

RESEARCH

Open Access



Executing discrete/continuous declarative process specifications via complex event processing

Leo Poss^{1*}, Stefan Schöning¹ and Fabrizio Maria Maggi²

*Correspondence:

Leo Poss

leo.poss@ur.de

¹University of Regensburg,

Regensburg, Germany

²Free University of Bozen-Bolzano,
Bolzano, Italy

Abstract

Traditional Business Process Management (BPM) focuses on discrete events and fails to incorporate critical continuous sensor data in cyber-physical environments. Hybrid declarative specifications, using Signal Temporal Logic (STL), address this limitation by allowing constraints over both discrete events and real-valued signals. However, existing work has been limited to monitoring and post-hoc conformance checking. This paper introduces a novel execution architecture based on Complex Event Processing (CEP) that enables the real-time execution and enforcement of hybrid declarative models. Our three-layer approach integrates STL-inspired predicates into the execution flow, allowing the system to actively trigger activities and enforce process boundaries based on continuous sensor behavior. This approach bridges the gap between hybrid specification and operational control.

Keywords Declarative modeling language, Signal temporal logic, Complex event processing, Process enactment, Operational control

Introduction

Business Process Management (BPM) has traditionally focused on modeling, monitoring, and optimizing processes based on *discrete event data*, i.e., sequences of activities representing the execution of business processes (Dumas et al., 2018). Declarative approaches such as *Declare* (Pesic et al., 2007, Burattin et al., 2016) have emerged as a powerful alternative to imperative modeling languages, shifting the focus from explicitly describing control flow to specifying *constraints* that must hold during process execution. This paradigm is particularly valuable in dynamic and flexible environments, where not all execution paths can be anticipated in advance (Maggi, 2019). However, modern digital systems increasingly operate in *cyber-physical* or *data-rich* environments, where discrete events alone no longer capture the full state of a process. In domains such as *Industry 4.0* (Bazan and Estevez, 2022) or *Internet of Things (IoT)*-enabled healthcare (Janiesch et al., 2020), processes unfold alongside continuous sensor data (e.g., temperature, pressure, machine vibrations, or patient vital signs) that directly influence process decisions. Traditional process models, including classical *Declare* specifications, cannot

© The Author(s) 2026. **Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

directly reason about such continuous signals, as they were designed for discrete traces only. Consider a chemical reactor in a smart factory. A process engineer specifies a safety boundary: if the reactor temperature stays above 80 °C for more than 10 consecutive seconds, the activity *StartCooling* must be executed within 5 seconds to prevent thermal runaway. Standard declarative workflows cannot express or enforce this rule as they lack continuous-time semantics and cannot reason about durations in real-valued sensor streams. Classical data-aware process extensions fare no better: they operate on static data snapshots at task boundaries and cannot detect signal fluctuations between tasks.

Recent work has begun to address this limitation by using *Signal Temporal Logic (STL)* (Maler and Nickovic, 2004), a formalism from the cyber-physical systems domain that allows the specification of temporal properties over real-valued signals. STL enables constraints such as *temperature must never exceed 40 °C* or *if pressure rises above 20 bar, a safety valve must open within 10 seconds*. Building on STL, researchers have proposed *hybrid declarative models*¹ that unify discrete events and continuous data streams by encoding discrete events as boolean signals and combining them with continuous predicates (Corea et al., 2025). This enables *conformance checking* of hybrid traces (including both discrete and continuous information) to verify whether observed executions satisfy these hybrid specifications. While this represents a significant step forward, these approaches remain limited to *monitoring* and *post-hoc conformance checking* (Burattin et al., 2016, Donzé and Maler, 2010). Hybrid declarative models have proven effective for verifying logs and sensor data *after execution* or for runtime monitoring; however, no approach currently supports their *execution*, i.e., using these specifications to actively drive and control process instances in real time.

In this paper, we address this gap. Building on our previous work on executing declarative specifications via Complex Event Processing (CEP) (Poss and Schöning, 2026), we extend this approach to support *hybrid declarative models* that integrate continuous sensor data with discrete event streams. Our approach introduces a signal-processing layer that transforms continuous data into real-time predicates and integrates them into a CEP-based execution engine. This enables the specification and enforcement of STL-like constraints during runtime, for example, triggering activities when continuous conditions hold or proactively enforcing process boundaries based on sensor behavior. In summary, the main contributions of this work are:

- We extend the execution paradigm of declarative process models from purely discrete event data to *hybrid event–signal* scenarios.
- We present a *CEP-based execution architecture* capable of evaluating STL-inspired predicates and constraints in real time.
- We show how hybrid declarative constraints can be *enforced during execution*, enabling reactive and proactive behavior in data-rich environments.

This work bridges the gap between the specification of hybrid declarative constraints and their execution, enabling the development of next-generation process-aware systems that are both *context-sensitive* and *sensor-integrated*.

The remainder of this paper is structured as follows. The [Preliminaries and fundamentals](#) section covers declarative modeling, STL, and the principles of CEP as a stream

¹The term *hybrid* in this context has been introduced in (Cimatti et al., 2009).

reasoning paradigm. The [Hybrid declarative process execution](#) section introduces our core contribution: the three-layer hybrid execution architecture and an implementation-independent mapping of hybrid constraints to CEP logic. The [Implementation](#) section describes the working prototype built on the Esper engine. The [Evaluation](#) section assesses expressiveness and scalability. The [Related work](#) section discusses related work, while the [Conclusion](#) concludes with future research directions.

Preliminaries and fundamentals

This section provides the theoretical foundation for our work. We first discuss the fundamentals of temporal logic and declarative process modeling based on LTL. We then present STL and explain its role in enabling hybrid declarative specifications that integrate discrete events with continuous data. Finally, we introduce the principles of Complex Event Processing (CEP) and the Esper engine, which serve as the operational runtime infrastructure for our hybrid execution paradigm.

Linear temporal logic and declarative process modeling

Linear Temporal Logic (LTL) (Pnueli, 1977) is a classical formalism for specifying temporal properties of discrete-event systems. LTL formulae are evaluated over *linear traces*, i.e., sequences of states ordered in time, and are used to specify constraints concerning the ordering and occurrence of events. Given a set of atomic propositions AP , the syntax of LTL is defined as:

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid X\varphi \mid F\varphi \mid G\varphi \mid \varphi_1 U \varphi_2$$

where $p \in AP$, X denotes “next”, F “eventually”, G “globally”, and U “until”. For example, the formula $G(a \rightarrow Fb)$ expresses that “whenever a occurs, b must eventually occur”. Such expressions have been widely used for verifying temporal properties and form the logical basis for many declarative approaches in business process management (Maggi, 2019).

Declarative process modeling builds on these principles and shifts the perspective from explicitly modeling control flow to defining a set of *constraints* that must hold during process execution. One of the most widely adopted declarative languages is *Declare* (Pesic et al., 2007), whose templates are typically defined as LTL (more specifically LTL over finite traces, LTL_f) formulae. Each Declare constraint captures a behavioral rule that restricts the space of acceptable traces. Examples of commonly used templates include:

- **RESPONSE**(a, b): $G(a \rightarrow Fb)$, stating that whenever a occurs, b must eventually occur.
- **NOTEXISTENCE**(a): $G(\neg a)$, specifying that a must never occur.
- **PRECEDENCE**(a, b): $(\neg b U a) \vee G(\neg b)$, meaning that b may occur only after a has occurred.

This modeling style offers a high degree of flexibility and is particularly suited to dynamic, knowledge-intensive processes (Maggi, 2019). It has been extensively used for runtime monitoring, conformance checking, and process discovery (Maggi, 2019). Beyond pure control-flow constraints, Declare has been extended to *multi-perspective* (MP) specifications through Metric First-Order Temporal Logic (MFOTL) (Burattin et al., 2016). MFOTL enriches LTL with first-order quantification over data variables (e.g.,

event payloads) and metric time intervals, enabling constraints such as `Response (A, B) [A.orderId=B.orderId]` that correlate activation and target events based on their data attributes. Our architecture builds on this MP-Declare foundation, further extending it with continuous signal predicates drawn from STL.

Nevertheless, LTL and Declare, even in their multi-perspective form, share a fundamental limitation: they operate over discrete, symbolic event logs and cannot naturally express conditions involving real-valued data or continuous-time behavior. In many cyber-physical or IoT-enabled scenarios, such as smart manufacturing or healthcare, process decisions depend on continuous sensor streams (e.g., temperature, pressure, oxygen saturation). Capturing such dependencies requires a more expressive temporal logic capable of reasoning over real-valued signals.

Signal temporal logic and hybrid declarative specifications

STL (Maler and Nickovic, 2004) extends temporal reasoning beyond discrete events to encompass real-valued, time-continuous signals. STL formulae are evaluated over signals $x : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}$, enabling the specification of temporal constraints involving thresholds and timing intervals. The syntax of STL is defined as:

$$\varphi ::= \top \mid x(t) \text{ op } c \mid \neg \varphi \mid \varphi_1 \wedge \varphi_2 \mid F_{[a,b]} \varphi \mid G_{[a,b]} \varphi \mid \varphi_1 U_{[a,b]} \varphi_2,$$

where $\text{op} \in \{<, \leq, =, \geq, >\}$ and $[a, b]$ denotes a time interval. STL also supports *robust semantics* (Donzé and Maler, 2010), which quantify how strongly a signal satisfies or violates a property, thereby enabling tolerance to noise and uncertainty.

To reason simultaneously about discrete events and continuous data, we adopt the hybrid model construction introduced by Corea et al. (Corea et al., 2025). We define the set of state variables as a pair $V = (D, C)$, where D is a set of discrete variables derived from event logs and C is a set of continuous variables derived from data streams (e.g., IoT sensor measurements). From a standard event log, we extract a *timed trace*, $tr = \langle (d_1, t_1), \dots, (d_n, t_n) \rangle$, where (d_i, t_i) denotes the occurrence of event $d_i \in D$ at time t_i .

Since discrete events lack a continuous representation, we can transform each $d \in D$ into an *augmented boolean signal* $S_d : T \rightarrow \{0, 1\}$, set to 1 whenever d is the most recent observed event, and to 0 otherwise. For notational convenience, we write $d[t] = S_d(t)$ to denote its value at time t . Continuous variables $c \in C$ are taken directly from their associated data streams, each producing values over the same continuous time domain $T = [0, m] \subset \mathbb{R}$. These components are then combined into a *hybrid continuous trace* $w : T \rightarrow \mathbb{R}^{|D|+|C|}$, assigning to each variable (discrete or continuous) its value at each time instant.

Based on this representation, the STL fragment we use in this work is defined by the grammar:

$$\varphi ::= \mathbf{true} \mid c[t] \text{ op } k \mid \text{dis}(d[t]) \mid (\neg \varphi) \mid (\varphi_1 \wedge \varphi_2) \mid (\varphi_1 U_I \varphi_2),$$

where $c \in C$, $d \in D$, $\text{op} \in \{<, \leq, =, \geq, >\}$, $k \in \mathbb{Q}$, and I is a time interval in T . The term $c[t] \text{ op } k$ expresses constraints over continuous variables (e.g., “the temperature must be larger than 100”), while $\text{dis}(d[t])$ refers to the boolean signal associated with discrete event d at time t .

Hybrid constraint templates

Standard Declare templates adapt naturally to hybrid constraints using STL semantics:

1. Hybrid Response (Φ_2)

Requires that whenever condition Θ_1 holds, condition Θ_2 must follow within 5 seconds:

$$G_{[0,m]}(\Theta_1 \rightarrow F_{[0,5]}\Theta_2)$$

Example: “Whenever the temperature exceeds 80°C, the cooling activity must start within 5 seconds”, where:

$$\Theta_1 := \text{temp}(t) > 80 \quad \text{and} \quad \Theta_2 := \text{dis}(\text{StartCooling}[t])$$

2. Hybrid NotExistence (Φ_1)

Specifies that a continuous condition Θ must never hold continuously for longer than duration d :

$$\neg F_{[0,m]}(G_{[0,d]}\Theta)$$

Example: “Temperature must never exceed 90°C for more than 10 seconds”, where:

$$\Theta := \text{temp}(t) > 90 \quad \text{and} \quad d = 10 \text{ s}$$

Consider a continuous trace of length $m = 25 \text{ s}$ with one continuous sensor variable $C = \{\text{temp}\}$ and two discrete activity signals $D = \{\text{StartCooling}, \text{Restart}\}$ ²:

- ▷ $t \in [0, 10) \text{ s}$: $\text{temp}(t) = 75^\circ \text{C}$; no activity performed.
- ▷ $t = 10 \text{ s}$: Temperature rises to 85°C and holds steady.
- ▷ $t = 12 \text{ s}$: Temperature spikes to 93°C and remains high.
- ▷ $t = 14 \text{ s}$: The `StartCooling` activity finishes executing.
- ▷ $t = 18 \text{ s}$: Temperature drops to 78°C and stabilizes.

Mathematically, we represent the trace as $\omega(t) = [\text{temp}(t), S_{\text{StartCooling}}(t)]$, where $S_{\text{StartCooling}}(t) = 1$ for $t \geq 14.0 \text{ s}$ and 0 otherwise. Note that the temperature remains above 90°C continuously from $t = 12 \text{ s}$ to $t = 18 \text{ s}$. Evaluating this trace against the two hybrid specifications yields:

- 1 **Hybrid Response (Φ_2)—Satisfied:** The condition $\text{temp}(t) > 80$ triggers at $t = 10.0 \text{ s}$, opening the 5-second response window $[10.0, 15.0] \text{ s}$. Because $S_{\text{StartCooling}}(t)$ becomes 1 at $t = 14.0 \text{ s}$ (inside the active window), the trace satisfies Φ_2 .
- 2 **Hybrid NotExistence (Φ_1)—Satisfied:** The constraint $\neg F_{[0,25]}(G_{[0,10]}(\text{temp}(t) > 90))$ monitors high temperatures. Although the temperature exceeds 90°C at $t = 12 \text{ s}$, it falls back down at $t = 18 \text{ s}$, giving a continuous high-heat duration of 6 seconds. Because this stays under the 10-second threshold d , Φ_1 is satisfied.

Hybrid declarative specifications combine discrete events and continuous signals in a single constraint model—a capability beyond traditional Declare. Existing work (Corea et al., 2025) applies these specifications to offline monitoring and conformance checking, but stops at post hoc verification. Our architecture extends them to runtime execution.

²The Restart task is not executed in this trace, but appears in later scenarios.

Complex event processing

Complex Event Processing (CEP) is a design paradigm engineered for processing high-velocity, continuous streams of real-time event data to identify actionable patterns, causal relationships, and temporal trends across multiple data sources (Jani-esch et al., 2020, Luckham, 2008, Wu et al., 2006). In contrast to traditional Database Management Systems (DBMS) that execute momentary queries over static, persistent data tables, a CEP engine maintains static, continuous queries over dynamic data streams.

Formally, an event stream can be defined as $S = \langle e_1, e_2, e_3, \dots \rangle$, an infinite sequence of event instances. Each individual event $e_i = (a, \vec{v}, t)$ is a tuple containing an event type identifier a , a vector of multi-perspective payload data attributes $\vec{v}$, and a discrete time-stamp $t \in \mathbb{R}_{\geq 0}$ denoting its time of occurrence. A *complex event* is a higher-level, semantically enriched event derived by the CEP engine when a set of primitive stream elements satisfies specific structural, relational, or temporal conditions. This pattern-matching mechanism typically uses sliding time windows (W) that bound the evaluation domain to a finite historical interval Δt .

While the conceptual foundations of stream reasoning remain implementation-independent, operationalizing these paradigms requires a concrete execution engine. For this work, the *Esper* CEP framework³ was selected as the underlying infrastructure. Esper provides several distinct architectural advantages suited to declarative process enforcement: First, Esper's Event Processing Language (EPL) (Langhi et al., 2025) extends standard SQL semantics with native regular expressions for event streams (via the `PATTERN` operator) and sequential causality matching using the linear-time operator (`->`). This maps directly to the forward- and backward-looking temporal structures of declarative templates. Second, it provides out-of-the-box support for programmatic time windows (`timer:within`, `timer:interval`), allowing the system to track real-time clock expirations and automatically emit complex negative events (such as timeouts). And finally, as Esper is built as an embedded Java library, it avoids the heavy network serialization overhead found in distributed stream frameworks (such as Apache Flink⁴ or Kafka Streams⁵).⁶

Hybrid declarative process execution

Corea et al. (2025) introduced a framework to specify hybrid declarative constraints in STL, but restricted its use to offline monitoring and post hoc conformance checking over historical logs. We extend this foundation to real-time execution and operational control using CEP. In our architecture, hybrid Declare specifications drive an execution engine that shapes process runtime in two ways: (i) the engine evaluates the current trace state to enable, disable, or block human tasks in the user interface; and (ii)

³<https://www.espertech.com/esper/>

⁴<https://flink.apache.org/>

⁵<https://kafka.apache.org/43/streams/>

⁶For additional context on the theoretical distinctions between Stream Processing and Complex Event Processing, see (Cugola and Margara, 2012). Their core taxonomy outlines the differences between data-flow models (such as Flink and Storm) and pattern-detection models (such as Esper and Drools Fusion). For a survey covering distributed CEP architectures and scalability trade-offs, see (Dayarathna and Perera, 2019). On the expressiveness of event pattern languages (Artikis et al., 2017), contrasts declarative, logic-based approaches with imperative, automaton-based ones. This distinction is directly relevant to Esper's hybrid EPL design discussed above.

when an STL temporal window expires, the engine can trigger automated task execution (such as opening a cooling valve). To support this execution model, our architecture processes event streams through a three-layer hierarchy, turning raw continuous signals into higher-level process states:

- 1 **Atomic Layer (L1)**: Detects primitive activation and target events from raw sensor streams and discrete logs, sending structural tokens to higher layers.
- 2 **Constraint Layer (L2)**: Handles pattern matching and tracks lifecycle transitions (activations, fulfillments, and violations).
- 3 **Process Layer (L3)**: Aggregates constraint states into a unified case view to compute task enablement and drive automated actions.

Layered architecture

The execution architecture is organized as a three-layer runtime, where each layer performs a distinct transformation, progressively abstracting raw real-time streams into an actionable, stateful process view. The complete system pipeline is summarized in Fig. 1.

Atomic Event Layer (L1)

The bottom layer, referred to as the *Atomic Event Detection Layer (L1)*, is responsible for normalizing input streams and transforming them into semantically meaningful data tokens. The underlying stream infrastructure handles two primary input types: (i) discrete *atomic events*, representing the completion of business activities, and (ii) continuous *signal samples*, representing time-stamped telemetry measurements. Each atomic event is described as a tuple $e = \langle caseId, activity, ts \rangle$, representing activity occurrences such as `PickItem` or `Ship`, with each active task tracked as a boolean signal $d[t]$. Continuous input is represented as $s = \langle caseId, sensorId, value, timestamp \rangle$, and evaluated against continuous predicates of the form $c[t] \text{ op } k$ in real time.

In contrast to post-hoc monitoring approaches which convert historical logs into augmented boolean signals after execution, this runtime engine performs hybridization continuously. Relational filtering predicates evaluate incoming metrics against static

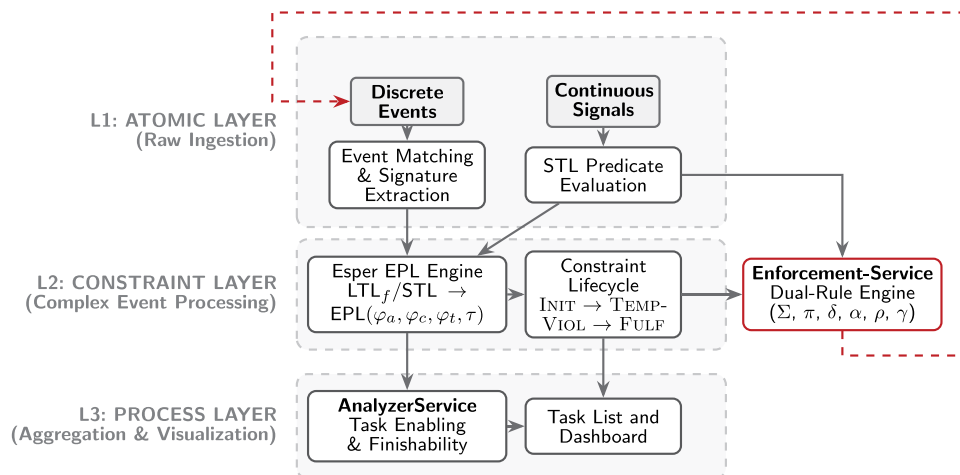


Fig. 1 DeclareCEP system architecture: raw task events and continuous signals are ingested at L1, processed through EPL-encoded MP-Declare templates at L2 with duration enforcement, and exposed to state aggregation and UI monitoring at L3

signal boundaries (e.g., comparing a telemetry sample value against a defined threshold) to broadcast binary state-change tokens. These structural tokens are explicitly categorized as **ACTIVATION** or **TARGET** signals and emitted into a consolidated intermediate stream along with their parent constraint identifier. This ensures that the engine is not forced to store massive raw signal traces in memory.

Constraint Status Layer (L2)

The second layer of the runtime, the *Constraint-Level Pattern Matching Layer (L2)*, is a stateless stream processor that detects temporal patterns defining a constraint's logic. It consumes the stream of standardized activation and target tokens from **L1** and maps sequential lifecycle shifts through pattern queries. For each constraint template, this layer deploys stream queries that look for specified sequential and metric dependencies between **ACTIVATION** and **TARGET** elements over sliding operational windows (Δt). For example, a **RESPONSE** constraint's fulfillment is detected by a pattern that identifies a valid target token following an activation within a given time window. Dynamic correlation constraints (e.g., matching a response parameter to an activation via cross-event data fields) are verified inline by mapping relational attributes across discrete event schemas. When a pattern is successfully matched or its deadline expires, **L2** emits a derived lifecycle milestone event, such as:

- **ConstraintFulfillment**: $\langle caseId, constraintId, activationId, ts \rangle$.
- **ConstraintViolation**: $\langle caseId, constraintId, activationId, ts, reason \rangle$.

This layer remains entirely stateless regarding process state; it does not track overall case history but rather announces critical transactional events that define individual constraint statuses.

Process Status Layer (L3)

The *Process Status Layer (L3)* builds on the lifecycle milestones emitted by the pattern-matching layer and provides a stateful view of execution for each unique process case. Its primary role is twofold: (i) to manage and aggregate the status of all individual constraints into an overall case-level picture, and (ii) to trigger process-level actions whenever significant state changes occur. This layer consists of stateful listeners that subscribe to the fulfillment and violation event streams from **L2**. For example, a constraint instance begins in an **Idle** state. Upon receiving an **ACTIVATION** token forwarded from **L1**, the tracking routine marks the constraint as **Activated**. When it subsequently receives a structural fulfillment token from **L2**, it transitions the state to **Fulfilled**.

Beyond tracking, this evaluation routine interprets lifecycle milestones to maintain active process state vectors. These state transitions dynamically compute allowed subsequent operations (**Task Enablement**) and trace-wide validity (**Instance Finishability**), rendering the current permissible actions transparently visible to actors or external applications.

An *Enforcement Service* runs alongside **L3** to monitor sustained signal conditions. It relies on a formal 6-tuple $\mathcal{R} = (\Sigma, \pi, \delta, \alpha, \rho, \gamma)$, whose parameters are detailed in Table 1. By separating the signal stream (Σ), metric predicate (π), and duration threshold (δ) from the triggered action (α), payload (ρ), and reset predicate (γ), this tuple allows **L3** to actuate physical equipment or trigger safety alerts. When a signal violation persists for

Table 1 Formal components of the enforcement service tuple

Element	Domain	Operational Definition
Σ	Stream Identifier	The target high-frequency signal stream source emitted by an external sensor or physical system (e.g., <i>temperature</i>).
π	Predicate Logic	The conditional expression evaluated over payload metrics (e.g., <i>value > 80.0</i>).
δ	Temporal Window	The minimum continuous duration threshold required to validate a breach.
α	Action Event	The discrete synthetic event injected into the stream upon violation to trigger machine actuation or alert human operators.
ρ	Response Payload	The key-value payload attached to the injected action event, allowing parameterized actuation (e.g., setting a target temperature or routing an alert to a specific operator).
γ	Reset Predicate	The condition (e.g., $\neg\pi$ sustained, or a separate recovery threshold) under which an active breach is cleared, determining whether α fires once per breach episode or repeatedly while π holds.

duration δ , **L3** can call external actuation services to progress the process instance, keeping business logic clear of the lower-level CEP query definitions.

Finally, **L3** is responsible for initiating automated or corrective actions; when a state transition to Fulfilled or Violated occurs, listeners can directly trigger external services, notify target systems, or launch compensating activities, effectively closing the semantic loop between continuous physical events and business execution.

By establishing these systematic mappings, the execution pipeline shifts from an ad hoc tracking routine into a deterministic compilation framework. The operationalization of these abstract components within individual runtime layers is explained below for two exemplary templates:

Hybrid Response constraint The HYBRID RESPONSE template requires that whenever an activation condition Θ_1 holds, a response condition Θ_2 must occur within a bounded interval, where I_a and I_r denote the activation and response windows. Its mapping proceeds as follows. At **L1**, two queries detect Θ_1 and Θ_2 , emitting ACTIVATION and TARGET tokens. At **L2**, a pattern checks whether a TARGET follows an ACTIVATION within I_r , producing a fulfillment event; a complementary pattern with a timer emits a violation token when no such match occurs. At **L3**, a listener updates the constraint state: it marks an ACTIVATION as Activated and transitions the instance to Fulfilled or Violated upon receiving the corresponding L2 event.

Hybrid NotExistence constraint The HYBRID NOTEXISTENCE template demands that a condition Θ never occur within a time interval I . Its mapping is simpler. At **L1**, a query detects whether Θ ever becomes true and emits an event if so. At **L2**, this is converted into a constraint violation event. At **L3**, the listener sets the constraint state to Violated. If no such event is received before the end of I , the constraint is considered fulfilled, and the listener updates the structural execution state accordingly.

Conceptual hybrid constraint mapping to CEP logic

To execute hybrid specifications over real-time event streams, formal declarative templates are translated into continuous query rules. This compiler pipeline maps Signal Temporal Logic (STL) formulae into intermediate Denotational Complex Event Logic (CEL) expressions (Grez et al., 2019), which are then compiled into executable Esper

EPL queries (Langhi et al., 2025). Below, we demonstrate this translation for two core templates:

- 1 A HYBRID RESPONSE constraint requires that whenever activation condition Θ_1 holds, target condition Θ_2 must follow within reaction window I_r .

- **Declare / LTL:** $G(a \rightarrow Fb)$
- **Hybrid STL Semantics:** $G_{I_a}(\Theta_1 \rightarrow F_{I_r}\Theta_2)$
- **CEL Semantics:** $((\Theta_1 AS x) \cdot \text{True}^* \cdot (\Theta_2 AS y)) \text{FILTER}(x.ts \in I_a \wedge y.ts - x.ts \in I_r)$
- **Esper EPL Execution Rule (L1/L2):**

```
INSERT INTO constraintStatus
SELECT b.id, 'ResponseConstraint' as name, 'FULFILLMENT' as
  type, b.timestamp
FROM pattern [every a=constraintStatus(type='ACTIVATION') → b
  =constraintStatus(type='TARGET')]
WHERE b.timestamp - a.timestamp <= I_r;
```

- 2 A HYBRID NOTEXISTENCE constraint enforces a safety rule forbidding condition Θ across time window I .

- **Declare / LTL:** $G(\neg a)$
- **Hybrid STL Semantics:** $\neg F_I(\Theta)$
- **CEL Semantics:** $\neg(\Theta AS x) \text{FILTER}(x.ts \in I)$
- **Esper EPL Execution Rule (L1/L2):**

```
INSERT INTO constraintStatus
SELECT id, 'NotExistenceConstraint' as name, '
  PERMANENT_VIOLATION' as type, timestamp
FROM GenericEvent
WHERE  $\Theta$  AND timer:within(I sec);
```

Multi-perspective component decomposition

To handle multi-perspective (MP) declarative constraints beyond standard control flow, we break each rule into four parts: *Activation*, *Target*, *Correlation*, and *Temporal conditions* (Burattin et al., 2016). Our architecture processes these components by separating payload filtering from temporal pattern matching:

- **Activation Condition** ($\varphi_a(x)$): Evaluates conditions on initiating events ($p_A(x) \wedge r_a(x)$ in MFOTL). The Atomic Layer (L1) handles this step: continuous queries evaluate incoming telemetry or event payloads against limits (e.g., `eventType='Temp'` and `value>80.0`) and send an ACTIVATION token to L2.
- **Target Condition** ($\varphi_t(y)$): Evaluates target events ($p_B(y) \wedge r_t(y)$). Also running at L1, the engine filters target occurrences using local attributes (e.g., `eventType='StartCooling'` and `mode='emergency'`) and passes a tagged TARGET marker to L2.
- **Correlation Condition** ($\varphi_c(x,y)$): Defines data relationships $r_c(x,y)$ between activation and target events. Evaluated at the Constraint Layer (L2), Esper's pattern engine runs relational joins across event schemas when ACTIVATION and TARGET

tokens arrive (e.g., matching `cast(b.payload('caseId'), String)` to `cast(a.payload('caseId'), String)`).

- **Time Condition (φ_τ):** Enforces time bounds ($y.ts - x.ts \in I$) between activation and target events.⁷ At **L2**, clock-driven execution windows track these limits (e.g., expecting a response within `timer:within(5 sec)`) and emit success or timeout events.

The appendix provides a complete component matrix mapping these elements from formal MFOTL semantics to Esper EPL implementations, along with mappings for all remaining Declare templates.

Expressiveness and structural coverage

To evaluate the expressiveness of this execution model, we examine how constraints map along two dimensions: Constraint Structure (predicate arity and payload complexity) and Temporal Directionality (forward versus backward time dependencies). Comparing these dimensions against the tables provided in the appendix shows that the pipeline supports core declarative template families without altering their formal semantics:

- 1 **Atemporal Unary Structures:** Templates like **EXISTENCE** and **NOTEXISTENCE**. The engine runs a sliding filter (Θ) at **L1**. Execution checks for the presence or absence of a matching token within interval I , so **L2** does not need to store sequence state.
- 2 **Forward-Looking Binary Structures:** Causal templates such as **RESPONSE** and **CHAIN RESPONSE**. **L1** captures activation (Θ_1) and target (Θ_2) conditions as separate stream entries. **L2** evaluates forward time intervals using sequential matching ($\rightarrow$) and time windows (`timer:within(I)`). If a target condition arrives within the window, the constraint is fulfilled; otherwise, a timeout sub-query emits a violation token.
- 3 **Backward-Looking Historical Structures:** Represented by the **PRECEDENCE** template. To evaluate historical dependencies over live streams, the engine converts forward sequences into backward lookups. When the target event Θ_2 arrives, the engine checks its timestamp against cached records. If an activation event (Θ_1) occurred within the lookback window I_a , execution succeeds; otherwise, it is blocked.
- 4 **Multi-Perspective Data-Correlated Structures:** To process data dependencies alongside control flow, the engine splits multi-perspective templates into three parts: activation filter $\varphi_a(x)$, target filter $\varphi_t(y)$, and correlation constraint $\varphi_c(x, y)$. **L1** filters streams using unary properties, while **L2** applies joins to match attributes across event schemas (e.g., matching a `caseId`).

Operational execution of the running example

To show how the three-layer CEP architecture executes hybrid declarative specifications in real time, we run the cooling process model introduced in Section “[Preliminaries and fundamentals](#)”. The engine processes raw sensor readings ($\text{temp}(t)$) and activity tokens live to enforce three constraints (Φ_1, Φ_2, Φ_3) instead of the offline trace evaluation of $\omega(t)$ in earlier approaches.

⁷Denotational CEL and MFOTL express temporal bounds using timestamp comparisons ($y.ts - x.ts \in I$), whereas the Esper EPL engine uses live clock constructs like `timer:within` or `timer:interval` to emit timeout events when an interval ends.

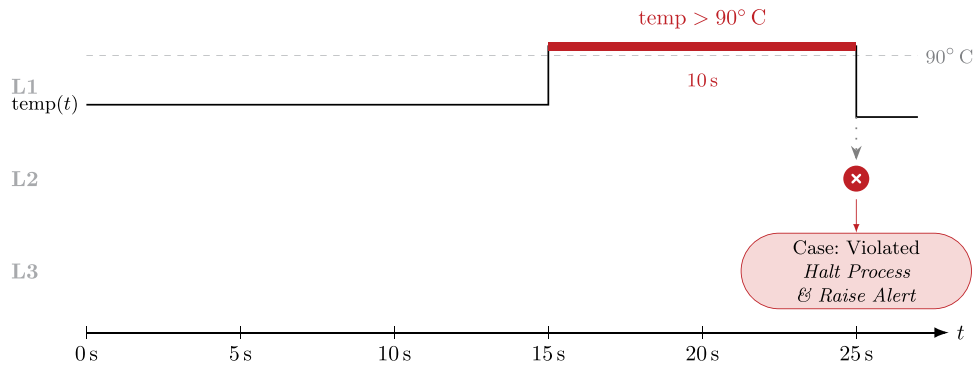


Fig. 2 Example NotExistence constraint execution via CEP streams

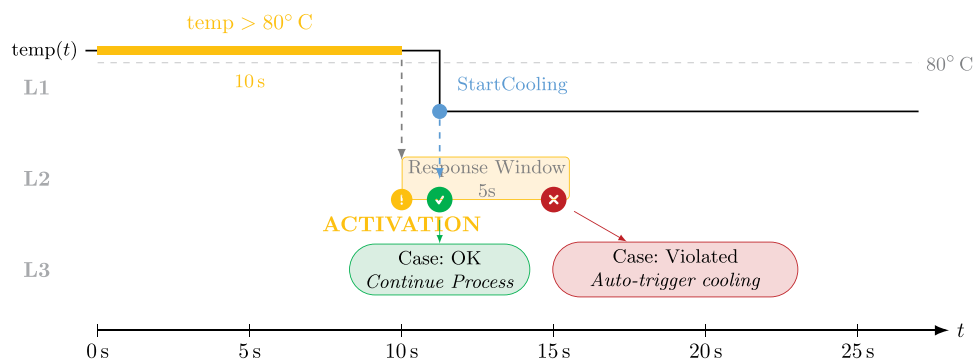


Fig. 3 Example Response constraint execution via CEP streams

Hybrid NotExistence (Φ_1)

Recall the NOTEXISTENCE safety constraint $\Phi_1 = \text{NOTEXISTENCE}_{[0,25]}(G_{[0,10]}(temp(t) > 90))$, which dictates that the temperature must never remain above 90°C for a continuous duration of 10 seconds or more. As illustrated in Fig. 2:

- **Layer 1 (L1):** The continuous predicate evaluator continuously monitors incoming telemetry stream samples. When $temp(t) > 90$ holds continuously for 10 seconds, L1 emits a TARGET event into the internal `constraintStatus` stream.
- **Layer 2 (L2):** The pattern matching captures the TARGET event and evaluates the constraint violation, immediately generating a `PERMANENT_VIOLATION` lifecycle event.
- **Layer 3 (L3):** The process status manager intercepts the violation event, transitions the constraint state to Violated, and triggers an immediate system halt and safety alert through an automated listener callback.

Hybrid Response (Φ_2)

Consider the HYBRID RESPONSE constraint $\Phi_2 = G_{[0,25]}(temp(t) > 80 \rightarrow F_{[0,5]}dis(\text{StartCooling}[t]))$, which requires the activity `StartCooling` to execute within 5 seconds whenever the temperature stays above 80°C for 10 consecutive seconds. As shown in Fig. 3:

- **Layer 1 (L1):** When $temp(t) > 80$ persists for 10 seconds, L1 emits an ACTIVATION event. Concurrently, discrete completion events for `StartCooling` are ingested as TARGET events.

- **Layer 2 (L2):** An active Esper pattern tracks whether a valid TARGET event succeeds the ACTIVATION within the 5-second response window ($\Delta t \leq 5s$). If matched within $t \in [10, 15]s$, L2 emits a FULFILLMENT event; otherwise, a timer expiration emits a VIOLATION.
- **Layer 3 (L3):** Upon activation, if the system is configured for automated execution, L3 invokes the TaskExecutorService to automatically launch StartCooling, ensuring fulfillment before the 5-second deadline elapses.

Hybrid PRECEDENCE (Φ_3)

Finally, consider the PRECEDENCE precondition constraint Φ_3 , which restricts the sensitive activity Restart from executing unless the machine has maintained a continuous cool state ($\text{temp}(t) < 50$) for at least 20 seconds. As depicted in Fig. 4:

- **Layer 1 (L1):** L1 tracks the duration of the condition $\text{temp}(t) < 50$ and emits a state event once the 20-second continuous stability window is satisfied.
- **Layer 2 (L2):** When an operator attempts to trigger Restart, L2 validates the attempt against the historical window. An attempt made at $t = 24s$ (before the 20-second countdown completes; remember: the temperature starts dropping at $t = 18s$) fails pattern matching and emits a BLOCKED state event.
- **Layer 3 (L3):** The stateful AnalyzerService uses the current L2 state to dynamically manage the user task list. The Restart task remains disabled in the interface until the 20-second cool-down is verified, actively preventing unsafe human execution.

By coordinating these three layers, the runtime transforms passive declarative formulae into a real-time execution engine capable of safety monitoring (Φ_1), automated reaction (Φ_2), and dynamic operational gatekeeping (Φ_3).

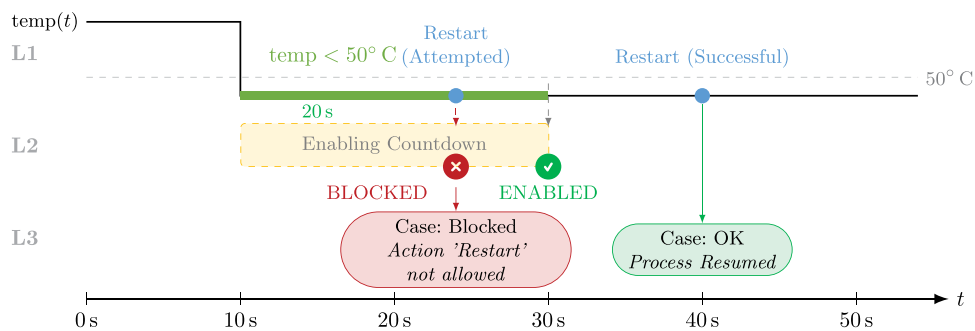


Fig. 4 Example Precedence constraint execution via CEP streams

Listing 1 Detection of ACTIVATION and TARGET events of constraints (L1).

```

--- NOT EXISTENCE TARGET for temperature > 90 without a parameter
timer
INSERT INTO constraintStatus
SELECT id, '1' as name, 'TARGET' as type, timestamp as timestamp,
       cast(payload('temp'), double) as temp
FROM GenericEvent
WHERE eventType = 'Temp' AND cast(payload('temp'), double) > 90

--- RESPONSE ACTIVATION where temp > 80 for 10 seconds
INSERT INTO constraintStatus
SELECT t1.id AS id, '2' AS name, 'ACTIVATION' AS type, t1.
       timestamp AS timestamp, cast(t1.payload('temp'), double) AS
       temp
FROM pattern [every t1 = GenericEvent(eventType = 'Temp', cast(
payload('temp'), double) > 80) → (timer:interval(10 sec) and
not GenericEvent(eventType = 'Temp', cast(payload('temp'),
double) <= 80)]]

```

Listing 2 Query for detecting fulfillment patterns (L2).

```

SELECT b.id, b.name, b.type, b.timestamp as timestamp
FROM PATTERN [every a=constraintStatus(type='ACTIVATION', name='2
') → b=constraintStatus(type='TARGET', name='2')]

```

Implementation

We developed a prototype that enforces hybrid declarative constraints via CEP rules.⁸ In contrast to existing approaches, where constraint conditions are purely discrete (e.g., the occurrence of an activity), hybrid constraints may involve continuous predicates evaluated over time windows. Our architecture is implemented as a layered, event-driven runtime built on top of the Esper CEP engine. Esper provides a high-performance environment for evaluating continuous queries over streaming data using its Event Processing Language (EPL), which is well-suited for operationalizing declarative process semantics.

L1: atomic event detection

This layer ingests and normalizes runtime data into typed event streams. Both discrete atomic events and continuous signals are represented using the `GenericEvent` class, which contains an `eventType` and a payload map specifying the type (Signal/Task) and either the measured value (e.g., `temp=100`) or task-related metadata (e.g., `userId=3`). The final step in L1 transforms these raw events into semantic atomic events in a unified `constraintStatus` stream, which serves as the input for L2 and contains all activations and targets relevant for constraint monitoring, as shown in Listing 1. The `constraintStatus` stream is instantiated upon the first insertion and stores the event ID, the constraint name, the event type (ACTIVATION or TARGET), a timestamp, and the payload used for correlation at L2.

Listing 3 Query for detecting violation patterns (L2).

```

INSERT INTO constraintStatus
SELECT '2' as name, 'PERMANENT_VIOLATION' as type, a.timestamp as
       timestamp
FROM PATTERN [
  a=constraintStatus(type='ACTIVATION', name='2')
  → (timer:interval(5 sec) and not b=constraintStatus(type='
       TARGET', name='2'))
]

```

⁸See <https://github.com/LeoPoss/quarkusCEP/tree/STL> for the code repository and a short guided introduction at <https://youtu.be/7FwQxgyozcM>

L2: constraint-level pattern matching

The second layer consumes the `constraintStatus` stream and uses EPL pattern statements to detect fulfillments and violations, emitting standardized lifecycle events. A fulfillment is implemented as a pattern query over the `constraintStatus` stream (see Listing 2), while violations are detected using negative patterns with timeouts (see Listing 3, where a `RESPONSE` must occur within 5 seconds of its activation).

L3: process-level state management

The output of **L2** forms the `constraintStatus` layer at the process level, consisting of lifecycle events that represent the current state of every constraint instance, thereby defining the system's overall state in relation to the frontend. We use Esper's listener mechanisms to consume these events and update the internal state accordingly. Additional logic can be embedded to respond to state changes, for example, triggering a compensation service whenever any constraint becomes permanently violated. The corresponding subscriber implementation invokes an external service directly from this layer, thereby closing the loop between declarative semantics and operational execution.

Integration and execution model

Declarative execution differs fundamentally from traditional imperative process enactment (e.g., token-passing in BPMN). Imperative engines dictate the sequence of operations, advancing tokens along explicit control-flow paths and executing service tasks directly. A declarative engine does not prescribe a specific execution path. Instead, it acts as a dynamic state-space constraint filter. The primary role of our CEP-driven runtime is to evaluate ongoing event-signal configurations to compute two real-time state vectors: (i) *Task Enablement*, which defines the exact subset of activities that an operator is permitted to perform, and (ii) *Instance Finishability*, which determines whether the cumulative process trace satisfies all mandatory constraints, allowing the instance to terminate safely. Our framework bridges this gap through restrictive gatekeeping and active operational automation, leveraging **L3** listeners. When an STL threshold activation or a discrete task transition occurs, the process layer intercepts the lifecycle state change and translates it into an event-driven trigger. This allows the system to invoke external APIs or inject automated activity updates, effectively matching the functional execution capability of imperative service tasks while preserving the structural flexibility of a declarative model.

The three layers operate as a continuous, reactive dataflow pipeline: events are ingested and normalized in **L1**, complex patterns are detected in **L2**, and lifecycle events are interpreted to manage process state and trigger actions in **L3**. All logic is expressed as continuous EPL queries and event-driven callbacks, enabling low-latency monitoring and control. For actual process execution, we also track the current tasks and signals to provide users with feedback on which tasks are possible or restricted. This is done in the spirit of (Ackermann et al., 2018), but fully separated from the CEP logic: decisions are based exclusively on the current constraints and the observed trace. We distinguish two checks: (i) whether the process instance can be completed, i.e., none of the constraints is temporarily or permanently violated, and (ii) whether executing a given task would induce a violation. This allows the system to inform users if a task is entirely forbidden

or only allowed under specific payload conditions (e.g., `RestartMachine` may not be executed by `userId=3` due to the target condition of a `NOTRESPONSE` constraint).

Limitations and discussion

While the three-layer architecture handles standard multi-perspective Declare templates, translating the full spectrum of STL and LTL_f into CEP syntax has three main technical limits:

- 1 Temporal logics permit arbitrary nesting of operators (e.g., $F(G(F_{[0,d]} \Theta))$). Esper's EPL uses a strict left-to-right sequential grammar via the linear-time developer operator ($\rightarrow$). Deeply nested temporal paths cannot always be expressed as a single continuous query. Executing them requires breaking the expression into a cascade of intermediate streams, in which **L2** feeds the complex-event output of a child sub-query back as an atomic trigger token into a parent statement. This is structurally achievable but adds configuration overhead and tracking complexity.
- 2 STL evaluates properties over true continuous-time real-valued functions. A discrete CEP engine processes inputs as discrete, chronologically ordered packets. If a physical signal fluctuates rapidly between two sensor ticks (e.g., a microsecond spike above a safety threshold), the engine cannot detect the transition unless the ingestion layer adds continuous interpolation or fine-grained edge sampling. Evaluation precision is therefore bounded by the sampling frequency of the physical IoT infrastructure.
- 3 Templates with long or unbounded metric intervals (e.g., tracking whether an event happens "anytime in the next three months") force the CEP engine to retain event tokens in memory for extended durations. For high-volume streams, this degrades performance and increases garbage collection. Our architecture mitigates this by focusing on metric temporal logic fragments with bounded sliding operational windows (Δt), which keeps memory profiles predictable and flat.
- 4 For complex interactions, data dependencies, or conflicting multi-perspective constraints, the engine evaluates each rule independently according to its compiled CEP logic. The engine does not currently include a priority or resolution mechanism for opposing actions. If a process engineer deploys contradictory constraints (such as a rule triggering an activity while an invariant forbids it), both rules execute independently, causing the process instance to stall or deadlock. We treat such operational conflicts as modeling errors in the process specification that must be resolved at design time before execution.

Evaluation

We evaluated the CEP execution architecture in two ways: a functional validation using an interactive dashboard, and a scalability benchmark measuring processing latency and resource use under different stream loads.

Functional validation & execution modes

The enforcement mechanisms of the architecture are demonstrated through an interactive process dashboard (cf. Fig. 5). The interface comprises a signal injection terminal for sending continuous telemetry samples and a task list that polls the `AnalyzerService` to determine which activities are currently executable. Together, these components exercise the two execution modes of the runtime engine:

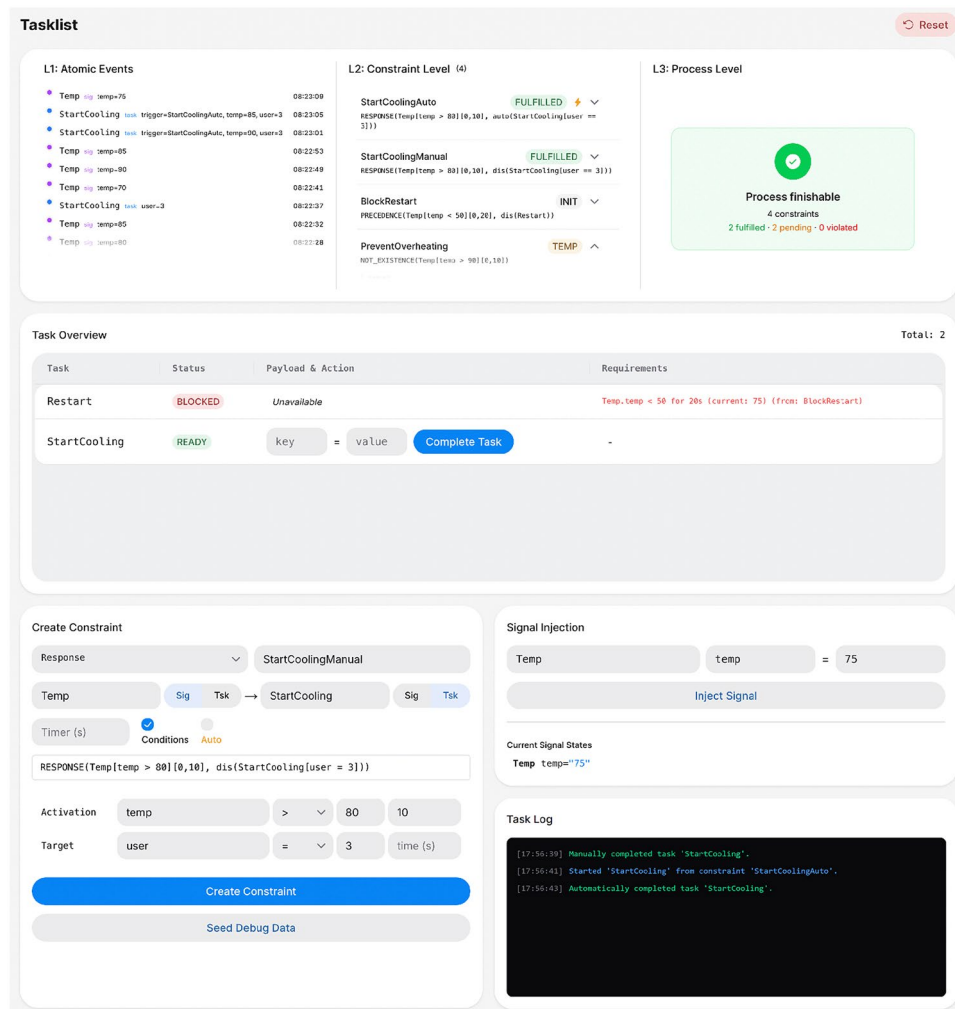


Fig. 5 Overview of implemented prototype frontend

- 1 The AnalyzerService evaluates each known task against the current constraint states and trace history. Only tasks that would not violate any active constraint are presented with an execute button; tasks blocked by unmet conditions (for instance, Restart while the temperature signal has not yet maintained the 20-second stability window required by Φ_3) are shown without an execution option, with the specific unmet requirements (e.g., $\text{temp} < 50$ for the required duration) displayed in a tooltip.⁹
- 2 Constraints can be configured with an auto-execute flag in the constraint creation interface. When the corresponding activation condition is detected (for example, a sensor push driving the telemetry stream above 80°C for 10seconds), the engine's listener triggers the TaskExecutorService, which actively executes the target task and emits its completion event into the stream. The dashboard reflects this loop in real time: the auto-executed task moves from the queue to the process trace without operator intervention, confirming that the architecture drives the process lifecycle directly in response to signal changes.

⁹This view is similar to existing implementations of tasks lists, and could be further adjusted in future iterations.

Scalability analysis

To evaluate the runtime scalability and resource efficiency of the architecture under high-frequency IoT signal volumes, we subjected the prototype engine to an intensive synthetic stress test. The entire execution suite was isolated within a Docker container limited strictly to 2 GB of RAM and 2 CPU cores. This resource ceiling simulates the processing constraints of industrial edge gateways, such as Raspberry Pi devices, deployed locally for shop-floor process control in Industry 4.0 environments.

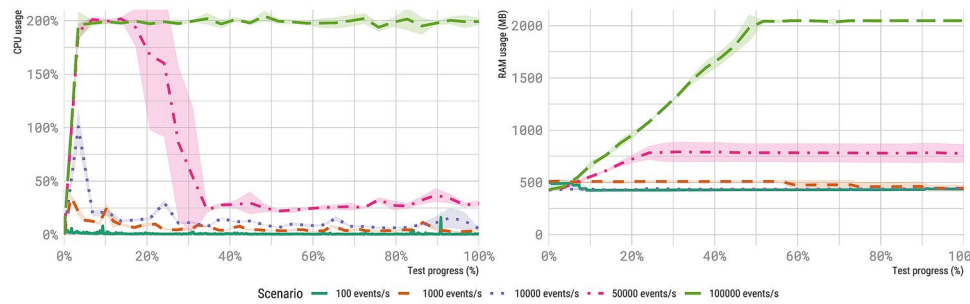
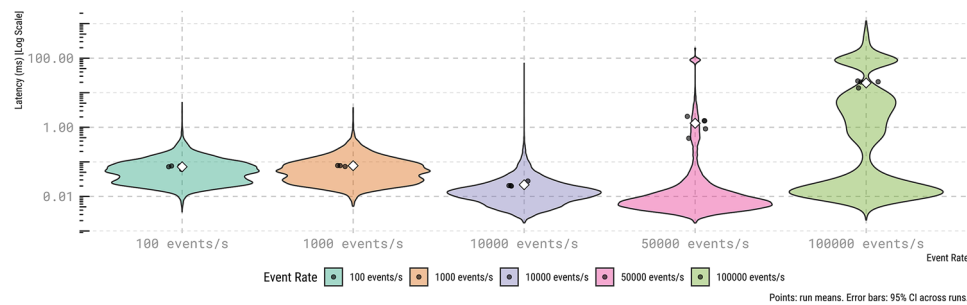
The test workload used a synthetic stream. The evaluation model consisted of 10 declarative constraints deployed simultaneously: five RESPONSE templates and five PRECEDENCE templates, each with randomized activation/target conditions and varying time windows. From the Esper engine's perspective, continuous signal events and discrete task events are structurally identical `GenericEvent` objects that differ only in their `eventType`; the EPL predicate (`cast(payload(...), double)`) runs the same bytecode whether the payload holds a temperature reading or a task attribute. A broader evaluation of templates for discrete-event data can be found in (Poss and Schöning, 2026). We subjected this configuration to five ingestion profiles: 100, 1,000, 10,000, 50,000, and 100,000 events per second. To gather robust latency samples at each rate, the observation window is scaled inversely to the event rate: the 100 EPS profile ran for 600 s ($\approx 60k$ events), 1,000 EPS for 60 s ($\approx 60k$ events), and the 10,000–100,000 EPS profiles for 30 s each ($\approx 300k$ – $3M$ events), with a minimum warm-up of 30 s across all configurations. Each profile was executed over $n = 5$ independent runs to compute averages (μ) and standard deviations (σ).

We define the evaluation metric as follows: *Processing Latency* measures the CEP detection delay (Δt) from the moment the *final stream element enabling an L1 match* enters the Atomic Layer (**L1**; recorded via `System.nanoTime()` during ingestion) until the Esper rule fires at the Constraint Layer (**L2**) and updates the constraint status to FULFILLED or TEMPORARY VIOLATION. For multi-event patterns, Δt is measured relative to the ingestion timestamp of the finalizing event. Latency samples triggered by timer-based PERMANENT VIOLATION rules are excluded from this measurement, as those rules fire after a configurable timeout window (`withinPeriod`) representing constraint semantics rather than engine overhead. Existing hybrid declarative frameworks perform asynchronous, offline, post hoc verification on static logs; they lack real-time ingestion, continuous pattern queries, and active execution loops. Our evaluation focuses on demonstrating that continuous signal analysis incurs minimal overhead during live process execution.

Table 2 reports resource utilization and latency measurements across the five load scenarios. The results show that the three-layer architecture scales efficiently for realistic IoT workloads. At event rates from 100 to 10,000 EPS, average CPU utilization scaled linearly from 0.9% to 14.4%, while memory consumption remained stable between 434 and 489 MB. Median end-to-end processing latency remained below 0.1 ms, with P_{99} values under 0.5 ms, confirming sub-millisecond response times consistent with linear scaling. At 50,000 EPS, CPU utilization rose to 66.9% with peak usage of 203.4%, indicating the onset of saturation on the 2-core configuration. Median latency remained low at 0.01 ms, though tail latency increased to $P_{99} = 73.3$ ms. At the stress ceiling of

Table 2 Performance and resource utilization metrics for scalability scenarios

Target Load	CPU (%)		Memory (MB)		Latency (ms)			Actual Throughput (events/s)
	μ	max	μ	max	μ	P ₉₀	P ₉₉	
100	0.9 ± 0.1	19.7 ± 2.2	434 ± 9	502 ± 30	0.1 ± 0.0	0.2 ± 0.0	0.4 ± 0.1	100.0
1,000	6.8 ± 0.4	35.3 ± 2.7	436 ± 19	514 ± 12	0.1 ± 0.0	0.2 ± 0.0	0.5 ± 0.1	998.9
10,000	14.4 ± 0.7	100.9 ± 17.2	489 ± 5	518 ± 4	0.1 ± 0.0	0.2 ± 0.0	0.7 ± 0.0	9991.8
50,000	66.9 ± 7.7	203.4 ± 3.2	733 ± 73	806 ± 91	2.4 ± 1.2	1.1 ± 0.6	73.3 ± 38.3	49882.4
100,000	192.1 ± 0.9	208.1 ± 5.2	1586 ± 11	2048 ± 0	39.3 ± 7.6	94.5 ± 3.8	442.9 ± 97.1	92513.6

**Fig. 6** CPU and memory utilization across event rates. Shaded bands show 95% confidence intervals across $n = 5$ runs**Fig. 7** End-to-end processing latency by event rate (logarithmic x-axis). Violins show the full latency distribution for each rate; white diamonds mark cross-run means with 95% confidence intervals; colored points show individual run means

100,000 EPS, the system averaged 192.1% CPU¹⁰, memory grew to 1,586 MB approaching the 2 GB budget, and median latency rose to 5.9 ms with $P_{99} = 442.9$ ms.¹¹

Figure 6 illustrates the linear scaling behavior of resource consumption as a function of event rate, with confidence intervals derived from $n = 5$ independent runs per rate. Figure 7 shows the latency distribution across event rates, with sub-millisecond response at typical IoT frequencies and graceful degradation under saturation. These results confirm that the layered CEP architecture introduces negligible overhead while meeting a critical requirement for IoT process control: genuine real-time constraint evaluation and enforcement. Overall, the evaluation confirms that hybrid declarative process execution

¹⁰CPU load is reported per core, so two cores equals maximum possible load of 200%.

¹¹As shown in (Poss and Schöning, 2026), JVM garbage collection occurs as long as sufficient headroom exists above the Esper and Quarkus baseline consumption.

is feasible on edge devices, addressing the core research gap of moving from post-hoc conformance checking to active runtime enforcement.

To validate these results on actual IoT hardware, we additionally deployed the same backend on a Raspberry Pi 5 Model B (8 GB RAM). Using the identical constraint configuration and workload generator, we ramped the event rate from 5,000 to 50,000 EPS in 5,000 EPS increments (30 s per step). The Pi sustained end-to-end latencies between 0.01 and 0.67 ms across all rates. Event delivery remained above 99.7% even at 50,000 EPS, indicating that the architecture had not yet reached its saturation point on the Pi 5. These results confirm that the CEP-based execution engine imposes negligible overhead on IoT-class hardware and can sustain throughput far exceeding typical industrial sensor frequencies without requiring cloud offloading or distributed infrastructure.

Related work

BPM has focused on modeling and monitoring processes using discrete-event data, capturing sequences of activities that represent process executions (Dumas et al., 2018). Declarative approaches, such as Declare, offer a flexible alternative by specifying constraints instead of explicit control flows, which is beneficial in dynamic environments (Pesic et al., 2007, Burattin et al., 2016). Classical declarative models and execution approaches (Ackermann et al., 2018, Käppel et al., 2019) address discrete-event traces and do not incorporate continuous sensor data, which is increasingly relevant in cyber-physical and data-rich domains (Bazan and Estevez, 2022, Janiesch et al., 2020). Recent work has advanced the integration of continuous data and discrete events by utilizing hybrid declarative models. Corea et al. (2025) extended declarative process models by adopting STL, originally developed for CPS, allowing temporal properties over real-valued signals to be formally expressed and verified. This enables conformance checking of hybrid traces combining boolean discrete events and continuous predicates. This approach primarily targets offline monitoring and post-hoc conformance, rather than real-time execution of hybrid specifications.

CEP systems have demonstrated potential for runtime analysis and reaction over large-scale real-time event streams, which suits the dynamic requirements of BPM in sensor-rich environments (Wu et al., 2006, Ruhkamp and Schöning, 2021). State-of-the-art CEP frameworks support declarative pattern specifications and efficiently evaluate complex event patterns (Ziehn et al., 2024). The authors in (Poss and Schöning, 2026, Ruhkamp and Schöning, 2021) propose a CEP-based monitoring architecture that integrates multi-perspective declarative models, enabling real-time monitoring of both sensor and event data. STL has a well-established role in specifying and monitoring continuous signals within CPS (Maler and Nickovic, 2004, Ahmad and Jeannin, 2021). Recent advancements extend its use to runtime enforcement, enabling proactive control based on continuous signal behavior. Ahmad and Jeannin (2021) introduced Signal Temporal Dynamic Logic (STdL), which enhances the expressiveness of reasoning about temporal properties over hybrid system executions.

While recent work has advanced hybrid specification and CEP-based monitoring, active execution of hybrid declarative models remains an open challenge. We integrate STL-inspired predicates into CEP engines to enable runtime enforcement of hybrid constraints.

Conclusion

This paper bridges the gap between hybrid declarative process specifications and operational control by introducing a CEP-based execution architecture for STL-enriched process models. Traditional declarative approaches have largely been limited to monitoring and post hoc conformance checking of hybrid traces, whereas our three-layer architecture enables real-time enforcement of constraints on both discrete events and continuous signals. By transforming STL-inspired predicates into CEP patterns, the system actively triggers activities and enforces process boundaries based on sensor behavior, extending declarative execution from purely discrete scenarios into IoT environments. Our evaluation demonstrates the practical feasibility of the approach on resource-constrained edge devices, achieving sub-millisecond latency at typical IoT event rates while maintaining stable resource consumption. This enables genuine real-time constraint evaluation in settings where processes must react to continuous sensor streams. The architecture successfully handles hybrid Declare templates, including unary, binary, and correlation-based constraints with both forward and backward temporal directionality. Future work will extend this architecture toward autonomous process control by leveraging STL robustness for graded, non-binary enforcement and by integrating machine learning techniques for predictive adaptation in distributed IoT/CPS environments.

Supplementary information

The online version contains supplementary material available at <https://doi.org/10.1007/s44311-026-00059-3>.

Supplementary Material 1

Author contributions

S. S. and L. P. conceptualized the general idea and laid the foundation for the approach. L. P. and S. S. wrote the main manuscript text, F. M. focused on the formal correctness and general applicability of the approach. L. P. implemented the prototype and focused on evaluating the approach. All authors reviewed and worked on revising the manuscript.

Funding

Open Access funding enabled and organized by Projekt DEAL. The authors did not receive support from any organization for the submitted work.

Data availability

No datasets were generated or analysed during the current study.

Declarations

Ethics declaration

Not applicable.

Competing interests

The authors declare no competing interests.

Received: 17 February 2026 / Accepted: 26 August 2026

Published online: 25 September 2026

References

- Ackermann L, Schönig S, Petter S, Schützenmeier N, Jablonski S (2018) Execution of multi-perspective declarative process models. In: On the Move to Meaningful Internet Systems. OTM 2018 Conferences. Springer, Cham, pp 154–172. https://doi.org/10.1007/978-3-030-02671-4_9
- Ahmad H, Jeannin J-B (2021) A program logic to verify signal temporal logic specifications of hybrid systems. In: Proceedings of the 24th International Conference on Hybrid Systems: Computation and Control. HSCC '21. ACM, New York City, pp 1–11. <https://doi.org/10.1145/3447928.3456648>
- Artikis A, Margara A, Ugarte M, Vansummeren S, Weidlich M (2017) Complex event recognition languages: tutorial. In: Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems. DEBS '17. ACM, New York City, pp 7–10. <https://doi.org/10.1145/3093742.3095106>

- Bazan P, Estevez E (2022) Industry 4.0 and business process management: state of the art and new challenges. *Bus Process Manag J* 28(1):62–80. <https://doi.org/10.1108/bpmj-04-2020-0163>
- Burattin A, Maggi FM, Sperduti A (2016) Conformance checking based on multi-perspective declarative process models. *Expert Syst Appl* 65:194–211. <https://doi.org/10.1016/j.eswa.2016.08.040>
- Cimatti A, Roveri M, Tonetta S (2009) Requirements validation for hybrid systems. In: *Computer Aided Verification*. Springer, Cham, pp 188–203. https://doi.org/10.1007/978-3-642-02658-4_17
- Corea C, Alman A, Maggi FM, Wittlinger PH (2025) Declarative process specifications over discrete/continuous event data. In: *Advanced Information Systems Engineering*. Springer, Cham, pp 277–294. https://doi.org/10.1007/978-3-031-94571-7_16
- Cugola G, Margara A (2012) Processing flows of information: From data stream to complex event processing. *ACM Comput Surv* 44(3):1–62. <https://doi.org/10.1145/2187671.2187677>
- Dayarathna M, Perera S (2019) Recent advancements in event processing. *ACM Comput Surv* 51(2):1–36. <https://doi.org/10.1145/3170432>
- Donzé A, Maler O (2010) Robust satisfaction of temporal logic over real-valued signals. In: *Formal Modeling and Analysis of Timed Systems*. Springer, Cham, pp 92–106. https://doi.org/10.1007/978-3-642-15297-9_9
- Dumas M, La Rosa M, Mendling J, Reijers HA (2018) *Fundamentals of Business Process Management*. Springer, Cham. <https://doi.org/10.1007/978-3-662-56509-4>
- Greß A, Riveros C, Ugarte M (2019) A formal framework for complex event processing. *Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany*. <https://doi.org/10.4230/LIPICS.ICDT.2019.5>
- Janiesch C, Koschmider A, Mecella M, Weber B, Burattin A, Di Ciccio C, Fortino G, Gal A, Kannengiesser U, Leotta F, Mannhardt F, Marrella A, Mendling J, Oberweis A, Reichert M, Rinderle-Ma S, Serral E, Song W, Su J, Torres V, Weidlich M, Weske M, Zhang L (2020) The internet of things meets business process management: a manifesto. *IEEE Syst Man Cybern Mag* 6(4):34–44. <https://doi.org/10.1109/msmc.2020.3003135>
- Käppel M, Schützenmeier N, Schöning S, Ackermann L, Jablonski S (2019) Logic based look-ahead for the execution of multi-perspective declarative processes. In: *Enterprise, Business-Process and Information Systems Modeling*, vol 352. Springer, Cham, pp 53–68. https://doi.org/10.1007/978-3-030-20618-5_4
- Langhi S, Tommasini R, Bonifati A, Bernhardt T (2025) Epl: The event processing language for streaming data. In: *Proceedings of the 19th ACM International Conference on Distributed and Event-based Systems*. DEBS '25. ACM, New York City, pp 62–73. <https://doi.org/10.1145/3701717.3730546>
- Luckham D (2008) *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Springer, Cham, pp 3–3. https://doi.org/10.1007/978-3-540-88808-6_2
- Maggi FM (2019) Declarative process mining. In: *Encyclopedia of Big Data Technologies*. Springer, Cham, pp 625–632. https://doi.org/10.1007/978-3-319-77525-8_92
- Maler O, Nickovic D (2004) Monitoring temporal properties of continuous signals. In: *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. Springer, Cham, pp 152–166. https://doi.org/10.1007/978-3-540-30206-3_12
- Pesic M, Schonenberg H, Aalst WMP (2007) Declare: Full support for loosely-structured processes. In: *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*. IEEE, New York City, pp 287–287. <https://doi.org/10.1109/edoc.2007.14>
- Pnueli A (1977) The temporal logic of programs. In: *18th Annual Symposium on Foundations of Computer Science (SFCS)*. IEEE, pp 46–57. <https://doi.org/10.1109/sfcs.1977.32>
- Poss L, Schöning S (2026) A synergistic engine paradigm for real-time, context-aware decision-making: integrating declarative processes and event streams. *Complex Intell Syst* 12(4). <https://doi.org/10.1007/s40747-026-02253-z>
- Ruhkamp N, Schöning S (2021) Execution of multi-perspective declarative process models using complex event processing. *Bus Inf Syst* 1:95–104. <https://doi.org/10.52825/bis.v1i.51>
- Wu E, Diao Y, Rizvi S (2006) High-performance complex event processing over streams. In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD/PODS06. ACM, New York City, pp 407–418. <https://doi.org/10.1145/1142473.1142520>
- Ziehn A, Grulich PM, Zeuch S, Markl V (2024) Bridging the gap: complex event processing on stream processing systems. In: *27th International Conference on Extending Database Technology (EDBT)*. <https://doi.org/10.48786/EDBT.2024.39>

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.