages the complementary strengths of these paradigms by adopting a neuro-symbolic stream processing architecture. In this design, the LLM functions as a semantic translation layer (Neural System 1), converting natural language intent into a constrained IR. This representation is subsequently validated and executed by a deterministic CEP engine (Symbolic System 2), ensuring correctness, explainability, and runtime efficiency. This separation of concerns enables semantic flexibility at the user interface level while preserving strict execution guarantees at the system core.

3. Related Work

The challenge of making CEP systems more accessible, flexible, and dynamically reconfigurable has been addressed from multiple perspectives in prior work. Because the editor explicitly requested a clearer relation to IoT, we organize the literature into four complementary lines: CEP as an IoT processing layer, distributed and edge/fog deployments for IoT, usability-oriented CEP systems, and AI-assisted or adaptive pattern engineering.

3.1. CEP as an IoT Processing Layer

The relationship between CEP and IoT is well established in the literature. IoT architectures are characterized by continuous streams of timestamped measurements, device states, and control signals that must be transformed into actionable information under latency and bandwidth constraints [3–5]. In this context, CEP acts as the event interpretation layer that converts raw telemetry into business-level situations, alarms, and automation triggers [6]. Bok et al. [7] specifically studied CEP for sensor stream data, showing how operator-aware optimization can improve processing efficiency for IoT-generated event streams. More recently, Gkoulis et al. [8] framed CEP as the mechanism that progressively transforms raw IoT events into meaningful business events in an edge-oriented smart farming architecture. These studies confirm that CEP is not peripheral to IoT systems; rather, it is one of the core mechanisms through which IoT data acquires operational meaning.

3.2. Distributed CEP for Edge and Fog IoT Deployments

Because IoT environments are latency-sensitive and geographically distributed, several works have studied how CEP should be deployed outside centralized cloud environments. Bezerra et al. [14] evaluated complex event processing in fog-based IoT systems for smart agriculture and showed that pushing event processing closer to data sources can support faster decision making while reducing cloud dependency. Complementing these deployment studies, Gkoulis et al. [22] proposed a hybrid simulation platform for evaluating quality-aware complex events in IoT environments, underscoring that realistic CEP assessment in IoT should account not only for throughput but also for timeliness, completeness, and the quality of derived events. PatternStudio aligns with this edge/fog-oriented view by targeting commodity hardware and runtime adaptability, but differs in focusing on zero-downtime rule management and natural language-driven orchestration.

3.3. Usability-Oriented and Dynamic CEP Systems

Traditional CEP engines such as Esper [11], Siddhi [10], and Apache Flink CEP [9] remain the de facto standards for high-throughput event detection. Their main strengths are mature pattern semantics and efficient execution, but they typically require technical

users to encode patterns in DSLs or host-language APIs. This usability barrier is particularly problematic in IoT scenarios, where domain experts such as plant operators, clinicians, or security analysts are often not stream-processing specialists.

To address this gap, earlier work proposed higher-level programming abstractions. Gökalp et al. [28] introduced a visual programming framework for IoT-centric distributed CEP, aiming to lower the expertise required to define and deploy CEP applications. Likewise, model-driven approaches such as MEdit4CEP and MEdit4CEP-SP seek to shield users from low-level implementation details while retaining event-driven execution capabilities [29,30]. These works significantly improve usability, but they still rely on dedicated visual or model-driven tooling and do not directly support natural language interaction or deterministic runtime injection of new rules into a running Flink-based pipeline.

3.4. AI-Assisted and Adaptive Pattern Engineering

Recent research has explored the integration of CEP with machine learning and AI to cope with dynamic environments. Roldán-Gómez et al. [31] showed that ML can help construct and refine CEP rules for real-time IoT attack recognition, reducing manual rule engineering in cybersecurity scenarios. This line of work is closely related to PatternStudio because it reduces rule-authoring effort, but it is domain-specific and focused on security analytics rather than on a general-purpose runtime orchestration framework.

At a broader level, LLM-based software assistance has demonstrated strong capability for code and configuration generation [17,32,33], although unconstrained generation remains vulnerable to hallucination and verification problems [18,25]. PatternStudio addresses this limitation by constraining the generative component to produce a schema-validated intermediate representation rather than executable code.

3.5. Comparison with Closely Related Approaches

Table 1 summarizes the positioning of PatternStudio with respect to closely related systems. The comparison highlights that the closest prior works usually optimize one or two dimensions at a time: efficient CEP execution, IoT-oriented deployment, higher-level authoring, or AI-assisted rule generation. By contrast, PatternStudio aims to combine all four dimensions in a single architecture.

Table 1. Qualitative comparison between PatternStudio and closely related CEP approaches in IoT-oriented settings.

Approach	IoT-Oriented	Higher-Level Authoring	Runtime Rule Updates	Deterministic Symbolic Execution	Main Emphasis
Esper/Siddhi/Flink CEP [9–11]	Partial	No	Limited/static	Yes	Efficient CEP execution
Gökalp et al. [28]	Yes	Visual DSL	No explicit zero-downtime orchestration	Yes	Usability and distributed IoT CEP
Bezerra et al. [14]	Yes	No	No	Yes	Fog/edge deployment for IoT CEP
Roldán-Gómez et al. [31]	Yes	ML-assisted rule generation	Domain-specific adaptation	Partial	IoT cybersecurity rule generation
Gkoulis et al. [8,22]	Yes	Parametric CEP abstractions	Partial	Yes	Event transformation and quality-aware IoT CEP
PatternStudio (this work)	Yes	Natural language to validated IR	Yes	Yes	Neuro-symbolic, zero-downtime CEP orchestration

3.6. Summary and Research Gap

In summary, prior work has addressed high-performance CEP execution, IoT-specific deployment, higher-level authoring, and AI-assisted rule engineering, but usually emphasizes only a subset of these dimensions at once. We therefore position PatternStudio not as a definitive closure of this research space, but as an attempt to combine four dimensions that

are still rarely discussed together in one implementation: (i) IoT-oriented event transformation and edge-compatible deployment assumptions; (ii) natural-language-driven pattern specification; (iii) runtime reconfiguration of CEP logic without service interruption; and (iv) deterministic execution through a validated symbolic layer.

4. Materials and Methods

This section describes the architecture, implementation details, and experimental protocol used to evaluate PatternStudio.

4.1. System Architecture

To address the rigidity of traditional CEP engines and the non-deterministic behavior of purely data-driven AI systems, PatternStudio adopts a decoupled, three-tier architecture (Figure 1). The design follows a strict separation of concerns, partitioning responsibilities across three orthogonal planes: the Neuro-Symbolic Control Plane, the Streaming Data Plane, and the Deterministic Execution Plane. This decomposition enables semantic flexibility at the interface level while preserving strong execution guarantees in the data plane.

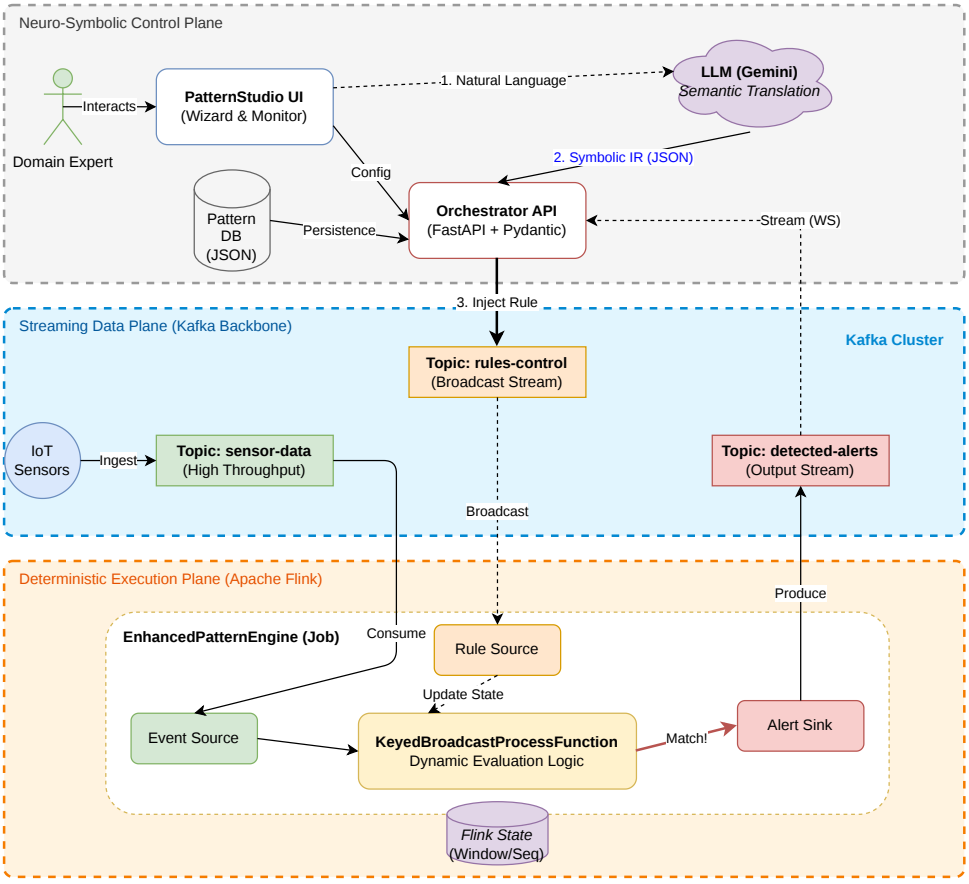


Figure 1. High-level architecture of PatternStudio.

4.2. Neuro-Symbolic Control Plane

The Neuro-Symbolic Control Plane mediates between human intent and executable CEP logic. Its primary responsibility is to transform unstructured natural-language specifications into formally constrained representations that can be safely deployed in a stream processing environment.

- Generative Translation Layer: User interaction is facilitated through a web-based frontend, where domain experts express detection requirements in natural language.

An LLM is employed exclusively as a semantic translation component. In the evaluated implementation, this role is served by the `gemini-2.5-flash` model. Rather than operating as an autonomous agent, the model is constrained via a system prompt with strict schema expectations, ensuring that vague temporal expressions are mapped to explicit configuration parameters.

- Intermediate Representation and Validation: A central design principle of PatternStudio is that raw LLM outputs are never executed directly. Instead, all generated specifications must conform to a rigorously defined IR. The IR is formalized using strongly typed validation models, which enforce both syntactic correctness and domain-specific semantic constraints. For example, the validator ensures that sliding-window patterns include an aggregation operator and that sequence patterns contain at least two ordered steps. Any deviation from these constraints is rejected prior to deployment. This validation layer functions as a symbolic guardrail, insulating the deterministic execution engine from the stochastic behavior of the neural component.

4.3. Streaming Data Plane

The Streaming Data Plane is responsible for propagating both event data and control logic in a consistent and fault-tolerant manner. PatternStudio deliberately abandons request–response-based rule updates in favor of a log-based architecture, treating rules as first-class streaming entities.

- Dual-Stream Ingestion Model: The system defines two logically independent input streams: `sensor-data`, a high-throughput stream carrying raw telemetry or transactional events; and `rules-control`, a low-throughput, high-priority stream conveying rule creation, update, and deletion commands. This separation ensures that bursts of data events do not require a complete job restart to propagate control updates.
- Sequential Consistency of Rule Updates: By modeling rule updates as an ordered log, the system aims to preserve sequential consistency across all parallel execution instances. Each worker receives rule updates in the same order and stores them in local broadcast state, reducing the risk of split-brain scenarios in which different workers evaluate events under divergent rule sets.

4.4. Deterministic Execution Plane

The Deterministic Execution Plane constitutes the runtime core of PatternStudio. It is implemented on top of Apache Flink using a generic meta-interpreter capable of executing CEP logic dynamically from the validated IR, rather than from statically compiled code.

- Broadcast State-Based Orchestration: To enable zero-downtime reconfiguration, PatternStudio leverages Flink's `KeyedBroadcastProcessFunction`. The `rules-control` stream is broadcast to all parallel task instances and stored in local broadcast state. Upon arrival of a new or updated rule, the state is updated immediately, making the new logic available for subsequent events without interrupting the running job or rebuilding the execution graph.
- Dynamic Evaluation Strategy: Incoming events are evaluated against the active rule set maintained in broadcast state. For each event, the engine dispatches execution to specialized evaluators based on the rule type defined in the IR: stateless evaluation for immediate predicates, and stateful evaluation for temporal patterns such as tumbling, sliding, session, join, and sequence rules.
- Interval Joins, Sequential Patterns, and Cleanup: Complex correlation patterns are implemented explicitly to preserve runtime flexibility. Interval joins maintain bounded temporal buffers, pruning expired events according to user-defined constraints to prevent unbounded state growth. Sequential patterns are realized as deterministic

state machines that track progression through ordered steps and reset when temporal constraints are violated.

Algorithms 1 and 2 summarize the two operational mechanisms identified as central to understanding the runtime behavior: dynamic graph injection through broadcast updates, and per-event evaluation with state pruning.

Algorithm 1 Dynamic rule update propagation

```
procedure ProcessBroadcastRule(ruleMsg)
    rule ← parse and validate ruleMsg
    state ← broadcast-state handle
    if rule.action = DELETE then
        remove rule.id from state
        remove or mark stale any keyed state associated with rule.id
    else
        upsert rule.id ↦ rule in state
        expose the new version to all subsequent event evaluations
    end if
    emit audit log entry for traceability
end procedure
```

Algorithm 2 Per-event evaluation and expiration cleanup

```
procedure ProcessEvent(event)
    rules ← current broadcast-state snapshot
    for all rule ∈ rules do
        prune expired tuples, windows, or timers associated with rule
        if rule.window_type = simple then
            evaluate predicate directly on event
        else
            update keyed state buffers required by rule
            compute the corresponding window, join, or sequence condition
        end if
        if rule matches then
            emit alert with rule metadata and matched event context
        end if
    end for
end procedure
```

4.5. Implementation Details

The neuro-symbolic translation layer used in the evaluated prototype relies by default on the `gemini-2.5-flash` model to transform natural-language specifications into the constrained intermediate representation described above. However, this translation stage is intentionally externalized behind a provider-agnostic interface, so the semantic parser can be replaced by alternative LLM backends without modifying the deterministic execution layer. In all cases, generated outputs are validated against the PatternStudio schema before deployment. This design ensures that generative assistance remains auditable, interchangeable, and separated from the deterministic execution path.

The deterministic runtime is implemented on Apache Flink and uses Kafka as the backbone for both data and control traffic. The current prototype includes support for immediate predicates, tumbling and sliding windows, session conditions, interval joins, and ordered sequences. Broadcast state is used to distribute rule updates, while keyed state stores the per-rule buffers, timers, and sequence progression needed by temporal operators.

4.6. Experimental Setup

To characterize the current prototype under controlled conditions, all reported experiments were conducted on commodity hardware rather than on a multi-node cluster.

- Hardware Configuration:
  - Processor: 13th Gen Intel® Core™ i7-13620H @ 2.40 GHz (10 cores: 6 performance cores and 4 efficiency cores).
  - Memory: 16 GB DDR5 RAM.
  - Storage: NVMe SSD to minimize I/O interference during Kafka persistence.
- Software Stack:
  - Containerized deployment using Docker.
  - Apache Flink v1.17, configured with a job-level parallelism of 1 in order to isolate the single-worker behavior of the prototype.
  - Apache Kafka v3.5 as the ingestion and control backbone.

This configuration should be interpreted as representative of an edge gateway, industrial PC, or near-edge server, not of a low-power sensor node or microcontroller.

4.7. Methodology

We developed a custom benchmarking tool to generate synthetic workloads and collect performance metrics. The evaluation follows a sustained-load stress testing protocol designed to identify stable operating regions under increasing rule cardinality:

1. Rule Injection: For each scenario, N distinct CEP rules were injected dynamically into the running system.
2. Warm-Up Phase: A 10-s warm-up period was introduced to allow Just-In-Time compilation and internal Flink optimizations to stabilize.
3. Stress Phase: The system was subjected to a continuous high-rate input stream exceeding 50,000 events per second.
4. Measurement: We recorded throughput (events processed per second), end-to-end latency (P95 and P99 percentiles), aggregate CPU utilization, memory consumption, and the alert-to-event ratio observed in the output stream.

CPU utilization is reported as aggregate utilization across all cores, following the Docker-style convention used by the measurement tool. For this reason, values above 100% indicate that more than one logical core was actively used; for example, 150% corresponds approximately to one and a half fully utilized cores on average.

Because the current study is based on one sustained-load characterization run per rule-cardinality setting, the reported analysis is descriptive rather than inferential. Accordingly, we focus on trend interpretation, relative performance changes, and cross-scenario comparisons rather than on hypothesis testing. We also executed three additional measurements on the same hardware: (i) a multi-slot configuration with four Kafka partitions and job parallelism 4, (ii) a fixed-rule static baseline for the 250-rule simple-pattern workload, and (iii) a two-stage gold-set evaluation of NL-to-IR translation. The latter comprises both an in-system assessment of the default `gemini-2.5-flash` translator on five representative prompts and a backend-substitution study on an expanded 19-prompt set using multiple external LLMs accessed through the same schema-constrained evaluation pipeline. These additions broaden the evidential basis of the manuscript by covering the distributed, orchestration, and authoring dimensions more directly.

5. Results

This section reports the observed throughput, latency, and resource utilization of PatternStudio under increasing rule cardinality.

5.1. Performance Results

Table 2 summarizes the observed performance across six configurations, ranging from 10 to 1000 concurrently active rules.

Table 2. Observed performance metrics under increasing rule cardinality. CPU is reported as aggregate multi-core utilization, so values above 100% indicate concurrent usage of more than one core.

Rules	Throughput (ev/s)	P95 Lat. (ms)	P99 Lat. (ms)	CPU Agg. (%)	Memory (MB)	Alerts/Event
10	40,325.74	3185.69	4592.40	38.81	1607.25	3.54
50	43,967.29	16,550.79	17,153.78	71.14	1875.84	30.94
100	42,888.25	25,923.63	26,987.80	104.43	1937.75	51.10
250	47,910.65	25,341.23	26,371.48	127.46	1985.32	47.17
500	17,775.24	112,143.41	116,784.76	148.68	1867.81	104.03
1000	14,303.56	125,281.07	127,654.34	146.58	1770.18	108.27

5.1.1. Throughput Profile and Operational Sweet Spot

Contrary to a naive expectation of monotonic degradation with increasing rule complexity, the results reveal a non-linear scalability profile. Throughput increases from 10 to 250 active rules, reaching a peak of approximately 47,900 events per second (Figure 2). Relative to the 10-rule configuration, the 250-rule configuration improves throughput by 18.81%, indicating that the framework benefits from higher computational density before saturation effects dominate.

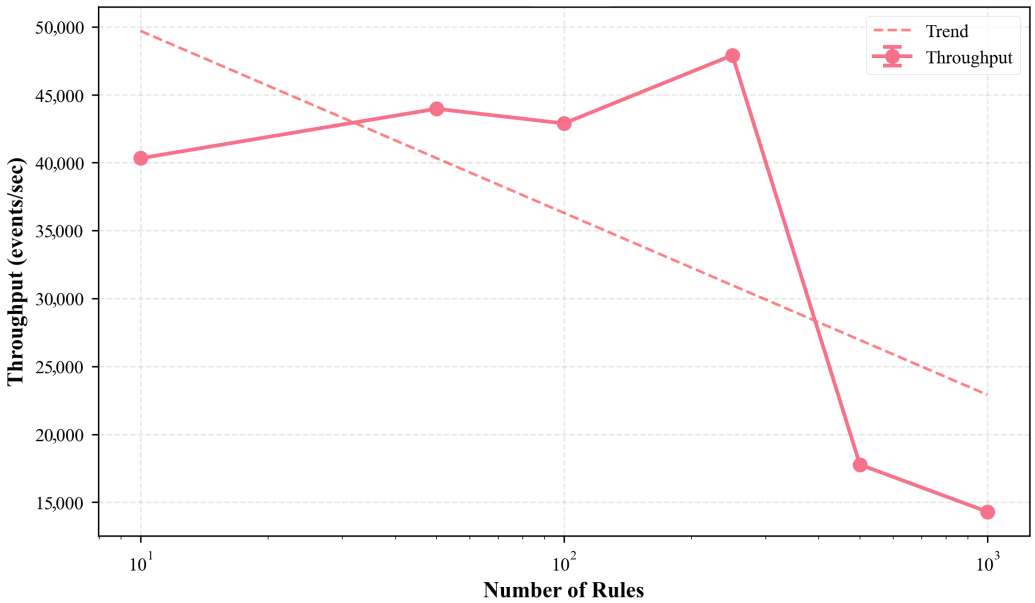


Figure 2. System throughput versus rule cardinality.

This behavior can be attributed to the amortization of fixed ingestion overheads. With a small number of rules, Kafka consumption and framework-level bookkeeping dominate execution time. As rule cardinality increases, a larger fraction of CPU cycles is devoted to actual pattern evaluation, leading to more effective utilization of available cores. The observed peak at 250 rules therefore represents the best operating point in the present single-node characterization study, not a claim of general linear scalability.

5.1.2. Hardware Saturation Boundaries

Beyond 250 concurrent rules, the system exhibits a pronounced performance decline. At 500 active rules, throughput drops to approximately 17,800 events per second, a 62.90% reduction relative to the 250-rule peak, coinciding with aggregate CPU utilization approaching 150% of nominal single-core capacity (Figure 3). At 1000 active rules, throughput remains 70.15% below the peak value. This indicates that the current deployment becomes compute-bound and that additional rule evaluation directly competes for CPU time.

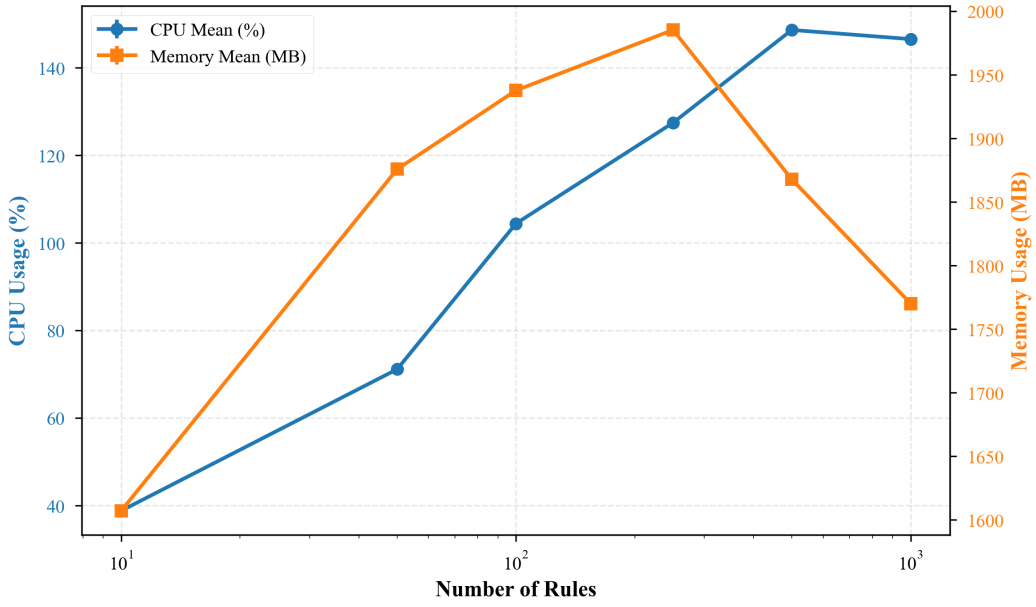


Figure 3. Aggregate CPU and memory consumption versus rule cardinality.

Latency degradation follows the same transition. Between 250 and 500 rules, P95 latency increases by 342.53% and P99 latency by 342.84%, confirming that the saturation point is visible not only in throughput but also in tail-latency behavior.

5.1.3. Why P95 and P99 Become So High

The very high P95/P99 values at 500 and 1000 rules are explained by two combined effects. First, the workload crosses a CPU saturation threshold, so incoming events wait longer in the operator pipeline before they can be evaluated. Second, the benchmark produces substantial alert amplification: the observed alert-to-event ratio grows from 47.17 at 250 rules to 104.03 at 500 rules and 108.27 at 1000 rules. In other words, once the rule set becomes large, each input event matches many active rules, so the runtime must both evaluate more predicates and emit many more output records per event. This queue buildup penalizes the tail of the latency distribution much more strongly than the median.

This observation is important for interpreting the current IoT relevance of the prototype. The reported overload scenarios are useful for capacity planning and saturation analysis, but their tail latencies are too high for many time-sensitive deployments. Accordingly, the practical message of these results is not that the framework is universally suitable for all edge workloads, but rather that the current single-node implementation remains efficient up to a moderate rule count and then requires either scale-out, rule prioritization, or admission control.

5.1.4. Memory Behavior Under Stress

Memory usage remains tightly bounded between 1.6 GB and 1.9 GB across the reported runs, with a mean of 1840.69 MB and a standard deviation of 135.48 MB. This

comparatively flat profile suggests that the state management strategy and expiration logic prevent immediate runaway growth under the evaluated workloads. However, because the experiments are short characterization runs rather than long-duration endurance tests, these observations should be interpreted as encouraging evidence of bounded short-run memory behavior rather than as a formal proof of no memory leaks or 24/7 stability.

5.1.5. Descriptive Interpretation and Scope

Given that each rule-cardinality configuration corresponds to a single sustained-load characterization run, the present study emphasizes descriptive rather than inferential statistics. Even under this constraint, the trends support three robust descriptive findings: (i) the best observed throughput occurs at 250 active rules; (ii) a marked saturation transition appears between 250 and 500 rules; and (iii) short-run memory usage remains comparatively stable across the evaluated scenarios.

5.1.6. Same-Host Parallel Validation

To validate the behavior of the orchestration model beyond parallelism 1, we repeated the same 250-rule and 500-rule workloads on the same host using four Kafka partitions for the data stream, four partitions for the alert stream, and Flink job parallelism 4. Table 3 summarizes this same-host parallel comparison.

Table 3. Comparison between the single-slot and multi-slot configurations on the same machine.

Mode	Rules	Throughput (ev/s)	P95 Lat. (ms)	CPU Agg. (%)	Memory (MB)
Single-slot	250	42,849.43	25,310.35	156.19	2118.26
Parallelism 4	250	43,665.74	26,184.95	162.92	1921.32
Single-slot	500	17,591.23	77,289.30	155.89	2200.62
Parallelism 4	500	21,233.74	76,792.45	155.09	1979.58

The multi-slot configuration does not materially change the 250-rule operating point, but it improves throughput by approximately 20.7% at 500 rules while slightly reducing P95 latency and memory footprint. This behavior is consistent with a same-host parallel deployment: at lower rule counts, partitioning, synchronization, and broadcast overhead offset most of the parallelism benefit, whereas under heavier load the extra execution slots begin to compensate for that overhead. Overall, the results show that the broadcast-state orchestration mechanism remains stable under a partitioned, parallel configuration on one machine. Cross-node behavior under network delay, failures, checkpoint coordination, and partition skew was not exercised in this setup and therefore remains outside the scope of the present validation.

5.1.7. Rule-Update Latency and Static Baseline

We also measured the delay between issuing a rule creation request and observing the first matching alert. In the single-slot configuration, the control-to-alert latency was 3339.81 ms, and the event-to-alert latency was 2314.73 ms. In the parallelism-4 configuration, the corresponding values were 4015.35 ms and 2985.74 ms. These measurements indicate that rule propagation remains observable within a few seconds in both settings, with the multi-slot path incurring the expected extra coordination cost associated with propagating updates across parallel task instances.

These measurements should be interpreted carefully because they capture operational observables rather than an isolated control-plane timestamp. Control-to-alert latency spans the entire interval from the API request that injects a rule until the first alert produced by that rule is observed downstream; it therefore includes control propagation, the arrival of at least one qualifying event, pattern evaluation, and alert emission. Event-to-alert

latency instead starts at the first probe event sent after rule creation and measures only the interval from that event emission to the first resulting alert. A pure rule-propagation latency would be narrower still, ending when all task instances have incorporated the new rule into broadcast state, but that internal timestamp is not exposed by the current runtime instrumentation. At the update boundary, events already being processed continue under the rule snapshot visible when they entered the operator, whereas subsequent events observe the updated rule set once the broadcast update has been applied by the corresponding task instance.

For the 250-rule simple-pattern workload, we also compared the dynamic engine against a fixed-rule static Flink baseline on the same hardware. Here it is important to distinguish between the canonical 250-rule point in Table 2 and the 250-rule single-slot point used in the matched baseline comparison. The former (47,910.65 events/s) comes from the main six-scenario characterization campaign reported in Table 2 and should be treated as the canonical reference for the throughput sweet spot of the original system-level study. The latter (42,849.43 events/s) comes from a separate follow-up run executed with the dedicated comparison harness used to measure the static and dynamic versions under the same experimental procedure. This matched follow-up run is therefore the appropriate reference for estimating orchestration overhead against the static baseline, even though its absolute value differs from the original characterization table.

Under this matched comparison setup, the static baseline processed 48,556.59 events/s, with P95 latency of 26,040.97 ms, aggregate CPU utilization of 121.48%, and memory usage of 2049.84 MB. Relative to the dynamic 250-rule follow-up run (42,849.43 events/s), this corresponds to an approximate throughput advantage of 13.3% for the static configuration. This result is useful because it provides a first estimate of the runtime overhead associated with dynamic orchestration and broadcast-state-based flexibility.

5.1.8. NL-to-IR Evaluation

The natural-language translation layer was evaluated in two complementary ways. First, the default in-system translator based on `gemini-2.5-flash` was evaluated on a five-prompt gold set covering representative rule-authoring requests. On this set, the exact-match rate was 80.0%, the structural-match rate was 100%, and the average manual correction burden was 0.2 fields per prompt. These results indicate that the schema-constrained translation layer can recover the intended rule structure reliably for common prompt patterns while keeping post-edit effort low.

Second, to verify that the translation layer is not tied to a single provider, we executed the same evaluation pipeline with alternative external LLM backends connected through the same schema-constrained adapter. Table 4 summarizes the observed results. On the expanded 19-prompt set, `openai/gpt-oss-120b:free` achieved the strongest overall balance among the tested substitute backends, reaching 10.53% exact match, 89.47% structural match, and zero API failures. `microsoft/phi-4` preserved structure in all 19 cases, but required more field-level correction to converge to the expected IR. `deepseek/deepseek-r1` showed lower robustness in this setup, with a structural-match rate of 63.16% and a non-zero API error rate.

Table 4. NL-to-IR evaluation of the default translator and alternative interchangeable LLM backends.

Model	Prompts	Exact Match (%)	Structural Match (%)	API Error (%)	Avg. Corrections
Gemini 2.5 Flash (default translator)	5	80.00	100.00	0.00	0.20
OpenAI GPT-OSS 120B (free)	19	10.53	89.47	0.00	2.79
Microsoft Phi-4	19	0.00	100.00	0.00	3.32
DeepSeek R1	19	0.00	63.16	36.84	3.05

Taken together, these results support two conclusions. First, the default translator already performs well on representative authoring requests when coupled with schema validation. Second, the translation layer is genuinely replaceable: alternative LLMs can be connected to the same pipeline without modifying the runtime, but their behavior differs substantially in terms of structural fidelity, exact semantic alignment, and operational robustness. This finding is important because it shows that PatternStudio's neuro-symbolic contribution lies in the validated interface and execution discipline, not in dependence on any single proprietary model.

5.1.9. Contextual Comparison with Prior IoT CEP Research

Direct head-to-head quantitative comparison with the state of the art remains difficult because closely related studies target different workloads, event schemas, deployment topologies, and evaluation metrics. For example, Bezerra et al. [14] emphasize fog-based decision support in smart agriculture, Gkoulis et al. [22] focus on quality-aware complex-event evaluation in IoT environments, and Gökalp et al. [28] prioritize usability and distributed IoT-centric authoring. Roldán-Gómez et al. [31] further show how AI assistance can reduce CEP rule-authoring effort in a domain-specific IoT security setting.

Accordingly, the fairest conclusion is qualitative: PatternStudio combines natural-language-driven authoring, deterministic symbolic execution, and runtime rule updates in a single architecture, while the present evaluation characterizes that architecture through single-node stress tests, a same-host multi-slot validation, a static baseline, and a gold-set translation assessment.

6. Discussion

The results support the central design hypothesis of PatternStudio: natural-language-driven CEP can remain practical in production-oriented settings when the neural component is restricted to semantic translation and the symbolic component retains full responsibility for validation and execution. In this configuration, semantic accessibility is improved without relinquishing determinism, which is especially relevant for operational environments that require auditable rule behavior.

The throughput profile further shows that the proposed architecture remains efficient on commodity hardware typical of edge gateways and near-edge servers. The peak observed at 250 active rules suggests that the framework can amortize fixed ingestion and orchestration overheads effectively before CPU saturation dominates. This is a useful systems result because it shows that introducing a neuro-symbolic control plane does not automatically make CEP execution impractical.

At the same time, the revised analysis makes the current limits of the prototype clearer. Beyond approximately 500 concurrent rules, latency and throughput degrade due to a combination of CPU saturation and alert amplification. This means that the present single-node deployment is compute-bound rather than memory-bound, and that overload mainly manifests as queueing delay in the tail of the latency distribution. For larger workloads, scale-out, operator parallelism, batching policies, or rule prioritization would be more appropriate than simply adding more rules to the same execution instance.

Another important result concerns semantic coverage and backend substitutability. The default neuro-symbolic translation layer produced an 80% exact-match rate and a 100% structural-match rate on the evaluated five-prompt gold set, which indicates that the schema-constrained design reliably preserves rule structure for representative temporal requests. The expanded multi-model evaluation further shows that this translation stage is not bound to one provider: GPT-OSS 120B preserved structure in 89.47% of the 19 evaluated prompts, Phi-4 in 100%, and DeepSeek R1 in 63.16%. This is relevant because it supports the

central claim that natural-language authoring can be integrated into CEP without allowing unconstrained generative output to reach the runtime. In that sense, the generative layer is best understood as a controlled authoring interface whose outputs remain auditable and symbolically validated before execution.

The same-host parallel measurements further clarify how the architecture behaves under parallel execution. On one machine, the runtime still processes rule updates correctly with partitioned streams and parallelism 4, and throughput improves by roughly 20.7% at 500 rules. The fact that the 250-rule case changes only marginally is also informative: at moderate load, the overhead of partitioning, coordination, and rule broadcast offsets most of the available parallelism gain, whereas at higher load the additional slots translate into measurable throughput recovery. At the same time, this should not be over-interpreted as a cross-node validation result, since the present study does not exercise network-induced delay, inter-node failures, checkpoint recovery across workers, or skewed partition placement.

Finally, the present evaluation should be interpreted as a systems characterization study rather than as an exhaustive benchmark against every alternative CEP platform. The current results include a static baseline, a same-host parallel comparison, rule-update latency measurements with clarified semantics, and both default-model and cross-model gold-set translation assessments, which together provide a substantially broader validation of the architecture's practical behavior.

7. Conclusions

PatternStudio shows that a neuro-symbolic approach can make CEP configuration more accessible without sacrificing deterministic execution. By combining constrained natural-language translation, schema validation, and a Flink-based runtime, the framework enables runtime rule orchestration without explicit job interruption while preserving a clear separation between the stochastic translation layer and the symbolic execution layer.

Within the current characterization study, the system achieved its best observed throughput at 250 active rules and maintained a bounded short-run memory footprint across all tested configurations. The revised analysis also clarifies that the severe P95/P99 latency inflation at 500 and 1000 rules is primarily a consequence of CPU saturation combined with alert amplification, not of abrupt memory instability. This finding is useful for capacity planning because it identifies where the current prototype stops behaving like an edge gateway solution and starts behaving like an overloaded single-node stress test.

The additional measurements strengthen this picture in three ways. First, a same-host parallel configuration with parallelism 4 improved throughput by about 20.7% at 500 rules, confirming that the orchestration model remains functional under partitioned execution on one machine. Second, the static 250-rule baseline reached 48,556.59 events/s, indicating an approximate 13.3% throughput overhead for dynamic orchestration in the matched follow-up comparison setup used for the static-versus-dynamic experiment. Third, the NL-to-IR evaluation showed both good default performance and backend portability: the default `gemini-2.5-flash` translator achieved 80.0% exact match and 100% structural match on five representative prompts, while the interchangeable OpenRouter-based backends exhibited markedly different profiles, with GPT-OSS 120B offering the strongest balance among the tested substitutes.

The paper's main practical contribution is therefore twofold: first, it demonstrates that natural-language-assisted CEP can be implemented in a controlled neuro-symbolic form rather than as unconstrained code generation; second, it characterizes the runtime envelope of that design on commodity hardware while also reporting direct evidence on

https://doi.org/10.3390/iot7020036

rule-update latency, static-baseline cost, multi-slot execution, translation quality, and model interchangeability at the semantic front end.

Future work will focus on multi-node scaling, direct measurement of rule-update latency and in-flight event behavior, gold-set evaluation of the NL-to-IR layer, and overload-management strategies for prioritizing high-criticality rules under extreme load.

8. Patents

The following patent is related to the work presented in this manuscript:

- J. Rosa Bilbao. "PatternStudio". Patent No. 2601164279875, granted on 20 January 2026.

Funding: This publication is part of the I+D+i grant PID2021-122215NB-C33 funded by MICIU/AEI/10.13039/501100011033 and by ERDF/EU.

Data Availability Statement: The source code supporting the findings of this study is openly available at https://github.com/JesusRosaB/PatternStudio accessed on 19 April 2026). No external dataset was used in the evaluation; the reported results were obtained from synthetic event streams generated through the benchmarking procedure described in the manuscript. Further inquiries can be directed to the corresponding author.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Cugola, G.; Margara, A. Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Comput. Surv.* **2012**, *44*, 1–62. [CrossRef]
2. Luckham, D.C. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*; Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 2002.
3. Atzori, L.; Iera, A.; Morabito, G. The Internet of Things: A survey. *Comput. Netw.* **2010**, *54*, 2787–2805. [CrossRef]
4. Gubbi, J.; Buyya, R.; Marusic, S.; Palaniswami, M. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Gener. Comput. Syst.* **2013**, *29*, 1645–1660. [CrossRef]
5. Borgia, E. The Internet of Things vision: Key features, applications and open issues. *Comput. Commun.* **2014**, *54*, 1–31. [CrossRef]
6. Boubeta-Puig, J.; Ortiz, G.; Medina-Bulo, I. Approaching the Internet of Things through Integrating SOA and Complex Event Processing. In *Advanced Solutions and Services for the Internet of Things*; IGI Global: Hershey, PA, USA, 2014; pp. 230–255. [CrossRef]
7. Bok, K.; Kim, D.; Yoo, J. Complex event processing for sensor stream data. *Sensors* **2018**, *18*, 3084. [CrossRef]
8. Gkoulis, D.; Bardaki, C.; Kousiouris, G.; Nikolaidou, M. Transforming IoT Events to Meaningful Business Events on the Edge: Implementation for Smart Farming Application. *Future Internet* **2023**, *15*, 135. [CrossRef]
9. Carbone, P.; Katsifodimos, A.; Ewen, S.; Markl, V.; Haridi, S.; Tzoumas, K. Apache Flink: Stream and Batch Processing in a Single Engine. *Bull. IEEE Comput. Soc. Tech. Comm. Data Eng.* **2015**, *36*, 28–38.
10. Suhothayan, S.; Gajasinghe, K.; Loku Narangoda, I.; Chaturanga, S.; Perera, S.; Nanayakkara, V. Siddhi: A Second Look at Complex Event Processing Architectures. In *Proceedings of the 2011 ACM Workshop on Gateway Computing Environments (GCE)*; ACM: New York, NY, USA, 2011; pp. 43–50. [CrossRef]
11. Bernhardt, T.; Vasseur, A. Esper: Complex Event Processing. 2016. Available online: http://www.espertech.com/esper (accessed on 15 January 2024).
12. Akidau, T.; Bradshaw, R.; Chambers, C.; Chernyak, S.; Davis, R.J.; Hwang, E.; Jayaweera, D.; Lan, A.; McCauley, M.; McVeety, S.; et al. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proc. VLDB Endow.* **2015**, *8*, 1792–1803. [CrossRef]
13. Fragkoulis, M.; Carbone, P.; Edlich, V.; Kaoudi, H. A Survey on the Evolution of Stream Processing Systems. *VLDB J.* **2024**, *33*, 507–541. [CrossRef]
14. Bezerra, S.D.; Filho, S.V.; Delicato, F.C.; da Rocha, A.R. Processing Complex Events in Fog-Based Internet of Things Systems for Smart Agriculture. *Sensors* **2021**, *21*, 7226. [CrossRef] [PubMed]
15. Rudin, C. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nat. Mach. Intell.* **2019**, *1*, 206–215. [CrossRef]
16. Garcez, A.d.; Lamb, L.C. Neurosymbolic AI: The 3rd Wave. *Artif. Intell. Rev.* **2023**, *56*, 12387–12406. [CrossRef]
17. OpenAI. GPT-4 Technical Report. *arXiv* **2023**, arXiv:2303.08774. [CrossRef]

18. Zhang, Y.; Li, Y.; Cui, L.; Cai, D.; Liu, L.; Fu, T.; Huang, X.; Zhao, E.; Zhang, Y.; Chen, Y.; et al. Siren's Song in the AI Ocean: A Survey on Hallucination in Large Language Models. *arXiv* **2023**, arXiv:2309.01219. [CrossRef]
19. Carbone, P.; Ewen, S.; Fóra, G.; Haridi, S.; Richter, S.; Tzoumas, K. State Management in Apache Flink: Consistent Stateful Distributed Stream Processing. *Proc. VLDB Endow.* **2017**, *10*, 1718–1729. [CrossRef]
20. Li, J.; Maier, D.; Tufte, K.; Papadimos, V.; Tucker, P.A. Semantics and Evaluation Techniques for Window Aggregation in Data Streams. *SIGMOD Rec.* **2005**, *34*, 5–16. [CrossRef]
21. Zhang, S.; Wu, Y.; Zhang, F.; He, B. Towards Concurrent Stateful Stream Processing on Multicore Processors. In Proceedings of the 2020 IEEE 36th International Conference on Data Engineering (ICDE), Dallas, TX, USA, 20–24 April 2020; pp. 1537–1548. [CrossRef]
22. Gkoulis, D.; Bardaki, C.; Nikolaidou, M.; Kousiouris, G.; Tsadimas, A. A Hybrid Simulation Platform for quality-aware evaluation of complex events in an IoT environment. *Simul. Model. Pract. Theory* **2024**, *135*, 102919. [CrossRef]
23. Bengio, Y. From System 1 Deep Learning to System 2 Deep Learning. In Proceedings of the Advances in Neural Information Processing Systems (NeurIPS), Vancouver, BC, Canada, 8–14 December 2019.
24. Gemini Team; Anil, R.; Borgeaud, S.; Yu, J.; Soricut, R.; Schalkwyk, J.; Dai, A.M.; Hauth, A.; Millican, K.; Silver, D.; et al. Gemini: A Family of Highly Capable Multimodal Models. *arXiv* **2023**, arXiv:2312.11805. [CrossRef]
25. Ji, Z.; Lee, N.; Frieske, R.; Yu, T.; Su, D.; Xu, Y.; Ishii, E.; Bang, Y.J.; Madotto, A.; Fung, P. Survey of Hallucination in Natural Language Generation. *ACM Comput. Surv.* **2023**, *55*, 1–38. [CrossRef]
26. Pan, J.Z. Constraints and Reasoning in Knowledge Graph. *IEEE Intell. Syst.* **2016**, *31*, 64–68. [CrossRef]
27. Harel, D. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.* **1987**, *8*, 231–274. [CrossRef]
28. Gökalp, M.O.; Koçyiğit, A.; Eren, P.E. A visual programming framework for distributed Internet of Things centric complex event processing. *Comput. Electr. Eng.* **2019**, *74*, 581–604. [CrossRef]
29. Boubeta-Puig, J.; Ortiz, G.; Medina-Bulo, I. MEdit4CEP: A Model-Driven Solution for Real-Time Decision Making in SOA 2.0. *Knowl.-Based Syst.* **2015**, *89*, 97–112. [CrossRef]
30. Corral-Plaza, A.; Boubeta-Puig, J.; Ortiz, G. MEdit4CEP-SP: A model-driven solution for user-friendly and inter-operable CEP and SP applications development. *J. Syst. Softw.* **2021**, *178*, 110974. [CrossRef]
31. Roldán-Gómez, J.; Boubeta-Puig, J.; Carrillo-Mondejar, J.; Castelo-Gómez, C.G.; Martínez-del Rincón, J. An automatic complex event processing rules generation system for the recognition of real-time IoT attack patterns. *Eng. Appl. Artif. Intell.* **2023**, *123*, 106344. [CrossRef]
32. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H.P.d.O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. Evaluating Large Language Models Trained on Code. *arXiv* **2021**, arXiv:2107.03374. [CrossRef]
33. Rozière, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X.E.; Adi, Y.; Liu, J.; Remez, T.; Rapin, J.; et al. Code Llama: Open Foundation Models for Code. *arXiv* **2023**, arXiv:2308.12950. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.