Complex & Intelligent Systems (2026) 12:136
https://doi.org/10.1007/s40747-026-02253-z

ORIGINAL ARTICLE

A synergistic engine paradigm for real-time, context-aware decision-making: integrating declarative processes and event streams

Leo Poss[1] · Stefan Schönig[1]

Received: 26 August 2025 / Accepted: 25 October 2025

Abstract
The abstraction gap between high-frequency IoT data and high-level business process logic creates a significant bottleneck for modern enterprises. Current architectures typically rely on separate middleware for event preprocessing, which introduces significant latency due to network hops and data serialization, and increases architectural complexity, creating multiple points of failure that hinder responsive operations. This paper introduces a synergistic engine paradigm that resolves this gap by leveraging a single complex event processing engine for both event abstraction and the direct execution of declarative MP-Declare models. Through a multi-level abstraction framework, process constraints are translated into executable queries, as demonstrated by a proof-of-concept. This unified approach provides a simplified architectural foundation for building highly responsive, event-driven applications that adapt intelligently to real-time conditions, as demonstrated by a proof-of-concept and a quantitative evaluation showing sub-millisecond latency at up to 10,000 events per second.

Keywords Declarative process execution · Complex event processing · Business process management · Event-driven architecture

Motivation

Modern enterprises operate in increasingly dynamic environments, demanding Innovative Information Systems (IIS) that can facilitate real-time, context-aware decision-making. The widespread growth of the Internet of Things (IoT) provides an ever-increasing stream of high-frequency data, offering organizations the potential to achieve highly responsive and intelligent operations [1, 2], which is a cornerstone for realizing data-centric smart environments [3]. This vision aligns with Event-Driven Architectures (EDA), where events guide process execution as they occur [4, 5].

However, a fundamental challenge prevents this vision from being fully realized: a persistent *abstraction gap* between the low-level, high-frequency event streams from IoT sensors and the high-level, semantic activities of a business process. Current architectures typically attempt to bridge this gap using separate middleware for event preprocessing [6]. This approach creates a fragmented and complex landscape that introduces critical bottlenecks, including significant latency from network hops and data serialization, increased integration complexity, and multiple points of failure, ultimately hindering responsive operations and delaying critical business decisions [7]. This architectural deficiency creates a significant bottleneck, preventing enterprises from fully leveraging real-time data to adapt intelligently to changing conditions.

To address this challenge, this paper introduces a novel synergistic engine paradigm that unifies event processing and declarative process execution within a single, unified system. Instead of treating event abstraction and process enactment as separate concerns handled by different technologies, we propose using a standard complex event processing (CEP) engine for both event abstraction and process enactment. The core of our approach is the direct execution of declarative, constraint-based process models—specifically MP-Declare [8, 9] from raw event streams. Declarative models are uniquely suited for the unpredictable nature of modern operations, as they define the rules of a process rather than a rigid path, affording the flexibility that dynamic environments demand [10]. By translating declarative constraints

✉ Leo Poss
leo.poss@ur.de

Stefan Schönig
stefan.schoenig@ur.de

1 University of Regensburg, Universitätsstrasse 31, 93053 Regensburg, Germany

into executable CEP queries, our paradigm eliminates the semantic gap and renders dedicated preprocessing middleware obsolete, offering a streamlined architecture for truly responsive, event-driven process execution.

This paradigm motivates a re-examination of the relationship between event processing and process execution, leading to our primary research questions:

RQ1. How can the core CEP capabilities of pattern matching, event filtering, and stream aggregation be re-envisioned to natively support and directly enact the execution semantics of declarative, constraint-based process models?

RQ2. How do these CEP mechanisms and declarative process paradigms synergistically enable novel architectural frameworks for process-aware information systems?

To answer these questions, this research follows the Design Science Research (DSR) methodology [11] to develop two key artifacts: (1) a multi-level event abstraction framework that formalizes the translation of MP-Declare constraints into executable CEP queries, and (2) a proof-of-concept implementation demonstrating the paradigm's feasibility. This implementation is then subjected to a rigorous quantitative performance evaluation to validate its scalability and low-latency characteristics under various workloads. Our findings confirm that this unified approach is viable, directly enacting declarative process logic from event streams and significantly reducing system complexity by eliminating the need for separate components.

The remainder of this paper is structured as follows: "Background and related work" provides the necessary background on (declarative) process management and complex event processing, and positions the research within related work across both domains and their intersection. "Research approach" outlines the Design Science Research (DSR) approach adopted. "Design and development" presents the conceptual design ("Concept") and the corresponding proof-of-concept implementation ("Implementation", with indepth information in the Appendix A). These are subsequently evaluated and discussed in "Evaluation" and "Discussion and limitations", respectively, before concluding in "Conclusion and future work".

Background and related work

Business process management (BPM) *Business Process Management* (BPM), a key driver of business value, enables the digitization, automation, and strategic management of operational processes [12]. A business process is a structured sequence of activities aimed at achieving an organizational objective [10], formalized through process models. These models serve a dual purpose: supporting stakeholder communication and providing executable specifications for Business Process Management Systems (BPMS) [12].

Imperative vs. declarative process management and modeling Process modeling paradigms differ significantly in how they define execution flow. Imperative approaches, such as BPMN, employ a flow-oriented method that prescribes activity sequences, permissible steps, and resource requirements, offering procedural clarity but limiting adaptability in dynamic contexts [13]. In contrast, declarative modeling defines constraints rather than explicit paths. It specifies a start, an outcome, and rules that must be adhered to, deeming any execution path valid if it does not violate these constraints. This *open-world assumption* significantly enhances flexibility, especially for complex or unpredictable processes [12, 14].

Declarative and multi-perspective declarative process modeling Declare is a prominent declarative approach that allows specifying processes via a set of constraints, providing templates for modeling flexible workflows [8]. To lay a foundation for shared understanding, the Response constraint that requires that one event A is eventually followed by an event B is formally defined using LTL_f [15] as: $\mathbf{G}(A \rightarrow \mathbf{F}B)$, which means **G**lobally, every A is eventually **F**ollowed by an event B (see Table 3 for further constraint specifications). However, a central shortcoming of the standard Declare language is that its constraints apply only to activities, while other critical perspectives, such as time and data, are often ignored. This is a significant limitation in real-world scenarios, such as IoT applications, where these factors are essential for modeling realistic processes.

To address this, the language was extended to MP-Declare, which enhances expressiveness by integrating multiple viewpoints (e.g., resource, time, data) directly into the process model [16]. The semantics of Declare are extended with data-aware conditions that refer to the payload of events. To satisfy a constraint, an *activation condition* (φ_a), a *correlation condition* (φ_c), and a *target condition* (φ_t) must be fulfilled.

These additions can be demonstrated using the example of a `Response(A, B)` constraint in a manufacturing process which would be formalized as $\mathbf{G}((A \wedge \varphi_a(x)) \rightarrow \mathbf{F}(B \wedge \varphi_c(x, y) \wedge \varphi_t(y)))$, which now includes all three types of conditions. With standard Declare, the constraint might be Response(*VibrationAnomaly*, *ScheduleMaintenance*). This ensures that maintenance is eventually scheduled after an unusual vibration is detected. With MP-Declare, the rule becomes much more intelligent:

- The activation condition (φ_a) could specify that the rule only triggers if the machine is a bottleneck asset in the production line, e.g., `machine.isCritical`

= `true`. An anomaly on a redundant or non-critical machine might be logged, but wouldn't trigger this high-priority workflow.

- The correlation condition (φ_c) could link the specific error code from the vibration sensor to the required maintenance team. For example, the scheduled technician must have a certification matching the machine's component type, e.g., `alert.componentType = technician.certification`.
- The target condition (φ_t) could ensure that the maintenance is scheduled during a planned downtime window to minimize production loss, such as `maintenance.startTime` is within `productionSchedule.plannedDowntime`.

The complete MP-Declare constraint describes a highly efficient process: if a critical machine shows a vibration anomaly, maintenance is scheduled with a correctly certified technician during the next planned downtime.

A second example can be demonstrated with the `AlternateResponse(A, B)` constraint, which ensures a flexible yet immediate reaction. It would be formalized[1] as $\mathbf{G}((A \wedge \varphi_a(x)) \rightarrow \mathbf{X}(\neg(A \wedge \varphi_a(x))\mathbf{U}(B \wedge \varphi_c(x, y) \wedge \varphi_t(y))))$. With standard Declare, the constraint might be ALTERNATERESPONSE(*QualityDefectDetected*, *ProductHandled*). This specifies that after a defect is found, the product must be handled immediately, without another defect being detected on the same product in the interim. With MP-Declare, this handling becomes a precise, context-aware decision:

- The activation condition (φ_a) could use data from a computer vision system to trigger the rule only for products on a high-value assembly line, e.g., `product.line = 'premiumElectronics'`.
- The correlation condition (φ_c) links the nature of the defect to the handling action. The target event *ProductHandled* can be either *RerouteToRework* or *RerouteToScrap*, and the choice is dictated by the defect's data payload, e.g., `(handling.type = 'rework' AND defect.isRepairable = true) OR (handling.type = 'scrap' AND defect.isRepairable = false)`.
- The target condition (φ_t) could add a compliance requirement to the handling action itself, ensuring that the decision is logged for auditing purposes by the supervisor on duty, e.g., `handling.log.supervisorID = 'currentQASupervisor'`.

Here, the MP-Declare constraint describes an automated and auditable quality assurance process: if a defect is detected on the premium electronics line, the product is immediately rerouted to the appropriate station (rework or scrap) based on its repairability, and the action is formally logged by the responsible supervisor.

[1] LTL defines **X** as next and **Y** as previous, **U** as until, **O** (once) as its counterpart, and **S** for since (cf. Pnueli [15]).

Internet of Things (IoT) and Big Data

The Internet of Things (IoT) further expands the possibilities for event generation by connecting uniquely identifiable tangible entities equipped with sensors and actuators [17]. This network enables continuous data gathering from the physical world (e.g., location tracking, environmental conditions) and allows for influencing the environment through actuators based on processed information. IoT fundamentally enables new functionalities by bridging the physical and digital realms.

The vast amount of heterogeneous data generated at high velocity by IoT devices contributes significantly to the phenomenon known as Big Data, often characterized by its volume, velocity, and variety [5, 18]. Effectively managing and extracting value from these massive data streams requires approaches such as EDA and technologies like CEP. They provide the architectural and processing capabilities required to handle the scale and speed of IoT data, enabling real-time analysis, pattern detection, and informed decision-making.

Event-Driven Systems (EDS) *Event-Driven Systems* (EDS), or *Event-Driven Architectures* (EDA), represent a design paradigm centered on the flow and processing of events. These systems often rely on *Complex Event Processing* to analyze and interpret event data originating from high-volume, heterogeneous streams [19]. A common mechanism for defining reactions is the *Event-Condition-Action* (ECA) rule model [20], in which a specific event triggers an action when a condition is met. While straightforward, ECA is less expressive than constraint-based declarative approaches, such as MP-Declare, for defining complex behaviors [8]. CEP is a technology that continuously monitors data streams to derive meaningful, potentially abstract events [21].

Related work

To establish a comprehensive foundation for our research, we conducted a structured literature review following the methodology proposed by Okoli [22]. This section synthesizes existing research at the intersection of BPM, CEP, and IoT, highlighting the current state of knowledge and identifying research gaps. Our literature search strategy focused on three domains: Business Process Management, Complex Event Processing, and the Internet of Things. Following Brereton et al. [23], we iteratively refined search queries for each domain shown in Table 1 and applied them across important digital libraries of information systems, including *ACM*, *IEEE Xplore*, *EBSCO*, *ScienceDirect*, and *AIS eLibrary*. To ensure relevance and methodological rigor,

Table 1 Overview of domain-specific search strings

Domain	Query
BPM	("MP-Declare" OR "Multi-Perspective Declare" OR "Event-Driven" OR "Event-Based") AND ("Event Stream" OR "Event Log") AND ("Process Management" OR "Process Execution" OR BPM)
CEP	("Complex Event Processing" OR CEP OR "Event Processing" OR "Event-Driven Architecture" OR EDA) AND ("Business Process Management" OR BPM OR "Process Management" OR "Process Execution")
IoT	(IoT OR "Sensor Data") AND ("Event Processing" OR CEP OR "Complex Event Processing") AND ("Process Management" OR BPM OR "Business Process Management")

we established the following criteria: **(IC1)** research must address practical applications of event-driven process execution; **(IC2)** research must integrate (IoT) event data in BPM: We excluded: **(EC1)** non-English publications; **(EC2)** non-peer-reviewed papers; and **(EC3)** studies that focus solely on theoretical aspects, formal CEP elements, or process modeling without IoT connections. The selection process followed PRISMA guidelines [24], ensuring transparency and applicability. We categorized selected publications along three dimensions (see Table 2): the *Data Source Focus*, referring to the types of data used (e.g., traditional enterprise data, IoT sensor data); the *Integration Approach*, describing how CEP integrates with BPM (e.g., for process execution, monitoring, or hybrid approaches); and the *Process Model Paradigm*, indicating the underlying process model type (imperative, declarative, or hybrid).

BPM and CEP Integration Soffer et al. [7] provide a comprehensive overview of approaches for combining BPM and CEP across imperative, declarative, and hybrid paradigms. Their classification identifies four quadrants of interaction between these domains: CEP for process mining, CEP for enhancing process model expressiveness, deriving CEP rules from process models, and executing business processes through CEP. Our concept will address the latter quadrant, extending it with high-frequency IoT data integration.

Several studies have explored the transformation of imperative models into event-based representations: Kopp et al. [25] convert BPEL block structures into events, preserving BPMN compatibility while noting the differences in expressiveness. In contrast, Cicekli and Cicekli [26] introduce a control flow graph approach that incorporates basic patterns, such as sequence and concurrency, and Hens et al. [27] decompose imperative models using LTL-based transformation rules, enabling the direct execution of process activities.

On the other hand, declarative frameworks offer greater flexibility for event-driven processes: Sadoghi et al. [28], for example, use a Guard-Stage-Milestone (GSM) model with ECA rules for CEP-based monitoring of events, while Soffer [29] presents a simplified ECA approach. Hildebrandt and Mukkamala [30] introduce Dynamic Condition Response Graphs to model activities as events within constraints, enabling a form of declarative execution based on graphs.

As a hybrid approach, van der Aalst et al. [13] integrate imperative and declarative modules in the *YAWL* engine, with events supporting log generation and monitoring rather than direct execution.

In addition, there are other diverse strategies for combining CEP and BPM: Cicekli and Cicekli [26] introduce the Event Calculus for workflow specification, Daum et al. [32] propose collaboration models, such as message broadcasting and event interception, and Janiesch et al. [33] suggest enhanced communication through standardized event formats. Finally, von Ammon et al. [34] conceptualize CEP as a parallel platform for BPM event definition and processing.

While our work focuses on the direct execution of declarative process models, other research has explored alternative formalisms, such as using high-level Petri nets for intelligent conflict detection in complex IoT service environments [50].

In contrasting our synergistic paradigm with existing approaches, two key differentiators emerge: architectural simplicity and the role of the CEP engine. While hybrid systems, such as the YAWL engine [13], integrate declarative modules, events are primarily used for logging and monitoring rather than driving the core execution logic. Similarly, most IoT integration patterns rely on distinct middleware or *bridging layers* that maintain a fundamental separation between event abstraction and process execution. This separation inherently introduces latency and complexity. Our approach is novel in that it elevates the CEP engine from a supporting component to the central execution runtime, directly enacting declarative logic and thereby eliminating the need for these intermediary layers.

BPM, IoT, and CEP Integration. We identified a critical research gap in IoT data integration, where predominant middleware-based preprocessing approaches introduce unnecessary complexity and potential failure points. Our work addresses this gap by proposing a novel direct integration solution, eliminating intermediary layers that currently bridge the abstraction gap between raw IoT data and business processes. Kirikkayis et al. [35] employ middleware for data acquisition, processing, and storage, connecting to BPM systems via interfaces or decision tables. Hu et al. [36] use a "bridging layer" to transform device data into business objects, similar to Valderas et al. [37] integrating synchronization and data structure layers within a microservice architecture with CEP mediation and Wehlitz et al. [39] integrating complex multi-system communication via plat-

Table 2 Concept matrix of BPM-CEP-IoT integration approaches following [49]

Reference	Year	Data sources	Integration type	Process paradigm	Key contribution	Limitations
CEP and BPM integration approaches						
Kopp et al. [25]	2011	T	E	Imp	Transformation of BPEL block structures to event-based representation	Tightly coupled to BPMN semantics; expressiveness limitations
Cicekli and Cicekli [26]	2006	T	E	Imp	Control flow graph methodology with Event Calculus formalization	Focuses on event-based process definition rather than true CEP integration
Hens et al. [27]	2014	T	E	Imp	Decomposition of imperative models into event-processable components using LTL	Limited to standard workflow patterns; event rules primarily for process coordination
Sadoghi et al. [28]	2015	T	M	Dec	GSM artifact model with ECA rules governed by sentries	CEP capabilities limited to monitoring functions; lacks execution integration
Soffer [29]	2013	T	M	Dec	Streamlined ECA-based approach for event processing	Primarily monitoring-focused; limited process execution capabilities
Hildebrand and Mukkamala [30]	2011	T	M	Dec	Dynamic Condition Response Graphs for flexible modeling	Conceptualizes activities as events rather than leveraging true event processing
van der Aalst et al. [31]	2009	T	M	Hyb	YAWL engine with combined imperative and declarative modules	Events primarily used for logging/monitoring rather than driving execution
Daum et al. [32]	2012	T	H	Imp	CEP as foundation system integrating BPMS events (Esper)	Processes function primarily as event sources for CEP; limited bidirectional integration

Table 2 continued

Reference	Year	Data sources	Integration type	Process paradigm	Key contribution	Limitations
Janiesch et al. [33]	2012	T	H	Mix	Bidirectional communication between BPMS and CEP; decision support focus	Requires specialized event producers and CEP support; maintains system separation
von Ammon et al. [34]	2010	T	H	Mix	Conceptual integration framework for CEP and BPM	Positions CEP as parallel platform for event analysis; lacks implementation details
IoT data integration approaches						
Kirikkayis et al. [35]	2022	IoT	Mix	Imp	Modeling, execution, and monitoring of IoT data within BPMS	Depends on middleware; implementation specifics underdeveloped
Hu et al. [36]	2023	IoT	Mix	Imp	Explicit "bridging layer" between IoT devices and BPM systems	Relies on multiple abstraction layers; increases architectural complexity
Valderas et al. [37]	2022	IoT	H	Mix	Synchronization/data layer in microservices; dual event buses with CEP	Complex middleware dependencies; dual abstraction levels
Panetti et al. [38]	2021	IoT	Mix	Mix	Process over Things (PoT) approach for IoT integration	Requires explicit broker between fog computing and BPM
Wehlitz et al. [39]	2018	IoT	H	Mix	Integration via multi-system communication infrastructure	Complex architecture with numerous middleware components

Table 2 continued

Reference	Year	Data sources	Integration type	Process paradigm	Key contribution	Limitations
Seiger et al. [40]	2022	IoT	H	Imp	Central process engine connected to CEP and specialized handlers	Depends on multiple integration components and middleware
Specialized applications						
Malburg et al. [41]	2020	IoT	M	Imp	CEP for monitoring faulty process executions using IoT data	Separate systems with direct database storage; limited runtime integration
Stoiber and Schönig [42]	2021	IoT	H	Imp	Systematic review of event-driven BPM in IoT; introduces "Event Mediator" for hybrid execution and monitoring; taxonomic classification of integration approaches	Middleware-dependent design; complex mediator architecture
Janiesch and Matzner [43]	2019	T	M	Imp	Business Activity Monitoring Notation; event-based activity tracking	Monitoring-centric approach; conventional data sources
Mandal et al. [44, 45]	2016	IoT	H	Imp	Framework leveraging IoT/CEP; standard event generation for process models	Implied middleware dependency; standardization focus
Other Approaches						
He et al. [46]	2019	IoT	H	Imp	Event-driven process control elements; hierarchical event structuring	Middleware-dependent; emphasizes CEP construct extension

Table 2 continued

Reference	Year	Data sources	Integration type	Process paradigm	Key contribution	Limitations
Decker et al. [47]	2007	T	M	Mix	Graphical modeling of complex events (BPMN/UML)	Concentrates on modeling rather than execution/integration
Krumeich et al. [48]	2014	T	P	Mix	Predictive/prescriptive capabilities using event processing data	Analysis-focused; limited core execution integration

Data sources: **T**raditional (enterprise systems data), **IoT** (sensor/device data), **C** (combined traditional and IoT data); *Integration Type*: **E**xecution (process execution driven by events), **M**onitoring (events for observation), **H**ybrid (combined execution and monitoring), **P**redictive (analytical applications), **Mix** (variable/multiple approaches); *Process Paradigm*: **Imp**erative (activity-focused), **Decl**arative (constraint-focused), **Hybrid** (combined), **Mix** (variable/unspecified)

form connectors and streaming platforms. Seiger et al. [40] place the process engine at the core, connected to various handlers, including CEP engines.

Our research introduces a novel, tightly integrated approach for event-driven BPM in dynamic, sensor-rich IoT environments. We address critical limitations by unifying declarative process specifications with CEP. A key distinction of our method is the ability to explicitly incorporate high-frequency IoT sensor data, enabling real-time process monitoring and adaptation driven by fine-grained environmental signals. Leveraging the inherent adaptability of the MP-Declare paradigm, which surpasses traditional imperative models in dynamic settings, our framework uniquely facilitates the direct execution of processes via CEP. This eliminates the need for preprocessing or middleware layers, significantly reducing system complexity and execution latency.

Research approach

This research aims to provide a first-of-its-kind synergistic integration of CEP and process execution, enabling declarative process models to run on standard CEP engines by defining the necessary constraints and events. We position our contribution within the design science research (DSR) paradigm [11, 51], applying the seven DSR guidelines outlined by Hevner et al. [52].

Design as an artifact This research produces an executable artifact (*method* and *instantiation* following [53]) for executing declarative process models using common CEP techniques. This method includes (i) transforming declarative constraints to a series of CEP events, (ii) formally defining the conceptual approach for the execution, and (iii) providing a proof-of-concept implementation for execution.

Problem relevance The domains of BPM and CEP are closely related (see below), and their integration has been a core part of research in recent years. Current approaches for high-frequency IoT data rely on traditional system architectures that include centralized control and data preprocessing, as traditional BPMSs are not designed to handle real-time sensor data streams. In addition to concretizing the architecture, our approach enables the execution of declarative process models, grounding this in a system-agnostic, formally defined approach.

Design evaluation The artifact is evaluated through a multi-phase approach (cf. "Evaluation"). This aligns with the DSRM process model [54], which emphasizes rigorous evaluation. We specifically consider the Framework for Evaluation in Design Science (FEDS) by Venable et al. [51] to structure our evaluation strategy, focusing on functional correctness and practical utility.

Research contributions The concept and implementation make clear and verifiable contributions to BPM by providing a practical approach for integrating real-time data into a unified platform that enables efficient processing and process execution. This offers a new perspective and novel *solution* [55] on declarative process models and their synergy with event-based architecture and execution.

Research rigor This research uses rigorous methods to construct and evaluate the artifact, based on theories and approaches in BPM, CEP, and information systems, which aligns with the DSR principle of research rigor (using standard research techniques in both the build and evaluation phases [52]).

Design as a search process The design is an iterative process involving multiple cycles of refinement based on feedback from stakeholders and domain experts, as well as ongoing internal review. This approach is reflected in the evaluation phase, where findings inform potential refinements to the artifact and suggest avenues for future research.

Communication of research The findings of this research are disseminated through publication. The research presentation includes the problem definition, solution objectives, design and development of the artifact, demonstration, and evaluation. The artifact is available for future improvements or modifications to enhance functionality (see "Implementation").

Design and development

Our framework integrates declarative process specifications with real-time event processing by enabling continuous monitoring and detection of complex event patterns from heterogeneous sources. The following design objectives guide the design of our framework:

❶ **Enable intelligent and responsive operations**: Support flexible process execution and runtime adaptation of process instances based on real-time event data from heterogeneous sources, including reacting to deviation from predefined activity sequences, or adding new activities in response to changing conditions exploiting the conceptual advantage of *microprocesses* (cf. [7, 56]) enabling just-in-time decisions in dynamic environments.

❷ **Leverage contextual parameters for real-time compliance**: Continuously monitor and enforce declarative constraints using real-time event data to ensure compliance within dynamic operational contexts.

❸ **Provide a clear separation of concerns**: Decouple the specification of *process logic* (declarative constraints) from *processing events*, simplifying the development and maintenance of process models, allowing for the independent evolution of process and event-processing logic.

Concept

The basic idea of synergistic integration is to manage declarative constraints using complex events, i.e., to describe and model the connections between different states of constraints in an event-based way by examining and querying a continuous flow of events at different levels of abstraction. Next to abstracting and semantically enriching the events at each level transition, we also remove any ambiguities in the constraint-level events, which is fundamental for all alternate and chain constraints, whereas every atomic event could falsely influence the constraint.

Fundamentally, Declare constraints defined in LTL_f can be separated into an activation and a target statement that form an event. Intuitively, activating a constraint constitutes a triggering condition that expects the fulfillment of the target condition during process execution. Table 3 shows an overview of the semantics of Declare constraints, including the LTL_f expression and the expected activation and target, as well as the category of constraint for this concept (see below). The synergistic approach is based on three basic steps and abstraction layers that are similar for all constraints (Fig. 1):

1. Detection of activation and target events from *atomic events* for each constraint that triggers another event at a higher abstraction level (`INSERT INTO`, see "Implementation").
2. Stateless management of the current constraint status using *constraint level events* for activation, target, and basic additional events like process start and end. This layer is also required to detect interruptions and more complex constraints, such as AlternatePrecedence (`SELECT`).
3. Current status of each constraint (`Listener`), i.e., process status, that implies currently available tasks at *process level*.

The four examples shown in Fig. 1 show the basic idea: based on atomic events at the lowest level, we can detect *activation* and *target* events for different constraints; these are input to the next abstraction level that contains constraint levels, which are then again used in complex events to detect the current status of Declare constraints. Examining the first examples Response(A, B) and NotResponse(A, B), we observe that event A occurs, which serves as the activation event for both constraints. The formal goal for the first constraint is to satisfy its LTL$_f$ expression, $\mathbf{G}(A \rightarrow \mathbf{F}B)$. While the Response constraints' status proceeds to 'temporarily violated,' which means we are waiting for the target (here: B) to happen to fulfill the constraint, the NotResponse constraint, defined by $\mathbf{G}(A \rightarrow \neg\mathbf{F}B)$, is 'fulfilled.' As soon as we detect the *target* event, the first constraint is 'fulfilled,' while the second one will now be 'permanently violated' as B is not allowed to happen after A. Another technique is needed for the Existence(D) constraint. The constraint is set to be 'temporarily violated' at the start of the process instance. As soon as we detect the *activation*, it is set to 'fulfilled.' After the process instance finishes, we can set it to 'permanently violated,' as D did not occur during this time frame. Finally, a more complex constraint, such as Precedence(D, C), requires looking back to detect if a *target* occurred before the *activation*. As we continually track

Table 3 Semantics of selected Declare constraints [9, 16]

Template	LTL$_f$ expression [14, 57]	Activation	Target	Category: Forward	Category: Backward	Category: State
EXISTENCE(A)	$\top \rightarrow \mathbf{F}A \vee \mathbf{O}A$	**start**	$\mathbf{F}A \vee \mathbf{O}A$	●	○	R
RESPONDEDEXISTENCE(A,B)	$\mathbf{G}(A \rightarrow (\mathbf{O}B \vee \mathbf{F}B))$	A	$(\mathbf{O}B \vee \mathbf{F}B)$	●	●	R
RESPONSE(A,B)	$\mathbf{G}(A \rightarrow \mathbf{F}B)$	A	$\mathbf{F}B$	●	○	F
ALTERNATERESPONSE(A,B)	$\mathbf{G}(A \rightarrow \mathbf{X}(\neg A \mathbf{U} B))$	A	$\mathbf{X}(\neg A \mathbf{U} B)$	●	○	V
CHAINRESPONSE(A,B)	$\mathbf{G}(A \rightarrow \mathbf{X}B)$	A	$\mathbf{X}B$	●	○	V
PRECEDENCE(A,B)	$\mathbf{G}(B \rightarrow \mathbf{O}A)$	B	$\mathbf{O}A$	○	●	V
ALTERNATEPRECEDENCE(A,B)	$\mathbf{G}(B \rightarrow \mathbf{Y}(\neg B \mathbf{S} A))$	B	$\mathbf{Y}(\neg B \mathbf{S} A)$	○	●	V
CHAINPRECEDENCE(A,B)	$\mathbf{G}(B \rightarrow \mathbf{Y}A)$	B	$\mathbf{Y}A$	○	●	V
NOTRESPONSE(A,B)	$\mathbf{G}(A \rightarrow \neg\mathbf{F}B)$	A	$\neg\mathbf{F}B$	●	○	F
NOTPRECEDENCE(A,B)	$\mathbf{G}(B \rightarrow \neg\mathbf{O}A)$	A	$\neg\mathbf{O}A$	○	●	V

Category State: R: Runtime, F: Fulfillment, V: Violation

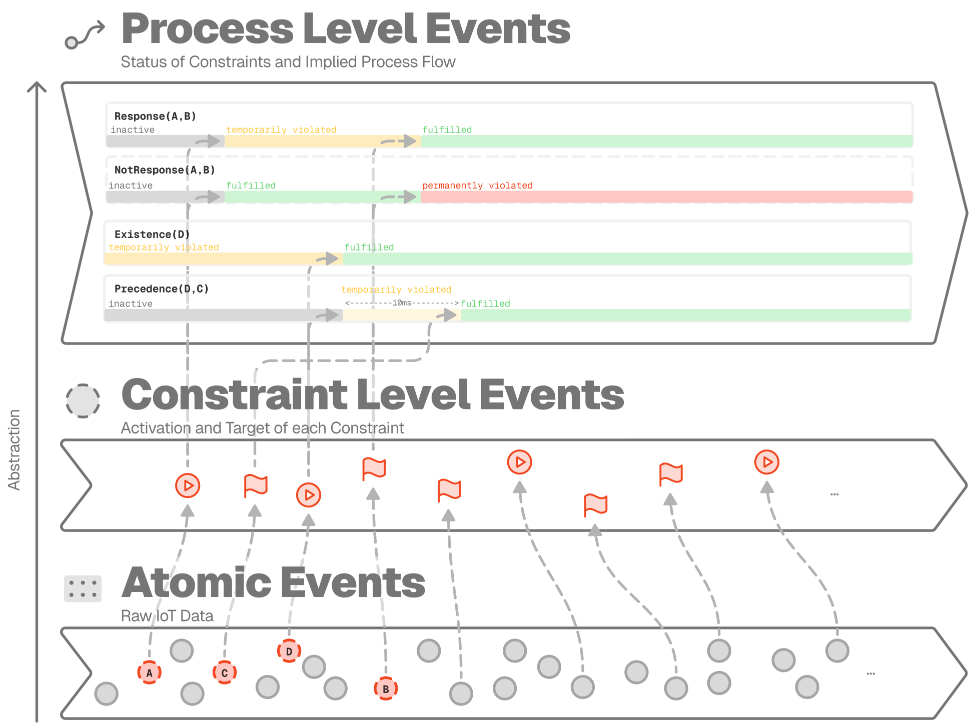


Fig. 1 Overview of different abstraction layers and events

all *activation* and *target* events, we can wait for the *activation* to occur and then check if, at our constraint level, a suitable *target* has already happened. This then either sets the constraint to be 'permanently violated' or, in our case, to be 'fulfilled.' For application- and event-based execution, we further categorize the constraints. As detailed in Table 3, we further categorize constraints for event-based execution into two categories:[2] those that examine the temporal relationship between *activation* and *target*, and those that consider possible transitions between constraint states. These constraint states can then be used to determine which tasks can be executed next based on their constraints, ultimately defining the process execution (cf. Di Ciccio and Montali [9, 58]).

Categories

To formulate the different constraints using CEP to detect activation and fulfillment, we can categorize them based on two characteristics: *temporal relation* and *constraint state type*.

[2] The activation, correlation, and target condition formulae are then attached to either side to support MP-Declare (see examples).

Temporal relationship The detection mechanisms for declarative constraints vary significantly based on their temporal orientation. For this, we introduce three distinct categories that have differential detection requirements:

1. **Forward-looking constraints**: These constraints are the most straightforward constraints; the activation occurs, and the target occurs. RESPONSE(A, B), for example, falls into this category. Simply put, if event A occurs, then B must follow in the future.
2. **Backward-looking constraints**: Opposite to the previous constraint, these constraints have *to look in the past*, i.e., as soon as the activation of a constraint occurs, it must be checked whether the target had happened before. One example is PRECEDENCE(A, B), where all cases are selected in which event A activated the constraints, and the target B has already occurred before under the given conditions.
3. **Hybrid constraints**: Finally, the combination of both constraint types results in hybrid constraints like RESPONDEDEXISTENCE(A, B) that are both forward and backward-looking and say if A happens, B must also happen. After the activation occurs, we must check in both directions: either B must follow A in the future, or the target B has already happened, which satisfies and fulfills the constraint.

Determining constraint state: fulfillment vs. violation The following classification outlines the constraint lifecycle phases and their operational implications for our concept, including the number of required queries for detection:

1. **Instantly fulfilled constraints:** When the activation condition occurs and is later followed by the required target condition, the constraint is fulfilled. For example, in RESPONSE(*OrderReceived*, *PaymentProcessed*), the constraint transitions from active to fulfilled once an order is received and subsequently processed for payment. The monitoring system tracks these activations until their fulfillment criteria are satisfied.
2. **Unrecoverably violated constraints:** This category encompasses Negating/Vulnerable constraints that specify prohibited patterns rather than required fulfillment. In NOTRESPONSE(*CreditCheckFailed*, *LoanApproved*), the constraint is violated if a credit check failure is followed by loan approval. These constraints are monitored by detecting inverse patterns; violations immediately invalidate the process instance. This requires continuous monitoring for prohibited sequences in addition to tracking fulfillment states.
3. **Runtime constraints:** This represents the constraints imposed on the runtime of each process instance and must start as temporarily violated or fulfilled and then transition to their respective states. An example is EXISTENCE(*PaymentVerification*), a fundamental cardinality requirement in declarative process models. It specifies that the *PaymentVerification* activity must occur at least once during process execution. Unlike temporal relationship constraints, which define the relative ordering of events, this constraint requires the presence of a specific event without specifying its temporal position during the process instance's runtime.

Process instances

For real-world applications, we must consider that multiple concurrent process instances are running. This implies the need to provide additional context throughout all layers and events to enable mapping events to specific process instances. This can be achieved by adding a parameter to each event that uniquely identifies a process instance (e.g., an order ID), by connecting events and instances, or by using tools provided by most CEP implementations. This can be `CONTEXT`s in Esper,[3] which allows for a later grouping by the parameter or context that represents the process instances, or the respective `keyBy` in Apache Flink[4] or `groupBy` in Apache Spark.[5]

Multi-perspective declare

Since MP-Declare allows us to add conditions in the form of payloads or parameters to the activation and target events of constraints, we can add additional constraint context and parameters to the CEP query. Even the correlation condition that must be satisfied in the activation and target events can be modeled as an additional payload for the query. If the activation or target condition is not met, the event will not be recognized by the CEP engine as expected. This payload mechanism enables our system to achieve context-awareness. For example, in a constraint RESPONSE(*MachineFailure*, *OrderMaintenance*), adding the payload `isProductionCritical=true` as either activation, correlation, or target condition (cf. "Background and related work"), allows the system to analyze the event's situational parameters and make an intelligent, context-specific decision, which is a core capability of an advanced Innovative Information System (IIS).

Implementation

Based on the conceptual model above, we now implement the requirements for synergistic integration. This involves

[3] https://www.espertech.com/esper/.

[4] https://flink.apache.org/.

[5] https://spark.apache.org/.

Listing 1 Query to detect RESPONSE activation and target and insert fulfillment `constraintStatus` event.

```
INSERT INTO constraintStatus
SELECT id, 'Resp' as name, 'ACTIVATION'
     as type
FROM GenericEvent WHERE eventType = '
    OrderReceived'

INSERT INTO constraintStatus
SELECT id, 'Resp' as name, 'TARGET' as
    type
FROM GenericEvent WHERE eventType = '
    PaymentProcessed'

SELECT a.id, a.name, a.type
FROM PATTERN [every a=constraintStatus(
    type='ACTIVATION', name='Resp') ->
    b=constraintStatus(type='TARGET',
    name='Resp')]
```

transforming all required events into queries that enable detection and integrating them into an executable program. CEP engines fundamentally differ in their approaches to event and query definition: Apache Flink utilizes functional APIs with embedded pattern definitions. At the same time, Esper's SQL-like EPL provides a clear separation between pattern logic (the *what*) and execution mechanics (the *how*), enabling dynamic query construction and deployment.

Exemplary constraints

Before describing the implementation, we present two exemplary constraints and their implementation specifics. Next to runtime constraints, the simplest constraint is detecting RESPONSE(A, B), which is formally defined by the LTL$_f$ formula $\mathbf{G}(A \rightarrow \mathbf{F}B)$. This requires detecting the *activation* A and the *target* B (*Atomic Events*, see Fig. 1). Based on these two events (*Constraint Level Events*), we can derive the current status of the constraint (*Process Level Event*). As soon as we detect the *activation*, the constraint is 'temporarily violated.' If we detect an *activation* followed by a *target*, we can set it to 'fulfilled,' as the response constraint is defined in Declare. Taking the example from above RESPONSE(*OrderReceived*, *PaymentProcessed*), this means detecting an event of type *OrderReceived* that triggers an event with type *activation* and a unique constraint identifier, as well as a similar event with role *target* shown in Listing 1. Additionally, we listen for these two events at a higher abstraction level, where we update the constraint state to 'temporarily violated' if we detect the *activation* and to 'fulfilled' if we detect the complex event described by *activation* followed by a corresponding *target*.

A complex constraint like ALTERNATEPRECEDENCE (*ShippingLabelGenerated*,*AuthorizationGranted*), defined as $\mathbf{G}((e(\text{ShippingLabelGenerated})) \rightarrow \mathbf{Y}(\neg(e(\text{ShippingLabel Generated}))\mathbf{S}(e(\text{AuthorizationGranted})))$, could be part of the same ordering process, where a return approval allows the generation of exactly one return shipping label. If the customer loses the label and needs another one, triggering the "generate label" event again after the initial approval is disallowed. For this, we need additional events to model the constraint. In addition to detecting *activation* and *target*, we must also detect a possible violation and update the constraint status from the middle layer accordingly. To detect precedence, after the *activation* is detected and the constraint is set to 'temporarily violated,' we must retrospectively look for a possible *target* that has already occurred, which is done by reversing the events in Listing 1. As this would also apply to future events, we need to manually create an event that violates this query if the precedence is not detected within a short timeframe. These possibilities are illustrated in Fig. 2: if we detect the *target*, the constraint is 'temporarily violated,' as we are unsure whether we have already detected a matching *target* or must be 'permanently violated.' If we have previously detected a *target*, the constraint is 'fulfilled'; if not, it is 'permanently violated.' Finally, if we detect another *target* event in between, the constraint is also set to 'permanently violated.' All other constraints can be constructed similarly based on Table 3; see Appendix A for details on our implementation.

Proof-of-concept implementation

To evaluate the concept of executing declarative process models and high-frequency data, possibly from IoT devices, we will use a proof-of-concept implementation [52] based on standard web technologies (Quarkus[6] and Next.js[7]) and Esper as a CEP engine that includes all the queries for the constraints listed in Table 3. Using Esper also enables us to easily add new queries at runtime and to simplify the creation of string-based constraints and the queries needed for them. The implementation[8] consists of two main components: the frontend, which is needed for the user to interact with, and the backend, which handles the actual execution of the model. The frontend as shown in Fig. 3 contains three different sections: one for creating new constraints, i.e., modeling the process, top left, and injecting or sending events (in practice, this would include both actors finishing their respective tasks and high-frequency IoT data), top right[9] Here, we can toggle MP-Declare com-

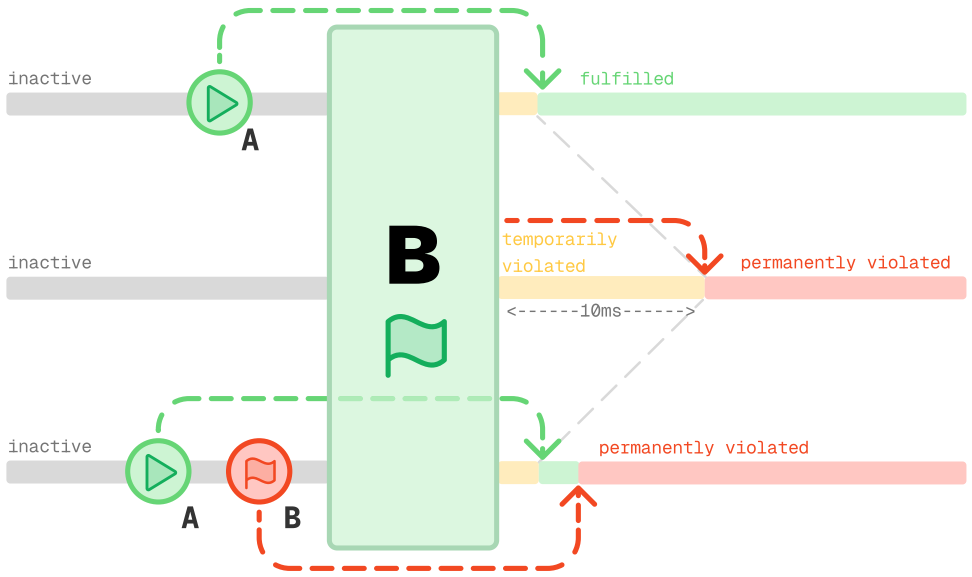
[6] https://quarkus.io/.

[7] https://nextjs.org/.

[8] The source code for the proof-of-concept can be found at https://github.com/LeoPoss/quarkusCEP/tree/base.

[9] A short, guided overview of the implementation can be found at https://youtu.be/2qUTwzLKCxM.

Fig. 2 Rule detection of ALTERNATEPRECEDENCE



ponents for activation, target, and correlation conditions, and event payloads. This is followed by an overview of currently deployed constraints and their events. For example, for RESPONSE(A, B[`critical` = `true`]) this shows EPL queries for detecting the *activation* and *target* for the constraint (inserting a new event of higher abstraction or semantic meaning into `constraintStatus`), as well as updating the status to 'temporary violation' and 'fulfillment' (querying complex events from `constraintStatus`). Important to note is that the constraints each have different possible transitions, a simple EXISTENCE constraint can only be 'fulfilled', whereas PRECEDENCE constraints contain queries for 'fulfillment', 'temporary violation', and 'permanent violation'. The backend integrates the CEP Engine and provides RESTful endpoints for creating and querying constraints, as well as receiving events. We implemented the concept presented above by splitting and structuring the constraints into smaller parts that could be handled as events in CEP: detecting *activation* and *target* events based on the data, and then handling constraint states at a higher semantical level in the constraint status layer and handling the additional logic for the constraints. A detailed breakdown of the implementation architecture and exemplary EPL queries is provided in the Appendix A.

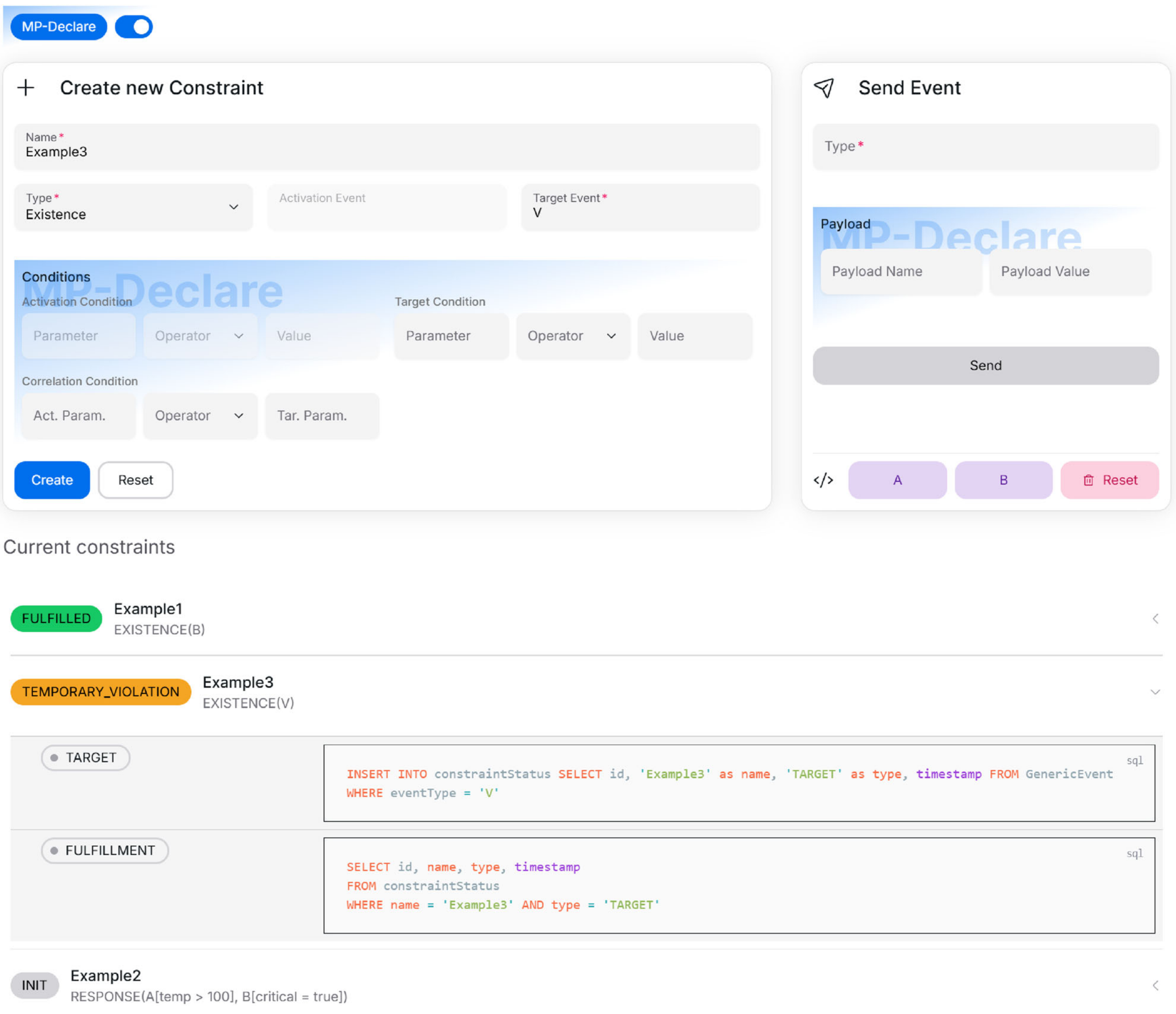
Evaluation

The evaluation of our synergistic integration artifact follows well-established DSRM principles [11, 52]. As Venable et al. [51] asserts, artifact functionality constitutes a fundamental evaluation dimension, particularly for novel technical implementations that bridge previously disparate paradigms. Our evaluation strategy uses a multifaceted approach that progresses from technical validation to practical application assessment and quantitative performance evaluation.

Functional validation

The technical evaluation of our artifact focuses on two critical dimensions: functional correctness and integrity of the abstraction layer. This corresponds to ex-ante assessment during the design cycles (*formative*) and ex-post evaluation of the final artifact (*summative*) [51].

Regarding functional correctness, we systematically validated the artifact's ability to correctly detect and handle the spectrum of constraint types defined for (MP-)Declare [8, 9, 16] (cf. Table 3). This includes all types of constraints and the constraint state categories. Each constraint type was evaluated against formal definitions from literature [9] to ensure semantic fidelity and functionality. This involved defining test cases with synthetic event sequences and verifying that the CEP engine produced the expected outcomes. We also verified the coherence and integrity of the information flow across the three abstraction layers. Particular attention was paid to the propagation of events between layers (Atomic, Constraint Level, Process Level) and the management of constraint states throughout process execution. This technical validation provides evidence that the synergistic integration concept is theoretically coherent. It directly answers **RQ1** by demonstrating that declarative constraints can be transformed into event-based representations while preserving semantic integrity across temporal relationship categories and constraint state types.



Name * Example3

Type * Existence

Activation Event

Target Event * V

FULFILLED Example1 EXISTENCE(B)

TEMPORARY_VIOLATION Example3 EXISTENCE(V)

TARGET

```sql
INSERT INTO constraintStatus SELECT id, 'Example3' as name, 'TARGET' as type, timestamp FROM GenericEvent
WHERE eventType = 'V'
```

FULFILLMENT

```sql
SELECT id, name, type, timestamp
FROM constraintStatus
WHERE name = 'Example3' AND type = 'TARGET'
```

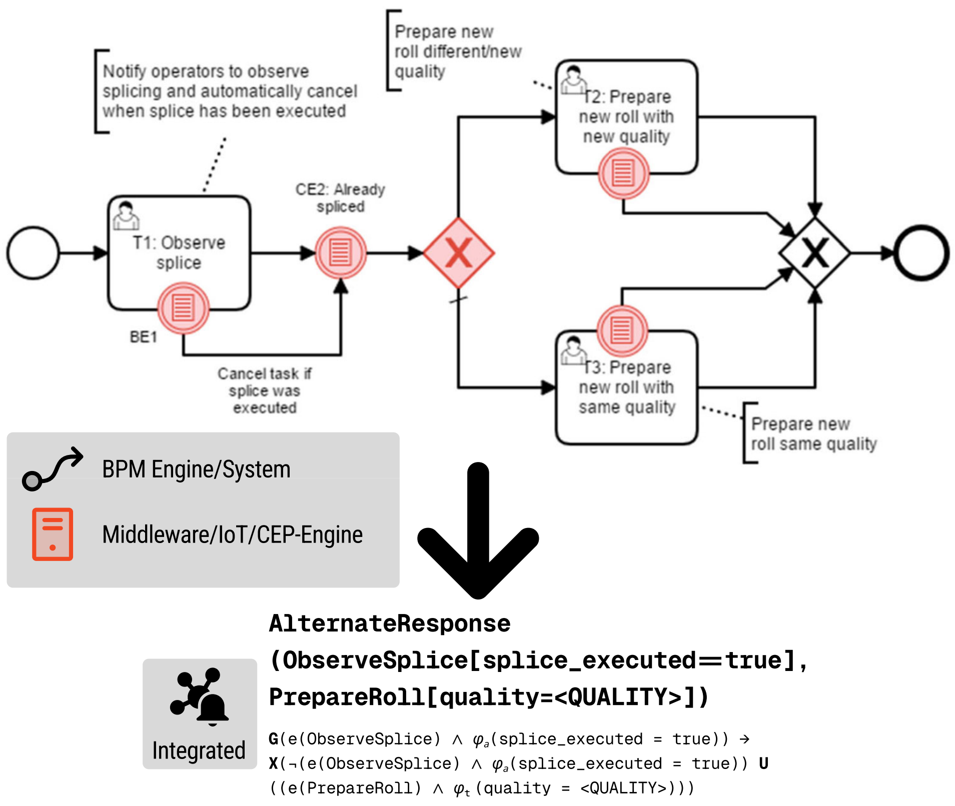
INIT Example2 RESPONSE(A[temp > 100], B[critical = true])

Fig. 3 Implementation with creation of constraints, sending events with MP-Declare, and the current constraint states and event queries

Use case application I: industry process

To demonstrate the practical application of our artifact, we apply it to a real-world manufacturing process from [1]. This use case exemplifies the need for real-time, context-aware decision-making in a dynamic industrial environment. The imperative process model and the resulting declarative version are shown in Fig. 4. The version of Schönig et al. [1] consists of two systems: handling and processing external data (middleware), preparing required data, as seen in many related works, and the engine itself. The process involves gluing together paper to create corrugated paper, the raw material used to produce cardboard boxes. To ensure continuous splicing, the next but one roll has to be prepared at the end of each roll. After observing the splice based on raw machine data with no remaining meters on the roll, either a roll of the same quality must be manually prepared or an alternative roll must be prepared. Using declarative process modeling, we can simplify it into a single system and a single ALTERNATERESPONSE constraint that includes the attribute conditions for activation. Compared to the imperative approach, which comprises 11 elements and requires two execution systems, our approach relies on a single system, and the declarative paradigm enables us to reduce it to a single ALTERNATERESPONSE constraint. This constraint is formally expressed in LTL as: $\mathbf{G}((e(\text{ObserveSplice}) \wedge \varphi_a(\text{spliceExecuted} = \text{true})) \rightarrow \mathbf{X}(\neg(e(\text{ObserveSplice}) \wedge \varphi_a(\text{spliceExecuted} = \text{true}))\mathbf{U}(e(\text{PrepareRoll}) \wedge \varphi_t(\text{quality} = \text{QUALITY}))))$ This formula mandates that after every `ObserveSplice` event, a `PrepareRoll` event must occur, with no other `ObserveSplice` events happening in between. Compared to the imperative approach, which requires two separate systems, our unified IIS enables execution with a single system and reduces the logic to a single

Fig. 4 Process snippet [1] and resulting MP-declare constraint


Prepare new roll different/new quality
Notify operators to observe splicing and automatically cancel when splice has been executed
T2: Prepare new roll with new quality
CE2: Already spliced
T1: Observe splice
BE1
Cancel task if splice was executed
T3: Prepare new roll with same quality
Prepare new roll same quality
BPM Engine/System
Middleware/IoT/CEP-Engine
Integrated
AlternateResponse
(ObserveSplice[splice_executed=true],
PrepareRoll[quality=<QUALITY>])
G(e(ObserveSplice) ∧ φa(splice_executed = true)) →
X(¬(e(ObserveSplice) ∧ φa(splice_executed = true)) U
((e(PrepareRoll) ∧ φt (quality = <QUALITY>)))

constraint. This architectural simplification is critical for developing robust and scalable intelligent systems, reducing potential points of failure, and streamlining process management.

Use case application II: high-value logistics

To further demonstrate the paradigm's applicability, we adapt a second real-world use case involving the transport of high-value, environmentally sensitive goods between two company branches, as described in Poss and Schönig [59]. This scenario highlights the need to process not just location data but also other high-frequency IoT sensor streams to ensure compliance and quality.

In the original process, a parcel containing orthopedic medical products is fitted with sensors for humidity and temperature. The delivery process involves multiple systems: one system for dispatch and task allocation, a separate IoT platform for monitoring sensor data, and manual checks at the destination. If a sensor threshold is breached during transit (e.g., humidity is too high), an alert may be logged in the monitoring system; however, reacting to it requires manual intervention and communication between departments, introducing significant delays and a potential for human error. To this end, the presented unified engine paradigm simplifies the process by modeling the entire logic within a single system. The business rules, sensor data conditions, and process flow are defined as a set of MP-Declare constraints, which are executed directly from the incoming event streams.

This eliminates the architectural complexity of integrating a separate BPMS and IoT monitoring platform. Critical compliance checks are no longer passive monitoring rules; they are integral, executable parts of the process logic. The core rules of the process can be modeled with the following MP-Declare constraints:

1. **Ensuring proper handover**: The transport cannot begin until the package has been officially scanned and registered at the departing branch. This is a simple but critical sequencing rule.
   - *Constraint*: PRECEDENCE(`Package ScannedAtBranch`, `BeginTransport`)
     The formal LTL$_f$ representation of this precedence constraint is: $\mathbf{G}(e(\text{PackageScannedAtBranch})) \rightarrow \mathbf{O}(e(\text{BeginTransport}))$. This specifies that always, if a `BeginTransport` event occurs, a `Package ScannedAtBranch` event must have occurred at some point in the past.
   - *Execution*: The CEP engine will not recognize a `BeginTransport` event as valid for the process instance unless a `PackageScannedAtBranch`

event for the same parcel ID has already been received.

2. **Real-time environmental compliance**: This is the most critical constraint. If the humidity exceeds a critical threshold at any point during transport, an escalation process must be triggered immediately. This moves the logic from a passive monitoring system directly into the core process execution.
   - *Constraint*: RESPONSE(`TransportEvent [humidity > 60%]`, `TriggerEscalationProcess`)
     Formally, this response to a sensor data condition is expressed as: $\mathbf{G}(e(\text{TransportEvent}) \wedge \varphi_a(\text{humidity} > 60\%) \rightarrow \mathbf{F}(e(\text{TriggerEscalationProcess})))$. This ensures that if a high-humidity event ever occurs, an escalation process must eventually follow.
   - *Execution*: The engine continuously processes `TransportEvent` messages, which contain location and sensor payloads. If any event arrives with a humidity value greater than 60, the constraint is activated, and the engine immediately expects a `TriggerEscalationProcess` event, ensuring an automated, instantaneous reaction.

By using these declarative constraints, the logistics process becomes more agile, resilient, and architecturally simple. There is no longer a delay between sensing an event (such as a humidity spike) and acting on it, as the sense-and-respond logic is unified within a single engine.

Quantitative performance evaluation

To empirically validate the performance, scalability, and practical viability of the synergistic engine paradigm, we conducted a comprehensive quantitative evaluation. The primary objective was to move beyond the conceptual proof of concept and rigorously assess the system's operational characteristics under realistic and stress-test conditions. We designed a series of experiments to measure the system's throughput, end-to-end latency, and resource utilization as a function of two key variables: the rate of incoming events and the complexity of the declarative process model.

This section is structured as follows: *First*, we detail the experimental setup, including the hardware, software, and measurement methodology. *Second*, we present a scalability analysis to understand how the system behaves under varying event loads. *Third*, we conduct a model-complexity analysis to assess the performance impact of executing more intricate declarative models. *Finally*, we synthesize these findings in a concluding discussion.

Experimental setup

Methodological rigor is essential for producing reliable performance metrics. Therefore, we established a controlled and reproducible experimental environment.

Hardware and Software Environment All experiments were executed on a host machine equipped with an Intel Core i7-12700K CPU (12 cores, 20 threads) and 96GB of DDR4 RAM. To ensure consistent resource allocation and isolate the test workload, the entire system was packaged and run within a Docker container. Unless otherwise specified, the container was strictly limited to 2 CPU cores and 4GB of RAM, simulating a typical cloud/edge, or lowest-end on-premise deployment scenario. The implementation itself is built on a standard technology stack: the backend logic is managed by the Quarkus framework, and the core stream processing is handled by the Esper CEP engine.

Methodology and Metrics To ensure a consistent evaluation, each experimental run followed a two-phase process. The first phase occurred at the start of each test. During this phase, the declarative process model, defined by a specific set of MP-Declare constraints, was deployed to the CEP engine. We measured the one-time cost of model initialization, which is not directly related to the presented artifact but rather to the use of declarative models for overall process execution. Following this, the second phase began, in which a dedicated client application started sending events to the system's REST endpoint at a precise, preconfigured rate. This event stream continued for a sustained period of n seconds, allowing the system to reach a steady state and capture performance over time. We captured a set of key performance indicators (KPIs) to form a comprehensive view of the system's behavior:

- **Actual throughput (events per second)**: This measures the raw processing capacity of the system. It is calculated as the total number of events successfully processed during the test window, ensuring that the system can keep up with the increasing event rate without dropping events.
- **End-to-end latency**: This is arguably the most critical metric for a real-time system. We define it as the time elapsed from the moment an event is received by the application's endpoint (ingress timestamp) to the moment the CEP engine signals a definitive change in a constraint's state based on that event (egress timestamp). This E2E measurement is important because it includes not only the CEP engine's processing time but also the application framework's overhead. We report the mean (μ), median (P_{50}), 95th percentile (P_{95}), and 99th percentile (P_{99}) latencies to understand both the typical and worst-case performance.

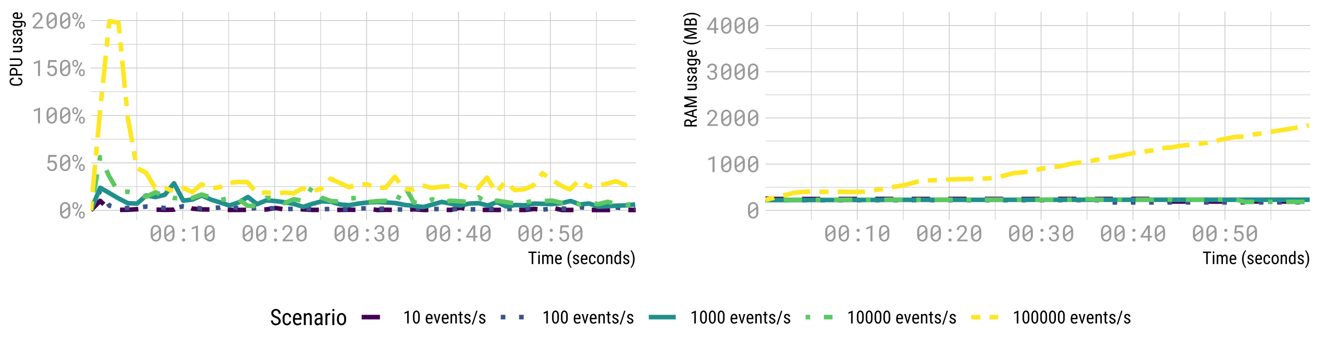


Fig. 5 CPU and RAM Usage for 10 active constraints with varying event rates

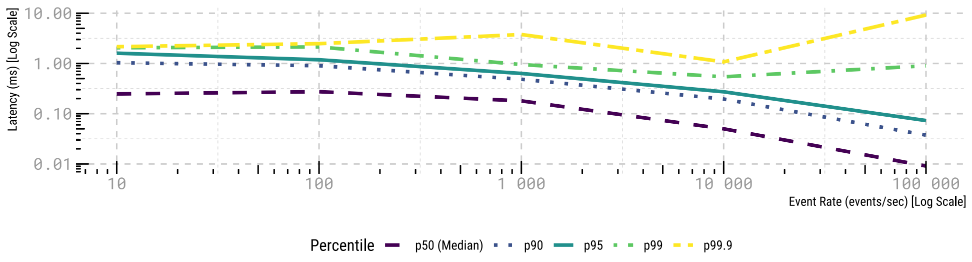


Fig. 6 Percentiles of latency for varying event rates

Table 4 Performance and resource utilization metrics for scalability scenarios

Scenario	Latency (ms)					CPU (%)		Memory (MB)		Actual throughput (events/s)
	μ	P_{50}	P_{95}	P_{99}	Max	Avg.	Max.	Avg.	Max.	
10 events/s	0.47	0.25	1.60	2.06	2.17	0.96	9.97	230.57	251.73	10.00
100 events/s	0.44	0.27	1.18	2.13	2.52	2.38	20.57	201.40	221.06	99.89
1000 events/s	0.25	0.18	0.63	0.95	5.07	8.49	28.51	227.13	230.03	998.90
10,000 events/s	0.09	0.05	0.27	0.54	2.45	12.05	56.12	224.13	243.82	9,989.16
100,000 events/s	0.11	0.01	0.07	0.90	88.90	33.82	199.45	955.24	1841.95	99,943.66

- **Resource utilization**: To assess the system's efficiency, we monitored the CPU (%)[10] and Memory (MB) consumption from the perspective of the Docker container. We captured both the average and maximum utilization over the test period to identify both steady-state behavior and potential resource spikes.

Scalability analysis

The primary objective of our evaluation was to assess the system's ability to scale with increasing data velocity. This is critical for IoT-driven contexts where event streams can range from sparse to torrential. In these scenarios, we maintained the process model's complexity by using a simple model with 10 active Response constraints and 10 active Precedence constraints, while varying the event rate from 10 to 100,000 events/s over 60 s. In addition, 99% of all events were noise that did not correspond to any constraint, allowing for uncertainty and variability in the workload and creating a scenario close to the real world.

The results, presented in Fig. 5 and Table 4, reveal a system that scales gracefully under pressure. The linear increase in RAM usage is characteristic of stream processing systems, as memory is consumed to buffer and manage in-flight events. The CPU utilization, however, remains remarkably low across a wide range of loads, averaging only 12.05% at 10,000 events per second. This indicates that the core processing logic is highly optimized and not CPU-bound, even under a significant load.

Perhaps the most compelling finding is the system's latency performance, detailed in Fig. 6. The median (P_{50}) latency remains well under 1 millisecond across all scenarios, while the 99th percentile (P_{99}), a key indicator of worst-case responsiveness, stays below the 1ms threshold up to 10,000 events per second.[11] Interestingly, the median

[10] Typically, each CPU core can reach 100% usage, which means that with four CPU cores, the total usage can be as high as 400%.

[11] The actual throughput of events per second was measured and included in the result tables: the differences to the target rates are negligible (0.11–0.05%).

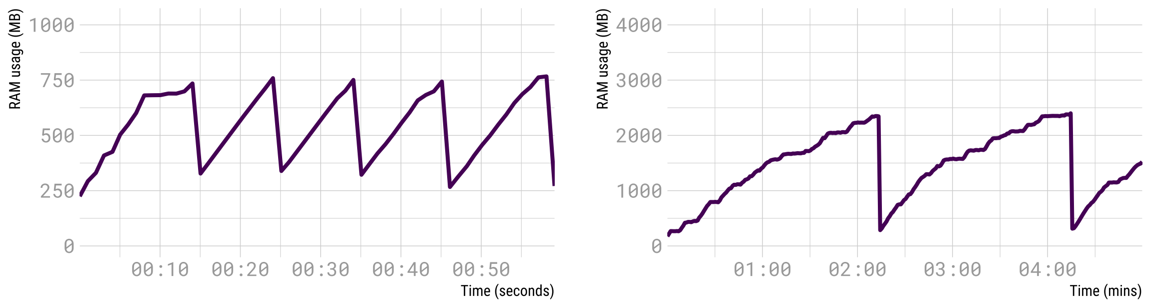


Fig. 7 CPU and RAM usage running into RAM shortages triggering garbage collection (60 s with 1GB vs. 5 min with 4 GB)

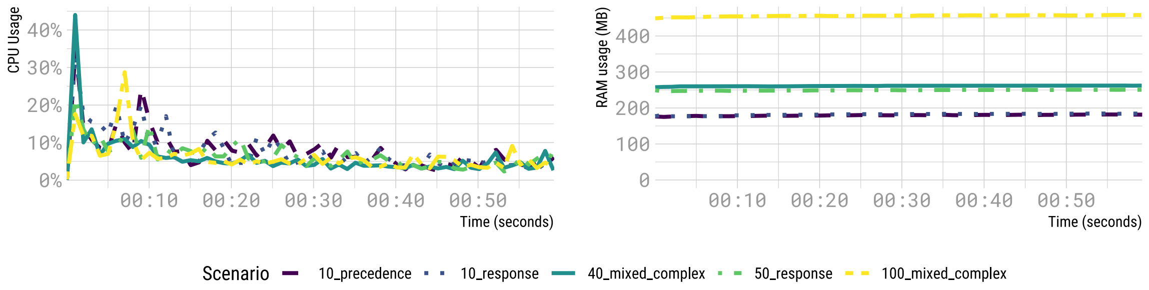


Fig. 8 CPU and RAM usage for constant rate of 1000 events per second with varying number and type of constraints

latency decreases as throughput increases. This phenomenon is often attributed to Just-In-Time (JIT) compilation and other dynamic optimizations within the Java Virtual Machine (JVM), as well as classical caching mechanisms included in CEP engines. At higher, more consistent event rates, the JVM "warms up," compiling frequently executed code paths into highly efficient native code, thereby reducing per-event processing time.

The system's throughput was robust, maintaining a pace with the ingress rate of up to 10,000 events per second. It was only when pushed to the extreme of 100,000 events per second in a resource-constrained environment (1 CPU, 512 MB RAM) that the system began to shed load, processing only 66,600 events per second. This behavior is visualized in Fig. 7, which shows the classic "sawtooth" pattern of the Java Virtual Machine's (JVM) memory management. This pattern is the expected cycle of the garbage collector (GC) at work: memory is allocated to process incoming events (the rising slope), and once a threshold is reached, the GC frees unused memory (the sharp drop). As the second graph illustrates, over a longer period with more resources, this is a healthy and continuous process. The system handles memory efficiently as long as it has sufficient RAM. Going below this at 256MB RAM, we have no overhead above the application framework, so the RAM usage stays around 100%, and performance significantly degrades.

Complexity

This analysis evaluates the system's performance when handling process models of varying complexity, while maintaining a high event rate of 1000 events per second. We defined five scenarios with different numbers and types of constraints, ranging from 10 simple RESPONSE constraints to a mix of 100 forward- and backward-looking constraints.

The results confirm that the engine is highly efficient at managing model complexity. As shown in Fig. 8 and Table 5, CPU and RAM utilization remains stable and low across all complexity scenarios. For example, the average CPU usage for the "100 Mixed Complex" scenario with 10,000 events per second was only 6.12%. This suggests that the primary performance driver is the event rate rather than the number of active rules. RAM usage increases predictably with the number of constraints, but the overall footprint remains manageable.

Figure 9 illustrates that latency remains consistently low and stable, regardless of model complexity. The median latency for all scenarios is approximately 0.2 ms, and even the outliers rarely exceed a few milliseconds. This is a significant finding, as it demonstrates that the synergistic paradigm can execute highly complex declarative models without sacrificing real-time performance.

Table 5 Performance and resource utilization metrics for complexity scenarios

Scenario	Latency (ms)					CPU (%)		Memory (MB)		Actual throughput (events/s)
	μ	P_{50}	P_{95}	P_{99}	Max	Avg.	Max.	Avg.	Max.	
10 Response	0.23	0.17	0.56	1.04	2.13	7.71	33.17	182.00	184.56	998.88
10 Precedence	0.26	0.11	0.90	1.90	8.34	7.87	35.17	179.60	182.03	998.99
50 Response	0.24	0.16	0.63	1.10	1.90	6.52	21.72	249.25	250.68	998.83
40 mixed complex	0.18	0.11	0.57	0.97	1.38	5.84	43.99	261.12	262.40	998.97
100 mixed complex	0.28	0.18	0.84	1.42	2.60	6.12	28.75	455.91	458.36	998.97

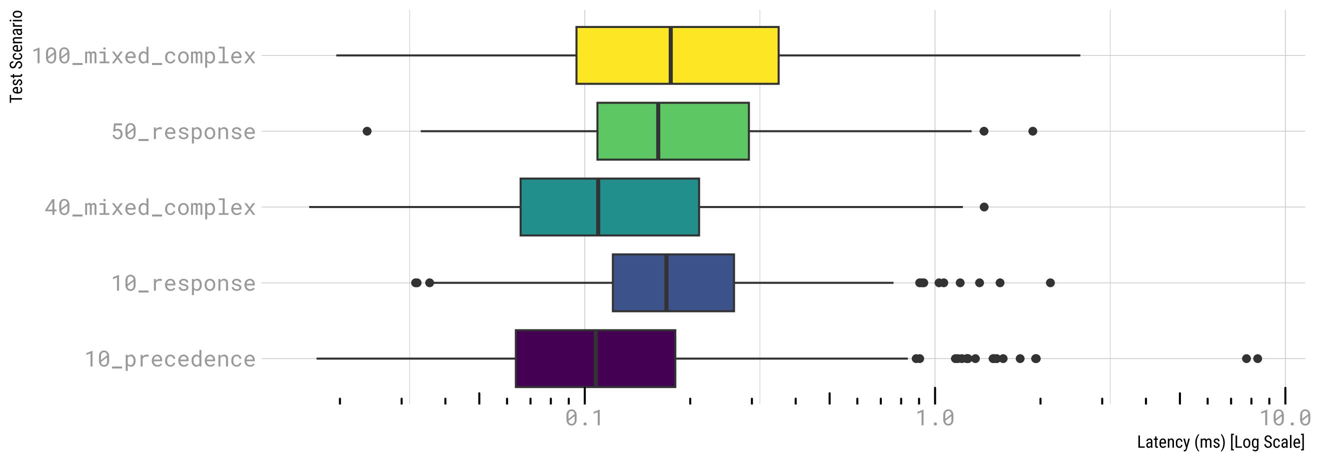


Fig. 9 Distribution of latency at 1000 events per second for different constraint scenarios

Finally, we measured the one-time cost of creating a large number of constraints. Table 6 shows that the memory and time required scale linearly, with the engine capable of sequentially initializing around six constraints per second within the test environment. For practical use, this would need to be executed once at runtime and could be made parallel or optimized in other ways during implementation.

Discussion of results

Our multi-faceted evaluation demonstrates that the proposed synergistic engine paradigm is not only functionally correct but also highly scalable and efficient. The quantitative analysis provides empirical evidence for the architectural benefits of eliminating dedicated middleware. Key findings from our experiments validate this claim:

- **Superior responsiveness**: The system processed up to 10,000 events per second with a 99th percentile latency around one millisecond. This level of performance would be highly challenging in traditional architectures (e.g., [60]) where network hops, data serialization, and inter-process communication for a separate middleware layer would introduce significant, often unpredictable, overhead.
- **Efficient scalability**: Performance is driven primarily by the event rate, not the complexity of the declarative model. The engine efficiently handles models with hundreds of constraints with minimal impact on latency or resource usage, demonstrating its suitability for complex, real-world processes.
- **Practical deployment insights**: The stress test at 100,000 events per second and 1 GB RAM, which triggered frequent garbage collection (Fig. 7), highlights a practical takeaway: while the paradigm is extremely CPU-efficient, proper memory allocation is the critical factor for maintaining performance in high-velocity production environments. This can easily be achieved by scaling computational resources, as the setup used in this evaluation was deliberately limited to very constrained resources (similar to those of a classical single-board computer, such as a Raspberry Pi).

In summary, the architectural simplification shown in both use cases is not just a theoretical benefit; it translates directly into quantifiable performance gains. This answers RQ2 by providing a viable and high-performance runtime execution solution for the proposed model.

Table 6 RAM usage and required time during constraint creation

# Constraints	Memory (MB)	Time (min)
400	584.07	1:14
2000	2007.95	5:48
4000	3673.23	11:25
8000	5771.17	22:52

Discussion and limitations

This research presents a novel approach for the synergistic integration of declarative process specifications, specifically MP-Declare, with CEP through a common engine paradigm. A core contribution is *bridging paradigms* by using a standard CEP engine (Esper) for not only event abstraction but also the direct evaluation and enforcement of declarative constraints derived from potentially high-frequency data streams, such as those originating from IoT environments. This represents a significant departure from prevalent architectures identified in literature (cf. Table 2), which typically rely on distinct middleware layers or separate systems for event preprocessing and process execution, thereby introducing latency and architectural complexity [35, 37, 38, 40]. The unification of these functions within a single engine offers a pathway towards simplified system landscapes and more responsive process execution, directly addressing the abstraction gap noted in [6, 7]. This architectural simplification also enhances system reliability, a key concern in managing complex systems [61], by reducing the number of integration points that could fail.

Our conceptual framework, centered around a multi-level semantical event abstraction (atomic, constraint, and process levels as depicted in Fig. 1), demonstrates significant potency, with *abstraction functioning as an enabler* for structuring the transformation from low-level events to high-level process event insights. The systematic translation of MP-Declare constraints and multi-perspective aspects, including data conditions [57], into executable Esper EPL queries directly addresses our primary research questions regarding CEP-enabled declarative model execution and runtime framework implementation, achieved here through fulfilling targeted design objectives: ❶ supporting flexible process execution, ❷ ensuring continuous process compliance, and ❸ maintaining a clear separation of concerns between IoT information and process-level semantics.

Execution via this mechanism inherently supports the *operational flexibility* demanded by declarative models [8, 13], allowing process flow to emerge from constraints driven by real-time events rather than following prescriptive paths. This work establishes a foundation for reconceptualizing event processing capabilities for declarative process management, particularly within increasingly dynamic and data-rich operational contexts characteristic of IoT environments [3, 56]. Furthermore, this approach manages the inherent complexity of declarative process models. By translating constraints into reactive CEP queries, the engine avoids the state-space explosion that can challenge traditional verifiers, enabling scalable, real-time compliance checking in complex, dynamic environments. Despite promising proof-of-concept results, there are potential threats to validity: Translating MP-Declare constraints into EPL queries can be complex, especially for intricate constraints and models. Moreover, the proof-of-concept may not support all combinations of MP-Declare constraints, particularly for complex temporal or resource-related patterns. Evaluation beyond the conceptual proof [62] of the current implementation, such as performance, is tied to the Esper engine; alternative CEP platforms may require additional adaptations but could lead to further improvements.

The practical benefits of this synergistic paradigm, demonstrated through our *manufacturing* and *logistics* use cases, extend to various other data-intensive domains. By unifying high-frequency event streams with declarative process logic, our approach enables real-time, context-aware decision-making for stakeholders across various industries.

For instance, in *healthcare*, the paradigm could transform real-time patient monitoring. By processing continuous event streams from wearable biosensors, the system could detect complex patterns indicative of a potential medical crisis, such as a specific combination of falling blood oxygen and an irregular heartbeat, and trigger immediate, context-specific alerts. This would directly benefit both patients through improved safety and clinicians by enabling faster and more informed interventions.

Similarly, in *financial fraud detection*, the system could protect against sophisticated fraud in real time. Financial institutions could process high-velocity streams of transaction and user login data. A complex declarative rule could identify a fraudulent pattern, such as a login from a new device followed within seconds by an uncharacteristically large international money transfer, and automatically block the transaction while triggering a multi-factor authentication challenge. This provides immediate protection for customers and reduces financial losses for the institution

A critical consideration for any system processing real-world IoT data is the handling of uncertainty [63]. Rather than being a limitation, our synergistic paradigm is explicitly designed to address this through the inherent flexibility of CEP. We treat the management of noisy, delayed, or ambiguous data as a modeling task that our framework directly supports. The CEP engine's core functionality is to match patterns against event streams over temporal windows [64]. This provides a native mechanism for filtering irrelevant data and correctly processing events that arrive delayed or out of order, as demonstrated in our quantitative evaluation. For ambiguous or potentially misclassified events, the solution lies in creating more sophisticated event detection rules, a task for which CEP is purpose-built. There is no architectural limit to the complexity of this logic. For example, a reliable "Machine Failure" event can be modeled to trigger only after corroborating multiple data streams, such as detecting a temperature spike, a vibration anomaly, and a pressure drop within a 5-second window. This ability to define arbitrarily complex, multi-faceted event signatures is a central strength of our approach, ensuring that high-level process decisions are based on reliable, well-vetted information.

Conclusion and future work

This paper introduces a novel architectural paradigm to address the critical need for systems capable of real-time, context-aware decision-making. By unifying declarative process specifications and CEP within a single engine, we demonstrated the viability of directly executing flexible, constraint-based process models from high-frequency data streams. Our framework, based on multi-level event abstraction, successfully bridges the *abstraction gap* between low-level events and high-level process logic. This synergistic integration simplifies system architectures and reduces reliance on traditional middleware, providing a foundational step towards more agile and responsive business operations.

While our proof-of-concept demonstrates the paradigm's potential on a single-node engine, future research is essential to mature this approach for large-scale enterprise deployment. A primary avenue for future work is to explore the implementation of this paradigm on distributed CEP engines, such as Apache Flink or Spark. This would involve investigating how to partition declarative constraints and event streams across a cluster to achieve horizontal scalability, a basic requirement for processing large volumes of IoT data. Furthermore, the intelligence of the system could be enhanced by incorporating adaptive capabilities for the underlying CEP rules, for instance, by leveraging techniques like reinforcement learning to dynamically update and optimize event patterns as discussed in Mdhaffar et al. [65], or integrating this CEP-based execution core with traditional BPMS functionalities, particularly for human task management, is a next step. This would involve developing mechanisms to present users with available tasks based on the current state of declarative constraints and to ingest task completion events back into the CEP engine, creating a comprehensive and practical system for intelligent process automation.

In conclusion, the synergistic paradigm presented here holds significant potential for advancing the agility and efficiency of business processes in the era of ubiquitous event data. By moving beyond fragmented architectures, this work lays the groundwork for a new class of intelligent systems that can sense, reason, and act in real time, transforming how organizations process event data to drive autonomous, context-aware operations.

Appendix A Implementation details

A.1 Architectural design of the proof-of-concept

Our proof-of-concept is implemented in Java and follows a modular, extensible design pattern for managing declarative constraints. The architecture is centered around three key components: a `BaseConstraintHandler`, specialized handlers for each constraint type, and a reusable `GenericStatusUpdateListener`. This design provides a clear separation of concerns between event detection, constraint logic, and stateful reaction.

The abstract class `BaseConstraintHandler` provides the foundational logic for detecting atomic events (cf. Listing 2). Its primary role is to create EPL queries that lift raw `GenericEvent` objects into a higher-level `constraintStatus` stream, which is used by all other queries. This process involves mapping a generic event type (e.g., "OrderReceived") to a semantically meaningful, constraint-specific event (e.g., an `ACTIVATION` event for the "`ResponseOrderPayment`" constraint).

For each declarative template, a specialized class inheriting from `BaseConstraintHandler` is created (e.g., `AlternatePrecedenceConstraintHandler`). This class is responsible for generating the specific EPL queries that define the fulfillment and violation conditions for that constraint (Listing 3). This modular approach makes the system easily extensible; adding support for a new declarative constraint is as simple as creating a new handler class.

The `GenericStatusUpdateListener` is a foundational, reusable component that decouples the reaction logic from the detection logic. When an EPL query (for fulfillment or violation) matches, the engine invokes this listener (Listing 4). Its responsibilities are to:

1. **Update constraint state:** It calls a central `ConstraintService` to update the status of the constraint (e.g., to `FULFILLED`).
2. **Measure latency:** It calculates the end-to-end latency from the event's ingress timestamp to the moment of detection, which was essential for our quantitative evaluation.
3. **Perform cleanup:** For terminal states, it can instruct the `EsperService` to undeploy the queries associated with that specific constraint instance, ensuring efficient resource management.

Listing 2 Base Handler and Event Detection

```
public void createDetectionQuery(
    StatementType type, String name,
    String event, ConditionRequest
    condition) {
     String query = """
               INSERT INTO
                   constraintStatus
               SELECT id, '%s' as name, '%
                   s' as type, timestamp
               FROM GenericEvent WHERE
                   eventType = '%s'
               """.formatted(name, type,
                   event);

     if (condition.isValid()) {
          query += condition.
              getConditionQueryPart();
     }

     var statement = esperService.
         deployStatements(name, query);
     addConstraintStatement(name,
         statement.getDeploymentId(),
         type, query);
}
```

Listing 3 Specialized Constraint Logic

```
@Override
public void createFulfillmentQuery(
    String name) {
     String query = """
               SELECT b.id, b.name, b.type
                   , b.timestamp
               FROM PATTERN [every a=
                   constraintStatus(type='
                   ACTIVATION', name='%s')
                                -> b=
                                    constraint

                            Status(type='
                                TARGET', name
                                ='%s')]
               """.formatted(name, name);

     var statement = esperService.
         deployStatements(name + "
         _fulfill", query);
     statement.addListener(new
         GenericStatusUpdateListener
         (...));
     // ...
}
```

This architecture enables a robust, maintainable system in which the Java code orchestrates deployment and response, while powerful EPL queries handle the actual real-time pattern-matching logic within the Esper engine.

A.2 Exemplary deployed EPL queries

This section provides an overview of the final EPL queries generated by our system for several foundational declarative templates. The initial `ACTIVATION` and `TARGET` detection queries are standardized and generated as shown in the figure above. The following examples focus on the more complex queries that define the fulfillment and violation logic for each constraint (Listing 5 and 6). Because we are using these templates, we can ensure a structured approach in which all constraint status updates are created in conjunction with the constraint.

To create these detection queries, we use a method that dynamically builds the query based on the event, activation, and target conditions, as well as correlation conditions and the included payload (Listings 7, 8 and 9).

Similar to the activation and target conditions, we also support correlation conditions that describe a correlation between the payload of the activation and the target event. This can be supported by looping through the payload and detecting the correlation on the Constraint Level Events layer, as shown in Listings 10 and 11.

Listing 4 Stateful Reaction and Latency Measurement

```
@Override
public void update(EventBean[]
    newEvents, ... ) {
     // 1. Latency Measurement
     if (event.get("timestamp") != null)
          {
          long startTime = (long) event.
              get("timestamp");
          long latencyNanos = System.
              nanoTime() - startTime;
          log.info("LATENCY,{},{}", this.
              constraintName,
              latencyNanos);
     }

     // 2. Generic Logic for status
         update and cleanup
     constraintService.getConstraints().
         get(constraintName)
          .updateStatus(statusToSet);
     if (removeOnUpdate) {
          esperService.removeConstraint(
              constraintName);
     }
}
```

Listing 5 Exemplary Activation and Target Queries for Declare and MP-Declare

```
-- ACTIVATION
INSERT INTO constraintStatus
SELECT id, 'C1' as name, 'ACTIVATION'
    as type, timestamp
FROM GenericEvent WHERE eventType = 'A'

-- TARGET
INSERT INTO constraintStatus
SELECT id, 'C1' as name, 'TARGET' as
    type, timestamp
FROM GenericEvent WHERE eventType = 'B'

-- ACTIVATION MP-Declare
INSERT INTO constraintStatus SELECT id,
     'C2' as name, 'ACTIVATION' as type
    , timestamp, payload('temp') as
    temp
FROM GenericEvent
WHERE eventType = 'A' AND cast(payload(
    'temp'), double) < 100

-- TARGET MP-Declare
INSERT INTO constraintStatus SELECT id,
     'C2' as name, 'TARGET' as type,
    timestamp, payload('temp') as temp
FROM GenericEvent
WHERE eventType = 'B' AND cast(payload(
    'temp'), double) = 80
```

Listing 6 Dynamic concatenation of conditions

```
     public void createDetectionQuery(
         StatementType type, String name
         , String event,
         ConditionRequest condition,
         CorrelationCondition
         correlation, java.util.Set<
         String> relevantKeys) {
          // ...
          // --- Build the WHERE clause
              from a list of all
              conditions ---
          java.util.List<String>
              conditions = new java.util.
              ArrayList<>();
          // Condition A: The base event
              type
          conditions.add("eventType = '"
              + event + "'");
          // Condition B: The simple
              activation/target condition
          if (condition != null &&
              condition.isValid()) {
               conditions.add(condition.
                   getConditionQueryPart()
                   );
          }
          // Condition C: The correlation
               existence check (for
              activations only)
          if (type == StatementType.
              ACTIVATION && correlation
              != null && correlation.
              isValid()) {
               String existenceCheck = "
                   payload('" +
                   correlation.
                   activationParam() + "')
                    IS NOT NULL";
               conditions.add(
                   existenceCheck);
          }
          // --- 3. Assemble the final
              query ---
          query += " WHERE " + String.
              join(" AND ", conditions);
          // ...
}
```

Listing 7 **RESPONSE(A, B)** This template requires that every occurrence of event A must be eventually followed by an occurrence of event B.

```
-- FULFILLMENT
SELECT b.id, b.name, b.type, b.
    timestamp as timestamp
FROM PATTERN [every a=constraintStatus(
    type='ACTIVATION', name='C3') -> b=
    constraintStatus(type='TARGET',
    name='C3')]

-- TEMPORARY_VIOLATION
SELECT id, name, type, timestamp
FROM constraintStatus
WHERE name = 'C3' AND type = '
    ACTIVATION'
```

Listing 8 **CHAINRESPONSE(A, B)** A stricter version of Response, this template requires that A must be immediately followed by B, with no other events related to this constraint occurring in between.

```
-- FULFILLMENT
SELECT b.id, b.name, b.type, b.
    timestamp as timestamp
FROM PATTERN [every a=constraintStatus(
    type='ACTIVATION', name='C4') -> b=
    constraintStatus(type='TARGET',
    name='C4')]

-- TEMPORARY_VIOLATION
SELECT id, name, type, timestamp
FROM constraintStatus
WHERE name = 'C4' AND type = '
    ACTIVATION'

-- PERMANENT_VIOLATION
SELECT c.id, c.name, c.type, c.
    timestamp as timestamp
FROM PATTERN [every a=constraintStatus(
    type='ACTIVATION', name='C4') -> c=
    constraintStatus(type!='TARGET',
    name!='C4')]
```

Listing 9 **PRECEDENCE(A, B)** This template requires that if event B occurs, event A must have already occurred at some point in the past.

```
-- FULFILLMENT
SELECT b.id, b.name, b.type, b.
    timestamp as timestamp
FROM PATTERN [every a=constraintStatus(
    type='ACTIVATION', name='C5') -> b=
    constraintStatus(type='TARGET',
    name='C5')]

-- TEMPORARY_VIOLATION
SELECT id, name, type, timestamp
FROM constraintStatus
WHERE name = 'C5' AND type = 'TARGET'

-- PERMANENT_VIOLATION
SELECT b.id, b.name, b.type, b.
    timestamp as timestamp
FROM PATTERN [every a=constraintStatus(
    type='TARGET', name='C5') -> (timer
    :interval(1 sec) and not b=
    constraintStatus(type='ACTIVATION',
     name='C5'))]
```

Listing 10 **RESPONSE(A,B) with** $temp_A < temp_B$ $\mathbf{G}(A \rightarrow \mathbf{F}(B \wedge A_{temp} < B_{temp})$

```
-- ACTIVATION
INSERT INTO constraintStatus SELECT id,
     'A' as name, 'ACTIVATION' as type,
     timestamp, payload('temp') as temp
     FROM GenericEvent WHERE eventType
    = 'A' AND payload('temp') IS NOT
    NULL

-- TARGET
INSERT INTO constraintStatus SELECT id,
     'A' as name, 'TARGET' as type,
    timestamp, payload('temp') as temp
    FROM GenericEvent WHERE eventType =
     'B'

-- FULFILLMENT
SELECT b.id, b.name, b.type, b.
    timestamp as timestamp
FROM PATTERN [every a=constraintStatus(
    type='ACTIVATION', name='A') -> b=
    constraintStatus(type='TARGET',
    name='A')]
WHERE a.temp < b.temp
```

Listing 11 **PRECEDENCE(A,B) with** $machineID_A = machineID_B$ $\mathbf{G}(B \rightarrow \mathbf{O}(A \wedge A_{machineID} = B_{machineID})$

```
-- ACTIVATION
INSERT INTO constraintStatus SELECT id,
     'A' as name, 'ACTIVATION' as type,
     timestamp, payload('temp') as temp
FROM GenericEvent
WHERE eventType = 'A' AND payload('temp
    ') IS NOT NULL

-- TARGET
INSERT INTO constraintStatus SELECT id,
     'A' as name, 'TARGET' as type,
    timestamp, payload('temp') as temp
FROM GenericEvent
WHERE eventType = 'B'

-- FULFILLMENT
SELECT b.id, b.name, b.type, b.
    timestamp as timestamp
FROM PATTERN [every a=constraintStatus(
    type='ACTIVATION', name='A') -> b=
    constraintStatus(type='TARGET',
    name='A')]
WHERE cast(a.temp, double) > cast(b.
    temp, double)

-- PERMANENT VIOLATION
SELECT b.id, b.name, b.type, b.
    timestamp as timestamp
FROM PATTERN [every a=constraintStatus(
    type='TARGET', name='A') -> (timer:
    interval(1 sec) and not b=
    constraintStatus(type='ACTIVATION',
     name='A'))]
WHERE cast(a.temp, double) > cast(b.
    temp, double)
```

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 The implementation is linked within the manuscript, additional data are available from the corresponding author upon reasonable request.

Declarations

Conflict of interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

1. Schönig S, Ackermann L, Jablonski S, Ermer A (2020) IoT meets BPM: a bidirectional communication architecture for IoT-aware process execution. Softw Syst Model 19(6):1443–1459. https://doi.org/10.1007/s10270-020-00785-7
2. Rahmani AM, Babaei Z, Souri A (2020) Event-driven IoT architecture for data analysis of reliable healthcare application using complex event processing. Clust Comput 24(2):1347–1360. https://doi.org/10.1007/s10586-020-03189-w
3. Ullah A, Anwar SM, Li J, Nadeem L, Mahmood T, Rehman A, Saba T (2023) Smart cities: the role of internet of things and machine learning in realizing a data-centric smart environment. Complex Intell Syst 10(1):1607–1637. https://doi.org/10.1007/s40747-023-01175-4
4. Etzion O (2011) Event processing in action. Manning, Stamford
5. Laney D (2001) 3D data management: controlling data volume, velocity, and variety. Technical report, META Group. http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf. Accessed on 20th Aug 2025
6. Schönig S, Ackermann L, Jablonski S, Ermer A (2018) An integrated architecture for IoT-aware business process execution. In: Enterprise, business-process and information systems modeling. Springer, Springer, Cham, pp 19–34. https://doi.org/10.1007/978-3-319-91704-7_2
7. Soffer P, Hinze A, Koschmider A, Ziekow H, Ciccio CD, Koldehofe B, Kopp O, Jacobsen A, Sürmeli J, Song W (2019) From event streams to process models and back: challenges and opportunities. Inf Syst 81:181–200. https://doi.org/10.1016/j.is.2017.11.002
8. 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, Annapolis, MD, USA. https://doi.org/10.1109/edoc.2007.14
9. Di Ciccio C, Montali M (2022) Declarative process specifications: reasoning, discovery, monitoring. Springer, Cham, pp 108–152. https://doi.org/10.1007/978-3-031-08848-3_4
10. Dumas M, Rosa ML, Mendling J, Reijers HA (2018) Fundamentals of business process management. Springer, Cham. https://doi.org/10.1007/978-3-662-56509-4
11. Johannesson P, Perjons E (2021) An introduction to design science. Springer, Cham. https://doi.org/10.1007/978-3-030-78132-3
12. Weske M (2019) Business process management, 3rd edn. Springer, Berlin. https://doi.org/10.1007/978-3-662-59432-2
13. Aalst WMP, Pesic M, Schonenberg H (2009) Declarative workflows: balancing between flexibility and support. Comput Sci Res Dev 23(2):99–113. https://doi.org/10.1007/s00450-009-0057-9
14. De Giacomo G, Dumas M, Maggi FM, Montali M (2015) Declarative process modeling in BPMN. Springer, Cham, pp 84–100. https://doi.org/10.1007/978-3-319-19069-3_6
15. Pnueli A (1977) The temporal logic of programs. In: 18th annual symposium on foundations of computer science (sfcs 1977). IEEE, pp 46–57. https://doi.org/10.1109/sfcs.1977.32
16. 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
17. Gershenfeld N, Krikorian R, Cohen D (2004) The internet of things. Sci Am 291(4):76–81. https://doi.org/10.1038/scientificamerican1004-76

18. Kitchin R, McArdle G (2016) What makes big data, big data? Exploring the ontological characteristics of 26 datasets. Big Data Soc. https://doi.org/10.1177/2053951716631130
19. 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
20. Dayal U, Blaustein B, Buchmann A, Chakravarthy U, Hsu M, Ledin R, McCarthy D, Rosenthal A, Sarin S, Carey MJ, Livny M, Jauhari R (1988) The hipac project: combining active databases and timing constraints. ACM SIGMOD Rec 17(1):51–70. https://doi.org/10.1145/44203.44208
21. Buchmann A, Koldehofe B (2009) Complex event processing. itit 51(5):241–242. https://doi.org/10.1524/itit.2009.9058
22. Okoli C (2015) A guide to conducting a standalone systematic literature review. Commun Assoc Inf Syst. https://doi.org/10.17705/1cais.03743
23. Brereton P, Kitchenham BA, Budgen D, Turner M, Khalil M (2007) Lessons from applying the systematic literature review process within the software engineering domain. J Syst Softw 80(4):571–583. https://doi.org/10.1016/j.jss.2006.07.009
24. Page MJ, McKenzie JE, Bossuyt PM, Boutron I, Hoffmann TC, Mulrow CD, Shamseer L, Tetzlaff JM, Akl EA, Brennan SE, Chou R, Glanville J, Grimshaw JM, Hróbjartsson A, Lalu MM, Li T, Loder EW, Mayo-Wilson E, McDonald S, McGuinness LA, Stewart LA, Thomas J, Tricco AC, Welch VA, Whiting P, Moher D (2021) The PRISMA 2020 statement: an updated guideline for reporting systematic reviews. BMJ. https://doi.org/10.1136/bmj.n71
25. Kopp O, Henk S, Karastoyanova D, Khalaf R, Leymann F, Sonntag M, Steinmetz T, Unger T, Wetzstein B (2011) An event model for ws-bpel 2.0. techreport, University of Stuttgart
26. Cicekli NK, Cicekli I (2006) Formalizing the specification and execution of workflows using the event calculus. Inf Sci 176(15):2227–2267. https://doi.org/10.1016/j.ins.2005.10.007
27. Hens P, Snoeck M, Poels G, De Backer M (2014) Process fragmentation, distribution and execution using an event-based interaction scheme. J Syst Softw 89:170–192. https://doi.org/10.1016/j.jss.2013.11.1111
28. Sadoghi M, Jergler M, Jacobsen H-A, Hull R, Vaculin R (2015) Safe distribution and parallel execution of data-centric workflows over the publish/subscribe abstraction. IEEE Trans Knowl Data Eng 27(10):2824–2838. https://doi.org/10.1109/tkde.2015.2421331
29. Soffer P (2013) A state-based intention driven declarative process model. Int J Inf Syst Model Des 4(2):44–64. https://doi.org/10.4018/jismd.2013040103
30. Hildebrandt TT, Mukkamala RR (2011) Declarative event-based workflow as distributed dynamic condition response graphs. Electron Proc Theor Comput Sci 69:59–73. https://doi.org/10.4204/eptcs.69.5
31. Aalst WMP, Adams M, Hofstede AHM, Pesic M, Schonenberg H (2009) Flexibility as a service. Springer, Cham, pp 319–333. https://doi.org/10.1007/978-3-642-04205-8_27
32. Daum M, Götz M, Domaschka J (2012) Integrating cep and bpm: how cep realizes functional requirements of bpm applications (industry article). In: Proceedings of the 6th ACM international conference on distributed event-based systems. DEBS '12. ACM, New York, NY, USA. https://doi.org/10.1145/2335484.2335503
33. Janiesch C, Matzner M, Müller O (2012) Beyond process monitoring: a proof-of-concept of event?driven business activity management. Bus Process Manag J 18(4):625–643. https://doi.org/10.1108/14637151211253765
34. Ammon R, Ertlmaier T, Etzion O, Kofman A, Paulus T (2010) Integrating complex events for collaborating and dynamically changing business processes. Springer, Cham, pp 370–384. https://doi.org/10.1007/978-3-642-16132-2_35
35. Kirikkayis Y, Gallik F, Reichert M (2022) IoTDM4bpmn: an IoT-enhanced decision making framework for BPMN 2.0. In: 2022 international conference on service science (ICSS). IEEE, Zhuhai, China. https://doi.org/10.1109/icss55994.2022.00022
36. Hu J, Wang G, Wang H, Bai W, Li J, Yu J (2023) Improving IoT services through business-process-aligned modeling method. Springer, Singapore, pp 57–71. https://doi.org/10.1007/978-981-99-4402-6_5
37. Valderas P, Torres V, Serral E (2022) Towards an interdisciplinary development of IoT-enhanced business processes. Bus Inf Syst Eng 65(1):25–48. https://doi.org/10.1007/s12599-022-00770-y
38. Panetti, T., D'Ambrogio, A., Bocciarelli, P.: Process over things (pot): an ontology based approach for iot-aware business processes. In: 2021 IEEE 30th international conference on enabling technologies: infrastructure for collaborative enterprises (WETICE). IEEE, Bayonne, France (2021). https://doi.org/10.1109/wetice53228.2021.00029
39. Wehlitz R, Rößner I, Franczyk B (2018) Integrating smart devices as business process resources—concept and software prototype. Springer, Cham, pp 252–257. https://doi.org/10.1007/978-3-319-91764-1_20
40. Seiger R, Malburg L, Weber B, Bergmann R (2022) Integrating process management and event processing in smart factories: a systems architecture and use cases. J Manuf Syst 63:575–592. https://doi.org/10.1016/j.jmsy.2022.05.012
41. Malburg L, Seiger R, Bergmann R, Weber B (2020) Using physical factory simulation models for business process management research. Springer, Cham, pp 95–107. https://doi.org/10.1007/978-3-030-66498-5_8
42. Stoiber C, Schönig S (2021) Event-driven business process management enhancing IoT—a systematic literature review and development of research agenda. Springer, Cham, pp 645–661. https://doi.org/10.1007/978-3-030-86800-0_44
43. Janiesch C, Matzner M (2019) Bamn: a modeling method for business activity monitoring systems. J Decis Syst 28(3):185–223. https://doi.org/10.1080/12460125.2019.1631682
44. Mandal S, Hewelt M, Weske M (2017) A framework for integrating real-world events and business processes in an IoT environment. Springer, Cham, pp 194–212. https://doi.org/10.1007/978-3-319-69462-7_13
45. Beyer J, Kuhn P, Hewelt M, Mandal S, Weske M (2016) Unicorn meets chimera: integrating external events into case management. In: Proceedings of the BPM demo track co-located with the 14th international conference on business process management 2016, pp 67–72
46. He S, Wang H, Cao Y, Zhao D (2019) A wide-deep event model for complex event processing in edge and cloud computing environment. In: Proceedings of the ACM Turing celebration conference - China. ACM, New York, NY, USA, pp 1–2. https://doi.org/10.1145/3321408.3321608
47. Decker G, Grosskopf A, Barros A (2007) A graphical notation for modeling complex events in business processes. In: 11th IEEE international enterprise distributed object computing conference (EDOC 2007). IEEE, Annapolis, MD, USA. https://doi.org/10.1109/edoc.2007.41
48. Krumeich J, Jacobi S, Werth D, Loos P (2014) Towards planning and control of business processes based on event-based predictions. Springer, Cham, pp 38–49. https://doi.org/10.1007/978-3-319-06695-0_4
49. Webster J, Watson RT (2002) Analyzing the past to prepare for the future: Writing a literature review. MISQ 26(2):xiii–xxiii
50. Yang R, Wu M, Gui X, Chen H (2024) Intelligent conflict detection of iot services using high-level petri nets. Complex Intell Syst 10(3):3789–3817. https://doi.org/10.1007/s40747-024-01349-8

51. Venable J, Pries-Heje J, Baskerville R (2016) FEDS: a framework for evaluation in design science research. Eur J Inf Syst 25(1):77–89. https://doi.org/10.1057/ejis.2014.36
52. Hevner M, Park J, Ram S (2004) Design science in information systems research. MIS Q 28(1):75–105. https://doi.org/10.2307/25148625
53. March ST, Smith GF (1995) Design and natural science research on information technology. Decis Support Syst 15(4):251–266. https://doi.org/10.1016/0167-9236(94)00041-2
54. Peffers K, Tuunanen T, Rothenberger MA, Chatterjee S (2007) A design science research methodology for information systems research. J Manag Inf Syst 24(3):45–77. https://doi.org/10.2753/mis0742-1222240302
55. Gregor S, Hevner AR (2013) Positioning and presenting design science research for maximum impact. MIS Q 37(2):337–355. https://doi.org/10.25300/misq/2013/37.2.01
56. 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
57. Burattin A, Maggi FM, Aalst WMP, Sperduti A (2012) Techniques for a posteriori analysis of declarative processes. In: 2012 IEEE 16th international enterprise distributed object computing conference. IEEE, Beijing, China. https://doi.org/10.1109/edoc.2012.15
58. Ackermann L, Schönig S, Petter S, Schützenmeier N, Jablonski S (2018) Execution of multi-perspective declarative process models. Springer, Berlin, pp 154–172. https://doi.org/10.1007/978-3-030-02671-4_9
59. Poss L, Schönig S (2024) Location-aware business process modeling and execution. Softw Syst Model 24(1):37–67. https://doi.org/10.1007/s10270-024-01224-7
60. Poss L, Dietz L, Schönig S (2023) Labpmn: location-aware business process modeling and notation. In: Sellami M, Vidal M-E, Dongen B, Gaaloul W, Panetto H (eds) Proceedings of the international conference on cooperative information systems (CoopIS) 2023. https://doi.org/10.1007/978-3-031-46846-9_11
61. Kaliszewski N, Marian R, Chahl J (2025) A reliability centred maintenance-oriented framework for modelling, evaluating, and optimising complex repairable flow networks. Complex Intell Syst. https://doi.org/10.1007/s40747-025-01787-y
62. Hevner AR (2007) A three cycle view of design science research. Scand J Inf Syst 19(2):87–92
63. Yazdi M, Zarei E, Adumene S, Abbassi R, Rahnamayiezekavat P (2022) Uncertainty modeling in risk assessment of digitalized process systems. Elsevier, Cambridge, pp 389–416. https://doi.org/10.1016/bs.mcps.2022.04.005
64. Cugola G, Margara A (2015) The complex event processing paradigm. Springer, Cham, pp 113–133. https://doi.org/10.1007/978-3-319-20062-0_6
65. Mdhaffar A, Baklouti G, Rebai Y, Jmaiel M, Freisleben B (2025) Rl4cep: reinforcement learning for updating cep rules. Complex Intell Syst. https://doi.org/10.1007/s40747-024-01742-3