Linköping University | Department of Computer and Information Science
Master’s thesis, 30 ECTS | Datateknik
2025 | LIU-IDA/LITH-EX-A--25/018--SE

Improving Event Aggregation in Automation of Software Development Workflows

Johan Christiansson
Lukas Ekberg

Supervisor : Máté Földiák
Examiner : Dániel Varró

External supervisor : Mattias Linnér and Emil Bäckmark

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Abstract

Modern CI/CD systems produce vast streams of events that reflect the state and progression of software development processes. Aggregating these events is essential for enabling automation and traceability. In this context, *aggregation* refers to the process of detecting and grouping sequences of events into higher-level constructs based on predefined rules. This thesis investigates alternative mechanisms for event aggregation, with a focus on graph-based approaches within the Eiffel framework.

By representing events and their relationships as a property graph, our proposed system enables declarative specification of patterns and supports incremental detection using trigger-based mechanisms in graph databases. The work implements and evaluates multiple aggregation systems, including graph databases and a complex event processing engine, and compares them based on throughput and resource efficiency under high workloads, using historical Eiffel data.

The evaluation includes three different potential event aggregation systems: Neo4j, Memgraph, and Apache Flink. The results show that both Memgraph and Flink exhibit significant limitations in terms of performance and functionality for the use case, and are ultimately deemed unsuitable. Neo4j, on the other hand, provide a more promising foundation for implementing graph-based pattern matching. The findings show that Neo4j enables flexible event aggregation whilst simplifying rule definition. However, performance degrades significantly when the number of active triggers increase. Addressing this scalability limitation remains an open challenge.

Acknowledgments

We would like to express our sincere gratitude to everyone who supported us and contributed to this thesis.

Our university supervisor, Máté Földiák, consistently provided valuable feedback and thoughtful guidance, which greatly contributed to the development of this thesis. We also thank our examiner, Dániel Varró, for his insightful comments and support in shaping the direction of the project.

Lastly, we would like to thank our supervisors at Ericsson, Mattias Linnér and Emil Bäckmark, for their technical advice, constructive feedback, and continuous support during the thesis. Their engagement and willingness to provide guidance greatly contributed to the quality of this thesis and made the process both productive and enjoyable.

Contents

List of Figures

List of Tables

1 Introduction

In today's fast-paced world of software development, delivering high-quality solutions quickly and efficiently has become a cornerstone of success. Behind the scenes, systems work tirelessly to organize the many processes required to bring software from concept to reality. One of these is the *continuous integration and continuous delivery* (CI/CD) pipeline. However, as these systems grow larger and more complex, understanding the vast amounts of information they produce becomes increasingly difficult, making it challenging to monitor, maintain traceability, and detect meaningful sequences of occurrences across processes.

To tackle these challenges, new approaches are required to ensure simplicity, traceability, and efficiency in managing these systems. This introduction highlights the importance of addressing these challenges and outlines the research questions and aims that this work seeks to answer.

1.1 Motivation

CI/CD is an integral practice within modern software engineering, enabling teams to deliver high-quality software quickly and efficiently [7]. In large-scale CI/CD systems, events can serve as the backbone for communication, providing traceability and meaningful insights into the state of various processes such as builds, tests, and deployments.

To accommodate event-based communication, Ericsson has created the open source *Eiffel Event Protocol*, in which each occurrence, or statement of fact, in a CI/CD system is represented as an event [24]. The Eiffel Event Protocol describes each event and the connection between the different events [81]. Typical events include that a build execution is finished or that automated regression tests have been completed [81]. A connection between the events might then be that a certain test case event links back to an event of its corresponding test suite [81]. As the events are posted in a stream, it is possible for consumers (listeners) to analyze and understand what has occurred [81].

Aggregation plays a vital role in event-driven systems, where large volumes of events are sent continuously in an event stream. Aggregations are sets of rules checked against the events that define how interconnected events should be grouped together [24]. By collecting and organizing events into a structured format, aggregation provides a simplified view over occurrences. Each event only contains the necessary information, meaning that data from a sequence of events need to be investigated in combination to make sense of high-level properties. By monitoring event streams to identify meaningful patterns, aggregations can serve as trigger points within CI/CD pipelines. When an aggregation is done, a notification can be sent. These notifications enable automated responses, such as initiating further test stages. As a result, aggregations become not just passive observers of event data but active components in orchestrating complex delivery workflows.

The component of the Eiffel landscape that handles aggregations is called *Eiffel Intelligence* [24], and in its current state it is very difficult to write aggregation rules, resulting in reduced adoption in practice. A secondary problem with Eiffel Intelligence is that it is unable to handle multiple aggregation rules within a single instance, requiring a separate instance to run for each active aggregation rule. These problems result in the creation and inclusion of new aggregation rules to be costly, as it is both time-consuming and resource-intensive to maintain.

1.2 Aim

This thesis aims to propose and evaluate novel event aggregation mechanisms for CI/CD pipelines within the Eiffel framework. By incorporating techniques from *complex event processing* (CEP) and graph-based approaches, the study seeks to simplify event handling, improve system usability, and enhance performance, particularly in detecting patterns across streams. To fulfill this aim, the following objectives are established:

- Ob1. Enable declarative specification of event aggregation patterns using expressive graph query constructs.
- Ob2. Implement a prototype system for empirical evaluation and validate it against realistic CI/CD scenarios using historical Eiffel event data.
- Ob3. Provide a comparative evaluation of alternative aggregation techniques under varying event loads with respect to throughput and resource efficiency.

1.3 Research Questions

In order to address the objectives presented in Section 1.2, this thesis defines the following research questions:

- (RQ1) What technologies can aggregate events while maintaining sufficient performance and efficient resource usage over time?
- (RQ2) How does the number of active patterns affect throughput on the evaluated technologies?

1.4 Approach

To answer the research questions, first a literature study was conducted to find existing tools that are used for aggregation in event streams. This was followed by an evaluation of tools in a small case study to analyze the most prominent alternatives. Based on the preliminary evaluation, the promising alternatives were further iterated to improve their performance, and thereafter tested against the current solution of Eiffel Intelligence. The tests included streaming data into the tools and evaluating how well the alternatives could process different aggregation rules of varying complexity.

1.5 Contributions

The contributions in this project align with and provide answers to the research questions from Section 1.3. The contributions presented in this thesis are as follows:

Co1. **Architecture proposal for event aggregation:** We propose an architectural design that enables scalable event aggregation in CI/CD pipelines. Our design reduces the need for multiple instances and allows multiple aggregation rules to coexist efficiently.

Co2. **Comparative evaluation of graph-based and CEP approaches:** We implemented and evaluated three distinct aggregation systems. This provides a comprehensive comparison between graph databases and CEP engines for event aggregation in terms of throughput and resource usage.

Co3. **Declarative graph-based representation of event patterns:** We demonstrated methods for defining event aggregation patterns declaratively using graph queries. By treating events and their interrelations as a graph, our approach enables precise and flexible specification of aggregation logic.

Co4. **Trigger-based incremental pattern matching in graph databases:** We designed and implemented custom triggers for graph databases to enable incremental graph pattern matching, which is further explained in Section 2.7.5. These allow patterns to be detected reactively as events are inserted, offering a novel mechanism for CI/CD automation.

1.6 Delimitations

Only a defined set of realistic event patterns, inspired by the commonly used ones in Eiffel Intelligence, are evaluated. More advanced or custom-designed patterns beyond this scope are not considered for evaluation.

All data used in this work originates from industrial CI/CD pipelines. This domain-specific focus limits the generalizability of the approach to other potential use cases.

Furthermore, the system is evaluated in a controlled environment using historical Eiffel data. It is not deployed in a production-like infrastructure such as a Kubernetes cluster. As a result, the study does not account for operational factors such as distributed deployment challenges, network latencies, fault tolerance, or real-time system integration.

Finally, no usability study or user evaluation is conducted as part of this work. While such a study could provide valuable insights into the practical applicability and user experience of the system, it falls outside the scope of this thesis.

1.7 Thesis Outline

This thesis is structured into the following chapters to provide a clear and coherent reading experience. Chapter 2 introduces the background and outlines the theoretical foundations relevant to the study. Chapter 3 situates the thesis within the context of existing research in the field. The methodology and design choices used to address the research questions are detailed in Chapter 4. Chapter 5 presents the results of the experiments, which are then analyzed and discussed in Chapter 6. Finally, Chapter 7 summarizes the key findings and offers suggestions for future research.

2 Background

This chapter provides the theoretical and technical background necessary to understand the problem of event aggregation in CI/CD pipelines. It begins with an overview of CI/CD systems and the role of event-based communication for traceability and automation in Section 2.1. In Section 2.2, the Eiffel protocol is then introduced as a structured approach to representing events and their relations. The chapter continues by discussing the foundation of event aggregation and its importance in orchestrating higher-level workflows in Section 2.3. Thereafter Eiffel Intelligence is introduced as technology for event aggregation in Section 2.4. To support the proposed solutions, the chapter introduces foundations of complex event processing in Section 2.5, and then technology of complex event processing in Section 2.6. Then, foundations of graph databases is introduced in Section 2.7, providing background to concepts such as graphs and pattern matching. Finally, graph database technologies are explained in Section 2.8, focusing on graph query languages and graph databases.

2.1 Foundation of CI/CD

Agile software development has transformed the way software projects are managed, emphasizing iterative workflows, collaboration, and responsiveness to change [7]. Central to agile methodologies is the focus on quick delivery and continuous improvement, made possible by continuous integration (CI) and continuous delivery (CD) practices [7]. CI/CD pipelines automate tasks such as code integration, testing, and deployment, which helps reduce the engineering time spent and human error associated with manual processes [7]. These practices align with agile principles by enabling frequent and reliable delivery of software while improving team productivity [7].

Traceability refers to the ability to link and track the relationships between various software artifacts throughout the development lifecycle [23]. This includes connections between requirements, design documents, source code, test cases, builds, and deployment outcomes [23, 81]. Traceability is a critical concept in software engineering, particularly for large-scale systems for its ability to link various artifacts to ensure efficient development processes, transparency and compliance [81]. In the context of CI/CD, traceability becomes even more significant due to the high frequency of changes and the need for rapid feedback cycles [81].

Traditional software development methods often struggle with traceability in large-scale CI/CD environments [81]. Factors such as scale, diversity of tools and technologies, and the rapid nature of agile methodologies amplify the challenge of traceability [81]. Without effective traceability, organizations face difficulties in tracking dependencies, understanding system states, and meeting compliance requirements [81].

To address these issues, frameworks like the Eiffel Protocol have been developed [81]. Designed for large-scale CI/CD environments, the Eiffel Protocol introduces an event-driven approach to traceability [81]. It generates real-time trace links and organizes them into a *directed acyclic graph* (see Section 2.7.2) structure, enabling efficient navigation through artifact relationships [24, 81].

2.2 Technology of CI/CD

While CI/CD systems are conceptually centered around automation and traceability, their implementation relies on specific technologies and protocols that manage the flow and structure of development data. Solutions like Jenkins [80], GitHub Actions [53], GitLab CI [38], and Travis CI [89] are commonly used to manage code integration, testing, and deployment processes. However, these tools often operate independently, without a shared standard for modeling event relationships in the software delivery process. This section introduces the Eiffel protocol, which provides a structured framework for modeling events, capturing dependencies, and enabling traceability in software delivery pipelines.

2.2.1 Eiffel

The Eiffel Protocol is an open-source protocol for managing and tracing events in CI/CD pipelines [24]. Its core purpose is to enable visibility and traceability as well as automating CI/CD processes in systems by modeling build, test, and deployment processes as a series of interconnected events [24, 81].

In the context of this thesis, Eiffel provides a structured data model that enables analysis and detection of patterns in CI/CD workflows. Its design makes it particularly suitable for graph-based modeling, pattern matching, and incremental event processing, as each new event may contribute to a new subgraph that fulfills a meaningful pattern.

In Eiffel, each event represents a discrete, immutable unit of activity within the software delivery lifecycle, such as artifact creation, test execution, or deployment [24, 81]. Conceptually, an event can be thought of as an object with a defined type, a unique identifier, and a set of attributes, like timestamps and metadata. Crucially, events also contain *links* to other events, describing relationships such as causality, dependency, or context [24]. These links can be seen as labeled references, connecting one event to another, much like edges in a graph. This structure naturally lends itself to a graph-oriented interpretation, where events are nodes with attributes, and links become typed, directional edges. Since each new event links only to previously created events, the resulting graph is a directed acyclic graph (DAG), described in Section 2.7.2.

The feature of explicit linking between events is achieved via unique identifiers, where each event may contain references to previous events using semantic relationships [24]. In Figure 2.1 exists a simplified event consisting of `meta`, `data` and `links` fields.

2.3 Foundation of Event Aggregation

In event-driven systems, individual events often carry only necessary information, making it difficult to understand broader happenings from isolated occurrences [83, 24, 27]. Aggregation is the process of evaluating rules that determine how a sequence of events should be combined into a meaningful unit. These rules are referred to as patterns, and they specify the conditions under which multiple related events form a logical unit [1].

```
{
  // Eiffel specific attributes
  "meta": {
    "type": "EiffelArtifactCreatedEvent",
    "id": "497dcba3-ecbf-4587-a2dd-5eb0665e6880",
    "time": 1746617672277
    // ...
  },
  // Producer specific attributes
  "data": {
    "identity": "pkg:maven/.../artifact-name@2.1.7"
    // ...
  },
  // Links to other events
  "links": [
    { "type": "CAUSE", "target": "50e14f43-dd4e-412f-864d-78943ea28d91" },
    { "type": "PREVIOUS_VERSION", "target": "7edb3b2e-869c-485b-af70-76a934e0fcfd" },
    { "type": "COMPOSITION", "target": "50e14f43-dd4e-412f-864d-78943ea28d91" },
    { "type": "ENVIRONMENT", "target": "67e32b59-3348-4dc3-9645-75c60b6f50cc" }
  ]
}
```

Figure 2.1: Example of an Eiffel event in JSON format

Patterns enable systems to reason about complex event relationships and facilitate downstream analysis or decision-making based on structured representations of related events. When an aggregation pattern is successfully matched, one or more consumers (e.g., services, monitoring tools, or pipeline components) can be notified and provided with a combined representation of the matched events. This enables downstream systems to react to complex situations that span across multiple individual events [75].

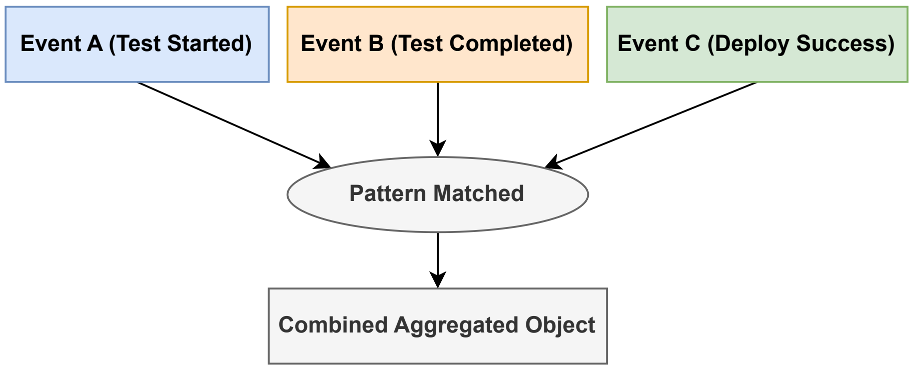


Figure 2.2: Event aggregation of event pattern

The key purpose of aggregation is the ability to detect higher-level occurrences from low-level event streams. In essence, it allows the system to determine whether a particular situation or condition has occurred [75]. This is essential for automation, traceability, and ensuring that subsequent actions in a CI/CD pipeline can be triggered appropriately.

The aggregation rules themselves are defined by the user based on their specific interests or needs. By specifying which types of events and their interconnections, users can create patterns that reflect important sequences or dependencies in the system.

2.4 Technology of Event Aggregation

Event aggregation is a central mechanism in CI/CD pipelines that enables systems to interpret sequences of events as higher-level constructs. Within the Eiffel framework, the primary component responsible for this is Eiffel Intelligence, which receives, interprets, and aggregates events based on user-defined rules [24].

2.4.1 Eiffel Intelligence

Eiffel has a collection of components that together form the Eiffel ecosystem. One of these components is *Eiffel Intelligence* [24], which serves as an aggregator component within the system. Its core purpose is to receive and interpret Eiffel events, evaluate predefined conditions or rules, and provide the possibility to send notifications when certain patterns are detected. This enables higher-level reasoning over the event flow and allows for automation within CI/CD pipelines [24].

Architecture overview

When a new Eiffel event is created, it is published to a message bus, to which multiple services can subscribe. One of these services is the *event repository*, which is a database that stores all events to be accessed later, and another service is Eiffel Intelligence.

Eiffel Intelligence processes newly created events to determine whether they match a predefined pattern. If a match is found, the corresponding consumer is notified. This process involves two main steps: first, creating an *aggregated object* based on incoming events; and second, checking whether this object contains information relevant to a consumer, as defined in their *subscription* [24].

Aggregation rules

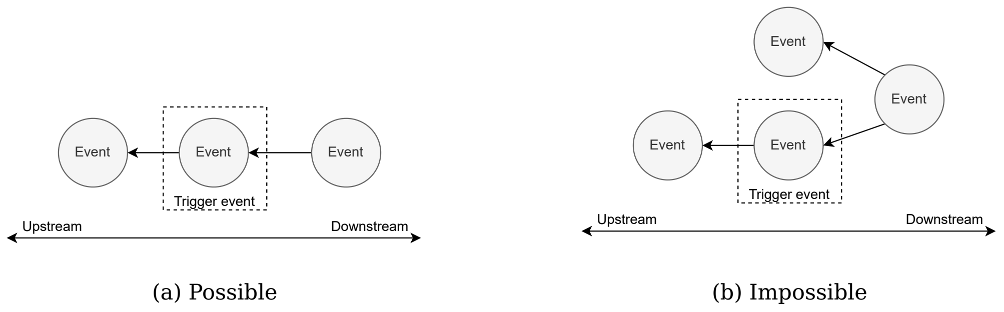
How a specific aggregated object is built is defined by *aggregation rules*, written using JMESPath, and can look like Figure 2.3 [24]. Each aggregation rule requires a separate instance of Eiffel Intelligence, which both complicates deployment and makes it more expensive. Additionally, writing new aggregation rules is complex and time-consuming. To start aggregation, each pattern starts with a *trigger event* (or *trigger node*), which is a specific event type specified by the user [24]. This event signals Eiffel Intelligence to start the aggregation process [24]. The trigger event is selected based on its suitability to initiate the pattern, but in practice, it can be any event type included within the pattern.

```
{
"TemplateName":"ARTIFACT_1",
"Type":"EiffelArtifactCreatedEvent",
"TypeRule": "meta.type",
"IdRule": "meta.id",
"StartEvent": "YES",
"IdentifyRules" : "[meta.id]",
"MatchIdRules": {"_id": "%IdentifyRulesEventId%"},
"ExtractionRules" : "{ id : meta.id, type : meta.type, time : meta.time, name : data.name, identity :
    data.identity, fileInformation : data.fileInformation, buildCommand : data.buildCommand,
    artifactCustomData : data.customData }",
"DownstreamIdentifyRules" : "links | [?type=='COMPOSITION'].target",
"DownstreamMergeRules": "{\"externalComposition\":{\"eventId\":%IdentifyRulesEventId%}}",
"DownstreamExtractionRules" : "{artifacts: [{id : meta.id}]}",
"HistoryIdentifyRules":"links | [?type=='COMPOSITION'].target",
"HistoryExtractionRules":"{id : meta.id, identity : data.identity, fileInformation : data.fileInformation}",
"HistoryPathRules": "{artifacts: [{id: meta.id}]}",
"ProcessRules" : null
},
```

Figure 2.3: Aggregation rule in Eiffel Intelligence

If the trigger event does not fulfill the entire pattern, meaning it requires events that have not yet arrived, then Eiffel Intelligence will wait for those remaining events [24]. Upon receiving the trigger event, it queries the event repository to retrieve any existing interconnected events that may satisfy parts of the pattern and monitors for additional incoming events as necessary to complete the aggregation [24]. If the pattern is satisfied, data from all events are combined into a new aggregated object (or *complex event*) based on the user's aggregation rule specification [24].

Aggregation rules are limited in the types of patterns they can define. Specifically, there exist two directions Eiffel Intelligence can search events in: backwards (upstream), meaning searching previously received events, and forwards (downstream), involving events that have yet to arrive. Eiffel Intelligence can with no problem search both downstream and upstream from a certain trigger event, shown in Figure 2.4a. Problematic patterns occur when trying to first search in one direction and then in the other direction from found events, shown in Figure 2.4b. While these limitations could be addressed, doing so would be a challenge to do in Eiffel Intelligence, furthermore, it would decrease the usability further as the aggregation rules would most likely grow increasingly complex.



(a) Possible (b) Impossible

Figure 2.4: Possible and impossible search in Eiffel Intelligence

Subscriptions

The next step is the subscriptions that users can create to filter which aggregated objects they want to be notified. The subscription consists of three main parts [24]:

1. **Filter**: The filter allows filtering on specific fields; for example, a specific product version on the aggregated objects matching a specific pattern.
2. **Endpoint**: The endpoint or email to which information should be sent.
3. **Information**: The specific contents of the aggregated object that need to be included in the notification.

2.5 Foundation of Complex Event Processing

Complex event processing (CEP) is a technique for analyzing continuous streams of data in real time to detect meaningful patterns and relationships between events [75, 60]. Unlike most traditional data processing approaches that operate on static data [19], CEP focus on dynamic, high-throughput inputs. At its core, CEP defines patterns using rules or queries that specify how primitive events are correlated and combined into complex events [75]. For example, a sudden drop in stock price followed by a surge in trading volume could be defined as a pattern indicating potential market manipulation. By identifying specific sequences of events as they occur, CEP enables real-time feedback, alerts, automated responses, or further analysis, which is critical in many domains [75, 39].

In complex event processing, a primitive event is an individual data point that includes a timestamp, which determines the event’s position in time and defines its order relative to other events [96, 27]. Primitive events include actions such as a code commit or the completion of a test [96].On their own, primitive events may give limited information, but when multiple such events are combined based on ordering or content, they can reveal patterns that point to more meaningful behaviour [75, 26, 25]. These combinations are known as complex events (CE) [55, 27]. Complex events simplify streams of events by highlighting meaningful situations, allowing the system to respond based on context rather than isolated data points [26].

CEP is commonly used in domains where finding meaningful patterns depends on analyzing the relationships among sequences of events [55].

2.6 Technology of Complex Event Processing

The practical execution of complex event processing relies on a set of technologies designed to handle continuous, high-volume event streams efficiently. At the center is the CEP engine, which enables real-time evaluation of event patterns by maintaining system state, applying temporal logic, and managing event flow [25].

The incoming primitive events from the event stream are continuously evaluated against all defined patterns using a *CEP engine* [27]. The CEP engine serves as the core component responsible for real-time event matching. It manages the internal state needed to detect partial and complete matches of patterns over time, handles event ordering and timing constraints, while ensuring efficient processing at scale [35]. When a defined pattern is satisfied, the engine triggers corresponding actions, such as emitting complex events, raising alerts, or invoking external services [52]. To maintain performance and reliability, modern CEP engines often incorporate features like time windowing, out-of-order event handling, and checkpoint-based fault tolerance [6, 30].

2.6.1 Flink

Apache Flink is a leading data-streaming framework used extensively in both industry and research communities [51].

Flink supports a wide range of applications, such as real-time analytics, historical processing, and iterative algorithms. [16]. At its core, Flink is a streaming engine that models applications as directed graphs of data transformations, where each node in the graph represents a computational operator, and the edges represent the flow of data between them. These operators continuously process incoming events as part of a stream, enabling low-latency, parallel execution across distributed environments. Flink's versatile APIs and windowing mechanisms enable native event-time processing, robust handling of out-of-order events, and guarantee that each event is processed exactly once [16].

In addition to being an efficient data streaming tool, Flink has a number of libraries that extend the core functionality to enable machine learning, graph analysis, and complex event processing [16, 55]. FlinkCEP is Flink's dedicated library for CEP [6, 16].

FlinkCEP allows users to define event patterns declaratively using a specialized API. These patterns may include sequences, branching conditions, repetitions, or negations, and can be further specified using contiguity conditions, consumption policies, and windowing strategies [6, 55]. Contiguity conditions specify how events in a pattern must occur in relation to each other, consumption policies define how detected patterns are handled after a match, and windowing strategies determine the boundaries within which a pattern is recognized. These mechanisms enable robust and precise pattern detection across data streams [6, 55].

However, while FlinkCEP provides powerful capabilities, it comes with a steep learning curve. Writing and maintaining pattern definitions programmatically requires significant expertise and can be cumbersome for non-technical users or rapid prototyping [55].

2.7 Foundation of Graph Databases

This section introduces the foundational concepts necessary to understand the concept of graph databases. It begins by describing how data can be structured as a graph, followed by defining property graphs, which enable attribute-based modeling of entities and their relationships. Then pattern matching techniques used to identify meaningful substructures in graphs is introduced, and concludes with an overview of incremental graph pattern matching, which is an essential concept for efficient detection of event patterns in dynamic data streams.

2.7.1 Building a graph

Graph theory is a study comprising *vertices* (also referred to as nodes) and *edges* (or links) [99]. In the context of software systems, nodes often represent entities (e.g., events, artifacts), while edges capture relations or dependencies between those entities [48, 36].

Definition 1 (Adapted from [99])**.** *A graph $\mathcal{G}$ is a pair $\langle V, E\rangle$, where V is the set of vertices and E is the set of edges connecting vertices.*

Figure 2.5 provides a simple visual example of a directed graph. It illustrates two vertices, v_1 and v_2, and a directed edge (v_1, v_2), representing a relationship from v_1 to v_2. This corresponds to the formal definition of a graph in Definition 1, where $V = \{v_1, v_2\}$ and $E = \{(v_1, v_2)\}$. In a software context, this could represent an event causing another.

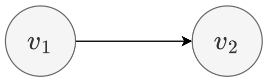


Figure 2.5: Illustration of a simple directed graph

2.7.2 Directed acyclic graph

A directed acyclic graph is a graph consisting of nodes connected by only directed edges, and no sequence of edges forms a cycle [41]. In other words, it is impossible to start at any node and follow a consistent direction of edges that leads back to the same node.

2.7.3 Property graphs

While basic graphs model relationships between entities using only nodes and edges, many real-world applications require additional context to be captured directly within the graph [82]. A *property graph* extends the traditional graph model by allowing both nodes and edges to store metadata in the form of key-value pairs, known as *properties*, and to carry one or more descriptive *labels* [3]. In this model:

- **Nodes** represent entities or objects and may have one or more labels (e.g., `Event`, `Artifact`) indicating their type or role. Nodes also carry properties such as `id`, `name`, or `timestamp` that provide contextual or identifying information.
- **Edges** represent directed relationships between nodes and have one or more labels (e.g., `CAUSE`, `PREVIOUS_VERSION`) describing the type of relationship. Like nodes, edges may also have associated properties, such as creation time or version number.

This enriched graph structure enables expressive modeling of complex, semantically rich data [3]. It supports querying and pattern matching (see Section 2.7.4), not only based on the topology of the graph, but also on the content in the labels and properties [3]. To formally define property graphs, the definition provided by Angles [3] is followed.

Definition 2 (Adapted from [3])**.** *A property graph is a tuple $\mathcal{G} = \langle V, E, \rho, \lambda, \sigma\rangle$ where:*

1. *V is a finite set of nodes;*
2. *E is a finite set of edges such that E has no elements in common with V;*
3. *$\rho : E \rightarrow (V \times V)$ is a total function that link each edge in E with a pair of nodes in V;*
4. *$\lambda : (V \cup E) \rightarrow SET^{+}(\mathbf{L})$ is a partial function that associates a node/edge with a set of labels from $\mathbf{L}$, where $\mathbf{L}$ is an infinite set of labels (for nodes and edges) and given a set X, then $SET^{+}(X)$ is the set of all finite subsets of X, excluding the empty set;*
5. *$\sigma : (V \cup E) \times \boldsymbol{P} \rightarrow SET^{+}(\boldsymbol{I})$ is a partial function that associates nodes/edges with properties, and for each property it assigns a set of values from $\boldsymbol{I}$, where $\boldsymbol{I}$ is an infinite set of atomic values and $\boldsymbol{P}$ is an infinite set of property names.*

2.7.4 Pattern matching

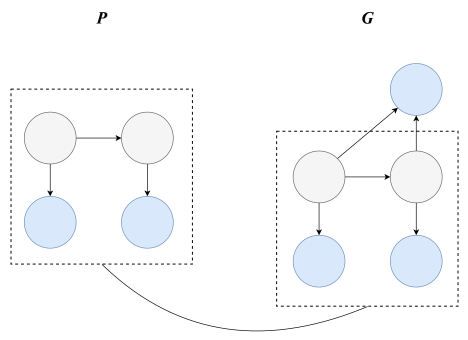
In graph theory, pattern matching refers to the process of identifying parts of a graph that match a specified structure, known as a *pattern graph* [14, 33, 9]. This pattern is typically a smaller graph that expresses a particular arrangement of nodes and edges that is meaningful within a larger data graph. By extending the graph structure to have labels, as defined in Definition 2, occurances of a labeled pattern graph can be found in a labeled data graph through the following model:

Definition 3 (Adapted from [14])**.** *Given a pattern graph $\mathcal{P} = \langle V_\mathcal{P}, E_\mathcal{P}, \lambda_\mathcal{P} \rangle$ and a data graph $\mathcal{G} = \langle V_\mathcal{G}, E_\mathcal{G}, \lambda_\mathcal{G} \rangle$, a pattern match is a function $\varphi : V_\mathcal{P} \to V_\mathcal{G}$ mapping the nodes of $\mathcal{P}$ to the nodes in $\mathcal{G}$ such that:*

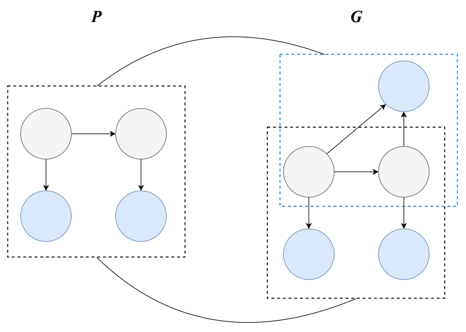
$$1.\ \forall v \in V_\mathcal{P} \Rightarrow \lambda_\mathcal{P}(v) = \lambda_\mathcal{G}(\varphi(v));$$
$$2.\ \forall (u, v) \in E_\mathcal{P} \Rightarrow (\varphi(u), \varphi(v)) \in E_\mathcal{G}.$$

The image of the set of nodes in an pattern is then $\varphi(V_\mathcal{P}) = \{\varphi(v) | v \in V_\mathcal{P}\}$.

This strict form of matching is known as *subgraph isomorphism*, ensuring that both the structure and labels of the pattern graph are preserved in the matched subgraph (see Figure 2.6a), but can also be adapted to consider properties [33].



(a) Isomorphism



(b) Homomorphism

Figure 2.6: Isomorphism and Homomorphism between a pattern P and a data graph G

In dialog-like graph query languages, this requirement can be relaxed to allow *graph homomorphisms*, where the mapping does not need to be exact, allowing multiple pattern nodes to map to the same node in the data graph (see Figure 2.6b) [32]. This allows for more flexible matching, which is commonly used in practical graph query systems [5]. However, even under homomorphism-based semantics, most query languages provide ways to enforce *injectivity* for specific variables, ensuring that certain pattern nodes must match distinct nodes in the data graph [5]. This enables users to control the strictness of the match when needed [5].

2.7.5 Incremental graph pattern matching

In dynamic graphs, where nodes and edges are added or removed over time, recomputing pattern matches from scratch after each update is often inefficient. *Incremental graph pattern matching* refers to the process of updating the set of matches in response to changes in the graph [33, 92]. The process is triggered by the insertion of a new node [33, 92]. The newly inserted node is treated as a potential candidate for one of the nodes in the pattern graph, and the matching process explores whether a complete match can now be formed as a result [33, 92]. This approach enables targeted, on-the-fly matching that avoids re-evaluation of unaffected parts of the graph.

Building upon pattern matching in Definition 3, incremental graph pattern matching refers to the evaluation of whether a pattern graph $\mathcal{P} = \langle V_{\mathcal{P}}, E_{\mathcal{P}}, \lambda_{\mathcal{P}} \rangle$ matches any new subgraph in a dynamic data graph $\mathcal{G} = \langle V_{\mathcal{G}}, E_{\mathcal{G}}, \lambda_{\mathcal{G}} \rangle$, as the graph evolves [92, 33]. Instead of re-evaluating all possible matches from scratch, the system checks whether the insertion of a new node $v' \in V_{\mathcal{G}}$ (and its incident edges) completes the pattern of $\mathcal{P}$ [33].

2.8 Technology of Graph Databases

Graph databases are purpose-built systems designed to store, manage, and query data represented as a graph. Unlike traditional relational databases that model data in tables, graph databases structure information as nodes and edges, often enhanced with rich metadata in the form of labels and properties (see Section 2.7.3) [71]. This structure makes them particularly suited for representing highly interconnected data, such as social networks, supply chains, or, as in this thesis, event streams [4].

By utilizing native graph storage and traversal mechanisms, graph databases enable expressive queries and efficient execution of pattern matching [4]. These capabilities are especially beneficial for use cases involving complex relationships and dynamic data [4], where traditional systems would require costly joins and precomputed views [93].

In this section, the technologies underpinning graph databases used in this thesis are described. It begins by introducing Cypher, a declarative query language used to express graph patterns. Then, the two database systems evaluated in this work, Neo4j and Memgraph, are presented, highlighting their core features, architectural differences, and support for reactive behavior through triggers. These technologies form the foundation for the graph-based event aggregation system implemented in this study.

2.8.1 Cypher

Cypher is a declarative graph query language used to express queries in graph databases [65]. It allows users to define patterns of nodes and relationships using an intuitive ASCII-art-like syntax (see Figure 2.7) and to retrieve subgraphs that match those patterns [34, 67]. This makes Cypher particularly suitable for graph pattern matching, traversals, and structural queries [34].

```
//Node
(variable:NodeLabel {property: "value"})

//Relationship
-[variable:EdgeLabel]->
```

Figure 2.7: Syntax example in Cypher

Cypher queries describe the desired graph pattern rather than specifying how to execute it, which aligns with the declarative nature of the language [67]. For example, a query might express a search for a node of a certain type connected to another node via a specific relationship, without detailing the traversal algorithm to use (see Figure 2.8) [67]. This abstraction allows developers to focus on the structure of the data rather than the underlying mechanics of the database engine [34].

```
//Pattern query
MATCH (node1:EiffelEvent)-[:CAUSE]->(node2:EiffelEvent)
RETURN node1, node2
```

Figure 2.8: Example of a query in Cypher

In this thesis, Cypher is used to query event graphs of Eiffel data. The query language enables the expression of complex event patterns, such as chains of linked events or branching structures involving multiple event types. These capabilities are central to the implementation of pattern detection and event aggregation in the system.

2.8.2 Neo4j

Neo4j is an on-disk graph database management system designed to store, manage, and query graph-structured data [76, 66].

One core strength of Neo4j is its support for efficient graph traversal and pattern matching [76], which makes it suitable for applications that involve complex relationships, such as recommendation systems, social networks, and event-based systems [95].

Neo4j, like many graph databases, is optimized for static queries but lacks native support for reactive queries that respond to changes as they happen [18, 31]. In dynamic systems such as CI/CD pipelines, events are continuously inserted, and re-evaluating all patterns after each update would be inefficient and impractical. To address this limitation, incremental graph pattern matching (see Section 2.7.5) can be employed, where only the parts of the graph affected by a new insertion are evaluated.

To implement this behavior, Neo4j can be extended using the *APOC* (Awesome Procedures on Cypher) library [64], which provides a set of utilities, including support for defining triggers. A trigger in Neo4j is a declarative hook that runs automatically after a graph update, such as the insertion of a new node, and can execute custom logic, like running a Cypher query to detect pattern matches based on a newly inserted node [64].

2.8.3 Memgraph

Memgraph is an in-memory graph database built for high-performance workloads and real-time analytics [21]. Like Neo4j, it implements the property graph model and supports Cypher as its query language, allowing shared tooling between platforms [63].

What sets Memgraph apart is its focus on streaming data and low-latency event processing. By keeping the entire graph in memory, Memgraph delivers efficient traversal pattern matching [63], making it well-suited for applications such as fraud detection, monitoring, and CI/CD systems.

A unique feature of Memgraph is its native support for database triggers [63]. Triggers are executed in response to data insertions or updates and can run Cypher queries to evaluate patterns and take action [63]. This enables incremental pattern matching, where only newly affected subgraphs are evaluated, drastically reducing the overhead compared to full graph scans.

3 Related Work

Modern event-driven systems use graph technologies and pattern detection to manage and interpret event streams. This chapter reviews related work on graph processing frameworks in Section 3.1, subgraph matching in Section 3.2, and CEP in Section 3.3.

3.1 Graph Processing Frameworks

Graph processing frameworks are designed to run effective analytics on large-scale graphs by utilizing the structure to restrict data access to relevant neighboring nodes [44]. The main focus is to run global algorithms on the entire graph, such as *PageRank*, and do this very efficiently on massive graphs [10, 57]. One such graph processing framework is *ShenTu* [58], which can traverse 70 trillion edges in seconds and run algorithms such as PageRank on a 12 trillion edge graph in less than 10 seconds. This is made possible by enabling a higher level of parallelism through efficient data routing, which allows ShenTu to fully utilize *high performance computing systems* [58]. *GraphIn* is another graph processing framework [79]. It enables analytics in real-time over evolving graphs by using the *incremental-gather-apply-scatter* (I-GAS) programming model [79]. By batching the incoming data stream and applying I-GAS, GraphIn is able to update only the affected portion of the graph when inserting new data and which makes updating algorithm results much more efficient [79]. *Unicorn* is another model that improves processing of large incremental graphs by eliminating redundant calculations [86].

Flink and Spark are big data processing frameworks, each with their own graph processing framework: *Gelly* for Flink and *GraphX* for Spark [50]. In recent work [73], the authors created two algorithms to approach the same problem: detecting user-user communities, *TUCFlink* for Gelly, and *TUCSGF* for GraphX [73]. These algorithms are used to analyze two real-world graph networks and were designed to fully utilize the underlying big data frameworks to run the algorithms efficiently. TUCFlink runs significantly faster than TUCSGF, but uses significantly more memory. [73]. Similar findings are found by Kaepke and Zukunft 2018 [50], who compare the two frameworks using both the PageRank algorithm and a custom one. Their results confirm that Gelly is faster, but more memory-intensive. A contrast is that Flink processes data as a continuous stream, while Spark uses batching, influencing their respective graph frameworks [50].

3.2 Pattern Matching

This section presents research on approaches for detecting graph patterns: on disk pattern matching frameworks and graph databases. Both enable pattern matching.

3.2.1 Pattern matching frameworks

Subgraph matching, explained in Section 2.7.4, is a problem that is usually solved using specialized algorithms for a specific pattern [57, 62]. There have been attempts at generic solutions that work for arbitrary patterns[2, 13, 94, 22, 72, 43], however, they do not perform as well as specialized solutions in practice [62].

GraphZero is an optimization query compiler for subgraph matching that compiles subgraph matching queries to specialized implementations for a specific system [62]. GraphZero addresses three core optimization challenges: computation redundancy, symmetry-induced overcounting, and runtime overhead from constraint enforcement. To eliminate redundant computations, GraphZero derives the complete automorphism group of the query pattern, capturing all symmetry-preserving permutations to identify and remove equivalent embeddings [62]. Second, to prevent overcounting, it enforces minimal ordering constraints on embedding vertices based on the automorphism group, ensuring each subgraph match is uniquely represented by a binary ordering [62]. Finally, GraphZero reduces runtime overhead by proving that only one ordering constraint per matched vertex needs to be checked, significantly lowering execution costs without sacrificing correctness. Together, these optimizations enable efficient and precise large-scale graph pattern matching [62]. Similar tools aiming to improve continuous pattern matching performance include *RapidFlow* [85] and index *CaLiG* [97].

For environments where graphs are too large or the update frequency is too high for a single node to handle efficiently, distributed subgraph matching approaches have been developed to utilize more compute power [57]. Lai et al. have implemented three general-purpose optimizations on timely dataflows which aim to reduce the gap between specialized algorithms and more generic approaches. Using these optimizations, the authors achieve better or similar results to those of specialized solutions [57].

3.2.2 Graph databases for pattern matching

Graph databases are a related set of tools that are able to handle tasks similar to graph processing frameworks. The main differences are that graph processing frameworks focus on loading new data and running global algorithms very efficiently, while graph databases are optimized for complex data relationships, business intelligence applications, and are usually ACID compliant [10, 70, 11].

Graph databases also have the additional benefit of not requiring a datastore [70], which is required by most graph analytics frameworks [56]. Using both tools together can be useful in some cases which can combine their advantages [10, 15]. At the same time, the differences between the two are decreasing in recent years as graph databases evolve to support more global computations and improve performance [10].

There are a lot of different graph databases out there, all with their own drawbacks and benefits. A few of them are; Neo4j [46], Sparksee [78], Titan [20], GraphBase [54], Memgraph [49], TigerGraph [29] and OrientDB [74]. In a study by Gubichev and Then, the performance of different graph database systems was compared using multiple patterns of varying complexity for pattern matching on the LUBM benchmark. The results show significant variation in how well systems handle pattern matching, particularly as the size of the data set grows. Systems using declarative query languages perform better than ones using imperative query languages due to more effective query optimization [40].

Szárnyas et al. creates a benchmark [87] which uses a variation of queries which can evaluate different DBMS. A generator is used to create the dataset, that can build graphs containing more than 100 billion edges with realistic characteristics [87]. The benchmark is tested on Neo4j, TigerGraph and Umbra to verify its portability to different database management systems [87]. A evaluation focusing on incremental graph pattern matching by Bergmann et al. [9]. extending previous evaluations [91]. The study finds that two benchmarks; a petri net benchmark and a object-to-relational schema mapping benchmark, benefit greatly from incremental pattern matching [9]. The execution speed increases, but an increase in memory consumption is a drawback [9].

Hölsch et al. [47] compare the query run time for a graph database with two *relational database systems* (RDBS) for queries [47] that test pattern matching, global graph analytics, and queries with cycles. Two limitations of their evaluation are that it does not include different graph characteristics, such as average node degree, which can significantly affect performance, and it does not filter on attributes for pattern matching. Their results indicate that the RDBS outperforms the graph database in analytical queries, particularly those that require access to most of the graph, such as centrality calculations and shortest-path queries. This performance advantage is largely due to more mature disk and memory management in RDBS [47]. The graph database demonstrates better performance for pattern matching when filtering on node labels or when the queries include cycles, the differences are larger for queries including more nodes. The study highlights that the choice of database system should depend on the dominant query types [47].

Ceri et al. introduce *PG-Triggers*, a formal proposal to enable reactive behavior in property graph databases through the use of triggers [18]. Native trigger functionalities which evaluate constraints in graph data effectively do not exist in most databases. The exception is Memgraph [49, 18], which has native support for triggers. The APOC library enables similar functionality for Neo4j [46, 18]. Many graph databases, such as Amazon Neptune [8] and OrientDB [74], support triggers in some way but do not fully utilize the graph structure of the property and lack many necessary features that a proper trigger system needs [18]. However, other graph databases, such as TigerGraph [29, 18] and GraphDB [42, 18], currently lack all trigger support.

Efforts to standardize graph query languages, such as the development of the *graph query language* (GQL), have focused on querying and schema definition, while reactive behavior, especially triggers, has been overlooked [28]. PG-Triggers seeks to fill this gap by proposing a syntax and execution model that mirrors established standards in relational databases. The framework enables triggers to evaluate on state changes and uses transition variables to carry intermediate data between steps, providing precise control for when and how triggers are activated and conditions are evaluated. PG-Triggers also address limitations in existing applications, such as lack of cascading, unclear trigger ordering, and restricted support for complex conditions or trigger composition [18].

3.3 Complex Event Processing

Complex event processing refers to a technique used to process real-time data streams and extract meaningful insights [60, 37]. Although powerful and flexible, it is difficult to use, which limits its usability [55]. To address this, the authors propose a declarative approach to CEP, enabling users to specify what patterns they want to detect rather than how to detect them. This is achieved through a novel *domain-specific language* (DSL) that enables concise, modular pattern definitions. An accompanying optimizer translates these high-level specifications into efficient execution plans for event processing [55].

Esper is a popular CEP tool used in the industry [61, 12, 100, 45]. Currently, there exists a lack of performance and scalability benchmarks [68], to address this, Ortiz et al. create a benchmark with multiple different event patterns that utilize commonly used CEP operators, providing a comprehensive assessment of system capabilities under varied workloads. The data set was evaluated with three software architectures: two centralized setups, one based on open-source Esper with RabbitMQ, another using Esper Enterprise with Kafka, and lastly a distributed deployment of Esper Enterprise Edition with Kafka. Performance was evaluated by measuring CPU usage, memory consumption, latency, and throughput across all configurations [68]. Results showed that RabbitMQ delivers higher performance but lower reliability than Kafka at very high throughputs, while distributed setups demand more resources but provide greater scalability. Other established CEP frameworks include Siddhi [84], Apama [17], and Tibco StreamBase [88, 98].

4 Method

This chapter outlines the methodological approach used to design and evaluate alternative mechanisms for event aggregation in CI/CD pipelines. The overarching goal is to construct and assess systems capable of detecting complex event patterns using scalable and expressive models. The evaluated technologies are in this case graph databases and complex event processing.

To achieve this, the chapter applies the *design science research* methodology as a guiding framework, focusing on the development and evaluation of artifact-based solutions in Section 4.1. The chapter transitions to introducing a generalized conceptual architecture, in Section 4.2, that captures the essential components required for detecting event patterns in a streaming context. Section 4.3 then introduces a set of relevant event patterns and explains how user-defined aggregation rules can be constructed to capture them. It also compares the expressiveness of the different approaches used to define these rules. Following this, the chapter then presents the architectural choices in graph databases, in Section 4.4. Thereafter the architectural structure for achieving event aggregation is explained in Section 4.5, for graph databases, and Section 4.6, for complex event processing. Finally, the chapter concludes with an overview of extended use-cases found with graph databases in Section 4.7.

4.1 Design Science Research Methodology

This thesis adopts the design science research (DSR) methodology, which is particularly suited for solving practical problems through the creation and evaluation of innovative artifacts. DSR provides a structured approach for designing and validating technical artifacts that contribute to practical applications. This aligns well with the goals of this study: to design an improved event aggregation system and evaluate it in the context of CI/CD pipelines.

The methodology is inspired by the widely cited framework by Peffers et al. [69], which outlines six activities. Each activity is stated below, along with how it is applied in the context of this study:

1. **Problem Identification and Motivation:** It is difficult to write and manage rules in Eiffel Intelligence, and there is a need for more flexible, efficient, and user-friendly aggregation system.
2. **Define Objectives for a Solution:** The system should support close to real-time event ingestion, user-defined pattern detection, and notification logic while improving usability.
3. **Design and Development:** Three system architectures were developed based on graph databases and a CEP engine. These systems use pattern matching and triggering mechanisms to support user-defined aggregation rules and notifications.
4. **Demonstration:** The functionality of the systems was demonstrated using realistic Eiffel event data replayed. Patterns that are commonly used in real life were implemented and validated.
5. **Evaluation:** System performance was evaluated by measuring throughput and resource consumption across different event loads and varying numbers of active aggregation rules.
6. **Communication:** The designs, methods, and results are presented in this thesis and may form the basis for future works or be directly applicable to industrial event aggregation systems, particularly in CI/CD pipelines.

The DSR methodology also emphasizes the dual goals of producing both a *working solution* (artifact) and a *scientific contribution*. In this thesis, the artifacts include architectural designs, pattern detection logic, and reusable notification mechanisms. These are evaluated not only on performance but also on resource usage over time.

The nature of DSR is reflected in the development of three separate systems with shared goals but different technical trade-offs. This allows the research to explore generalizable insights across different design choices.

4.2 Conceptual Architecture

The architecture of an event aggregation system fundamentally shapes its scalability, maintainability, and performance. In systems that process large volumes of continuous data, such as CI/CD event streams, the choice of architectural design directly influences how effectively patterns can be detected, rules managed, and notifications generated. For this reason, it is essential to understand not only the high-level components of such a system, but also how different design choices affect the interaction between data ingestion, rule evaluation, and output.

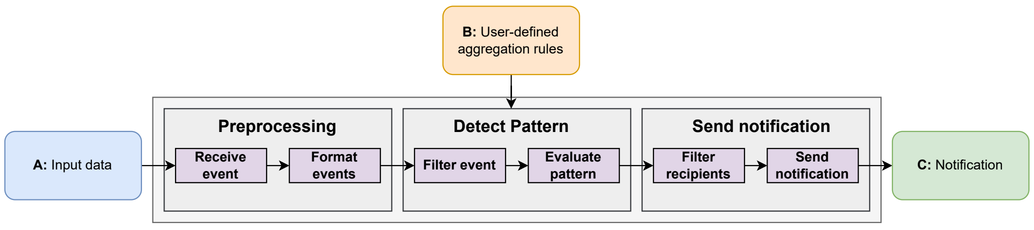


Figure 4.1: High-level architecture regarding external elements and internal components

This section presents and compares the architectural designs underlying the evaluated systems. Figure 4.1 illustrates the high-level architecture for the proposed event aggregation system, highlighting the interaction between external inputs and outputs, as well as the internal components. The system is designed to handle three external elements: *(A) Input data*, which represents the raw events entering the system, and can look like

Figure 2.1; *(B) User-defined aggregation rules*, which define how events should be combined to form higher-level patterns; and *(C) Notification*, which is the output generated when a specified pattern is detected.

The external elements correspond to distinct actors that interact with the system, shown in Figure 4.2. The input data is generated by CI/CD systems as part of ongoing software development and delivery activities. These systems act as event producers, emitting structured events that represent actions such as builds, tests, or deployments. The user-defined aggregation rules are created by users, typically developers or DevOps engineers, who define the logic for how events should be grouped into meaningful patterns. Lastly, the resulting notifications are delivered to consumers, which may be individuals, services, or automated components that react to detected patterns, thereafter they can perform arbitrary actions based on the implementation, such as logging the detected pattern, forward it, or trigger further processing.

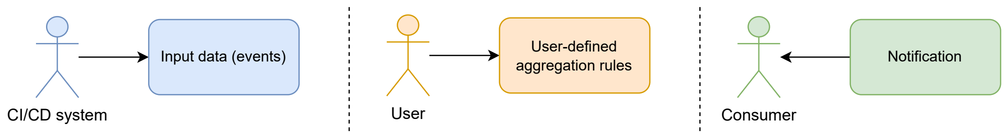


Figure 4.2: Actors of the system

Internally, the system consists of three main components. The first is responsible for *preprocessing* events, acting as the system's entry point for raw input data and formats them into a consistent and structured representation suitable for rule evaluation. Once formatted, the events enter the *detect pattern* component, where events are first filtered based on event type, and thereafter evaluated against defined aggregation rules to identify higher-level patterns of interest. When a pattern is matched, a final component, *send notification*, handles the generation and dispatching of notifications by filtering aggregated objects on recipients interests. Together, these components form a streamlined processing pipeline that enables rule-driven aggregation of event streams.

4.3 Specification of Aggregation Rules

To enable event aggregation, users must be able to define how incoming events should be grouped into higher-level patterns. This section describes how aggregation rules can be specified in graph databases and complex event processing engines. It concludes with a comparison of the expressiveness and flexibility offered by Cypher (used in graph databases), FlinkCEP, and Eiffel Intelligence.

4.3.1 Graph databases

In this section, several graph patterns relevant to the system are described. First, three commonly used patterns are presented that are currently supported by the Eiffel Intelligence and are the most commonly used. Then, a fourth pattern is introduced to illustrate a case that cannot be detected with the existing implementation.

Artifact created pattern

The *artifact created pattern* is the simplest patterns defined within the Eiffel protocol. It consists of only two nodes connected by a single edge. The trigger node in this pattern is an event of type `EiffelArtifactCreatedEvent` which is linked to an event of type `EiffelFlowContextDefinedEvent` via a relationship of type `FLOW_CONTEXT`. The function of the different event types and links are:

- **EiffelArtifactCreatedEvent (ArtC)**: Indicates that an artifact, such as a binary or a software package, has successfully been created.
- **EiffelFlowContextDefinedEvent (FCD)**: Event that establishes the context or workflow in which the artifact was created, connecting it to a particular CI/CD pipeline or development process.
- **FLOW_CONTEXT**: Link type which specifies the flow context of the event, indicating the CI/CD pipeline in which it occurred.

Graphically, the pattern can be visualized as in Figure 4.3. In Cypher, this pattern can be used within a trigger to achieve its intended functionality. Figure 4.4 illustrates how such a pattern can be defined.

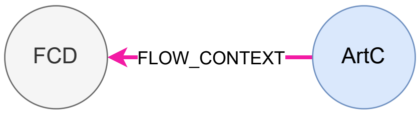


Figure 4.3: Artifact Created Pattern

```
// Filter node type on newly inserted events
WITH [n IN $createdNodes WHERE n.type = "EiffelArtifactCreatedEvent"] AS nodes

// Query for each recieved trigger node
UNWIND nodes AS n
MATCH (n)-[:FLOW_CONTEXT]->(e:Event {type:"EiffelFlowContextDefinedEvent"})
```

Figure 4.4: Artifact Created Pattern query in Cypher

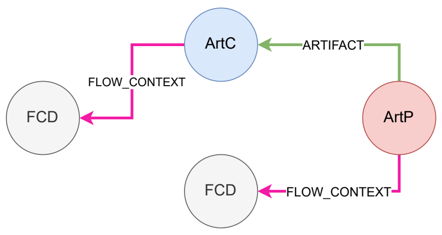
Practical Use Case

This pattern is used in fast-paced development environments where developers push frequent changes and prioritize rapid feedback over full validation. When an `ArtC` occurs and is linked to a known context via `FCD`, the system recognizes that a new build has been produced in a valid pipeline.
This pattern is attractive when the developer does not require the artifact to undergo testing or verification before it becomes visible or usable, for example, in early-stage prototyping, internal experiments, or rapid iteration cycles.

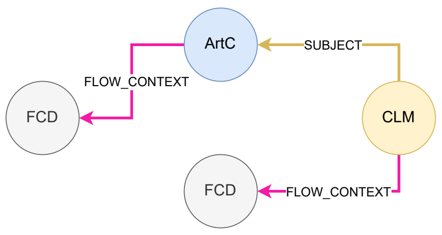
Artifact published pattern

The *artifact published pattern* builds upon the artifact created pattern but links to an `EiffelArtifactPublishedEvent` over an `ARTIFACT` link. This event also links to an additional `FCD`. The patterns can be homomorphic, meaning that the two different `FCD`s can be the same or different. It looks like in Figure 4.5a. The new event and link types can be explained as follows:

- **EiffelArtifactPublishedEvent (ArtP)**: Signifies that a software artifact, previously declared by an `EiffelArtifactCreatedEvent`, has been successfully published and is now accessible for retrieval from one or more specified locations.
- **ARTIFACT**: A link type that identifies the artifact that was published.



(a) Artifact Published Pattern



(b) Confidence Level Reached Pattern

Figure 4.5: Artifact Published Pattern and CL Reached Pattern

```
// Filter node type on newly inserted events
WITH [n IN $createdNodes WHERE n.type = "EiffelArtifactPublishedEvent"] AS nodes

// Query for each recieved trigger node
UNWIND nodes AS n
MATCH (n)-[:FLOW_CONTEXT]->(e:Event {type:"EiffelFlowContextDefinedEvent"})
MATCH (n)-[:ARTIFACT]->(c:Event {type:"EiffelArtifactCreatedEvent"})
MATCH (c)-[:FLOW_CONTEXT]->(f:Event {type:"EiffelFlowContextDefinedEvent"})
```

Figure 4.6: Artifact Published Pattern query in Cypher

Such a pattern uses `ArtP` as its trigger node. Shown in Figure 4.6 is an example of how the pattern can be detected with a query.

Practical Use Case

This pattern adds an extra layer by requiring that a built artifact has been successfully published to a repository. It is commonly used in projects that require that all artifacts are made accessible.
Compared to the artifact created pattern, this informs the user that the artifact is both built and made available, which is vital in multi-team environments where other components rely on published outputs.

Confidence level reached pattern

The *confidence level reached pattern* is very similar to the artifact published pattern. The only difference is that instead of the `ArtP`, an event of type `EiffelConfidenceLevelModifiedEvent` is used, and its link to `ArtC` is of type `SUBJECT`. It looks like in Figure 4.5b. The new event and link types can be explained as follows:

- **EiffelConfidenceLevelModifiedEvent (CLM)**: Indicates that an entity has met (or failed to meet) a defined confidence level, often used to mark its readiness for progression in the CI/CD pipeline.
- **SUBJECT**: Link type that specifies the subject to which the confidence level applies.

The trigger node for this pattern is `CLM`. A query of such a pattern is shown in Figure 4.7

```
// Filter node type on newly inserted events
WITH [n IN $createdNodes WHERE n.type = "EiffelConfidenceLevel..."] AS nodes

// Query for each recieved trigger node
UNWIND nodes AS n
MATCH (n)-[:FLOW_CONTEXT]->(e:Event {type:"EiffelFlowContextDefinedEvent"})
MATCH (n)-[:SUBJECT]->(c:Event {type:"EiffelArtifactCreatedEvent"})
MATCH (c)-[:FLOW_CONTEXT]->(f:Event {type:"EiffelFlowContextDefinedEvent"})
```

Figure 4.7: Confidence Level Reached Pattern query in Cypher

Practical Use Case

This pattern is relevant in production-grade CI/CD workflows where artifacts must meet quality standards before being promoted. This pattern ensures that an artifact not only exists, but also satisfies predefined quality metrics (e.g., full test coverage, security, performance, regression tests).
Compared to previous patterns, this one introduces a quality gate, making it appropriate for production pipelines or organizations with strict release policies. It helps ensure that only safe artifacts proceed further into the pipelines.

Artifact approved pattern

The *artifact approved pattern* represents a composite pattern that combines the functionality of all previously described patterns. It involves an `ArtC` that is referenced by both an `ArtP` and a `CLM`. A visual of this pattern is shown in Figure 4.8.

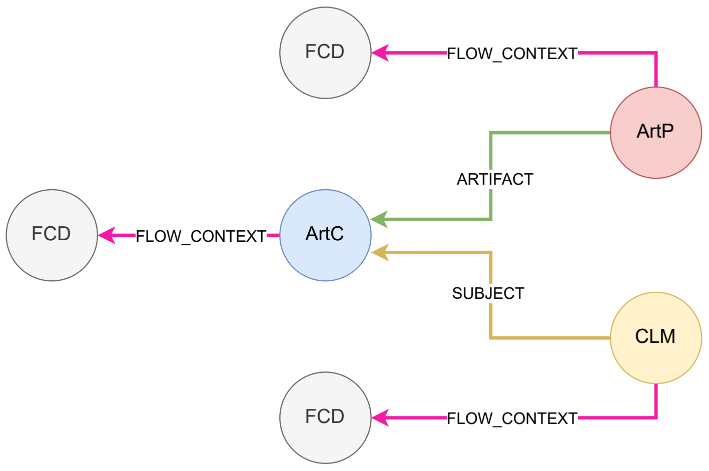


Figure 4.8: Artifact Approved Pattern

This pattern presents two key challenges from a detection perspective. First, it includes two distinct trigger events, `ArtP` and `CLM`, either of which may serve as the entry point for evaluation. Second, it requires both upstream and downstream traversal of the event graph. For example, if a `CLM` event is received, the system must first search upstream to locate the corresponding `ArtC` event, and then continue downstream to determine whether an `ArtP` event exists that references the same artifact. This combination of bidirectional traversal and multiple trigger nodes requires the system to have such capabilities, which Eiffel Intelligence does not have.

To solve these challenges, queries such as in Figure 4.9 can be implemented. By using *if statements* for each trigger event, it is possible to have several at once. By also using a graph query language, which does not have any restrictions on upstream and downstream search, it is possible for both problems to be solved.

```
n.type = "EiffelConfidenceLevelModifiedEvent", // If the new node is a CLM, do
    // Pattern query
    MATCH (n)-[:FLOW_CONTEXT]->(e:Event {type:"EiffelFlowContextDefinedEvent"})
    MATCH (n)-[:SUBJECT]->(c:Event {type:"EiffelArtifactCreatedEvent"})
    MATCH (c)-[:FLOW_CONTEXT]->(f:Event {type:"EiffelFlowContextDefinedEvent"})
    MATCH (c)<-[:ARTIFACT]-(d:Event {type:"EiffelArtifactPublishedEvent"})
    MATCH (d)-[:FLOW_CONTEXT]->(g:Event {type:"EiffelFlowContextDefinedEvent"})
...

n.type = "EiffelArtifactPublishedEvent", // If the new node is an ArtP, do
    // Pattern query
    MATCH (n)-[:FLOW_CONTEXT]->(e:Event {type:"EiffelFlowContextDefinedEvent"})
    MATCH (n)-[:ARTIFACT]->(c:Event {type:"EiffelArtifactCreatedEvent"})
    MATCH (c)-[:FLOW_CONTEXT]->(f:Event {type:"EiffelFlowContextDefinedEvent"})
    MATCH (c)<-[:SUBJECT]-(d:Event {type:"EiffelConfidenceLevelModifiedEvent"})
    MATCH (d)-[:FLOW_CONTEXT]->(g:Event {type:"EiffelFlowContextDefinedEvent"})
```

Figure 4.9: Artifact Approved Pattern query in Cypher

Practical Use Case

The artifact approved pattern is designed to detect more comprehensive situations by combining the functionality of all previous patterns. Specifically, it verifies that an artifact was created, published, and reached a certain confidence level, all within the correct CI/CD flow context.
This pattern is useful in pipelines that require strict traceability and quality assurance before promotion or deployment. For example, instead of acting on each event in isolation, a deployment system can wait until the artifact passes all conditions, build completion, publication, and quality approval, and then trigger promotion to staging or production. By integrating these criteria into a single pattern, the system ensures that only fully approved artifacts proceed further.

4.3.2 Complex event processing

Complex event processing (CEP) is a promising approach for pattern matching in graphs. CEP systems are designed to identify and analyze patterns in large volumes of events in real-time [39], which closely aligns with the aim of this thesis.

Flink is a popular stream processing framework with a powerful CEP library, and is one of the tools that this thesis seeks to evaluate. In FlinkCEP, a pattern is defined by explicitly declaring a sequence of events and the conditions under which they should occur. Each step in the sequence is added using methods such as `.next()` or `.followedBy()`, forming an imperative specification of the pattern. Figure 4.10 shows an example of how the artifact created pattern could be declared in this way. This design inherently limits FlinkCEP to downstream searches, as it constructs patterns by consuming events in the order they arrive, building forward from the initial trigger.

Because each step and condition must be specified manually, defining complex or large patterns quickly becomes difficult and error-prone. The lack of a high-level declarative abstraction means users must manage the pattern structure in detail, making it particularly challenging to express patterns with multiple trigger events or flexible event ordering.

```
Pattern<Event, ?> artifactCreated = Pattern.<Event>begin("FCD")
  .where(SimpleCondition.of(event -> event.getMeta().getType()
  .equals("EiffelFlowContextDefinedEvent")))
  .followedByAny("ArtC")
  .where(new IterativeCondition<Event>() {
      @Override
      public boolean filter(Event event, Context<Event> ctx) throws Exception {
          //Exit early due to wrong type or missing links
          if (event.getLinks().isEmpty() || !Objects.equals(event.getMeta().
              getType(), "EiffelArtifactCreatedEvent")){
              return false;
          }
          //Verify correct link
          Event previousEvent = ctx.getEventsForPattern("FCD").iterator().next();
          return HelpFunctions
              .linksToEvent(event, previousEvent, "CONTEXT_DEFINED");
      }
  });
```

Figure 4.10: Code to detect the Artifact Created Pattern using FlinkCEP

As a result, FlinkCEP is best suited for linear, time-constrained sequences rather than graph-like or highly interconnected event patterns.

4.3.3 Comparing expressiveness in technologies

The expressiveness of an aggregation system determines the range and complexity of event patterns that can be defined and detected. It directly affects the system's ability to support diverse use cases and adapt to evolving requirements. This subsection compares the expressiveness of Cypher (used in graph databases), FlinkCEP, and Eiffel Intelligence, by examining how each allows users can define aggregation rules. Table 4.1 compares the possibilities of expressiveness for each technology. If a feature is possible in the tool, it is denoted with a checkmark, if it is possible, but difficult, then it is a warning sign. Finally, if it is not possible at all, then the feature is marked with a cross.

Feature	**Cypher**	**FlinkCEP**	**Eiffel Intelligence**
Downstream search	✓	✓	✓
Upstream search	✓	✗	✓
Upstream-downstream combination	✓	✗	✗
Multiple trigger events	✓	⚠	✗
Aggregate large patterns	✓	⚠	⚠

Table 4.1: Comparison of expressiveness in Cypher, FlinkCEP, and Eiffel Intelligence

Overall, Cypher offers the highest expressiveness, supporting all listed features. FlinkCEP supports basic downstream pattern detection and struggles with more complex structures, such as multiple trigger events or combined aggregation of larger patterns, which require workarounds. Eiffel Intelligence supports only basic linear aggregation and lacks the ability to express more complex pattern combinations or handle multiple triggers, making it the most limited in terms of rule expressiveness.

This comparison highlights the trade-off between ease of use and expressive power, with Cypher offering a more flexible and developer-friendly approach to defining diverse and reusable event patterns.

4.4 Architectural Design Choices in Graph Databases

To enable accurate actions to changes in CI/CD pipelines, it is essential to detect and respond to event patterns. This work explores a trigger-based architecture using graph databases, where in order to detect and respond based on input events, three steps need to be done:

1. **Filtering:** An event is only interesting if it has some specific type.
2. **Querying:** If an event of an interesting type is inserted, a query is then done to see if the event fulfills a pattern.
3. **Routing:** If a pattern is fulfilled, then a notification should be sent, which is done by routing the aggregated information in the pattern to a specific endpoint.

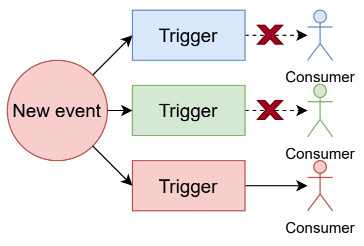
Triggers act as listeners within the graph database and are central to the detection of patterns as well as sending notifications. They are executed automatically in response to updates in the graph, such as insertions, deletions, or modifications of nodes or relationships. Through this mechanism, the system can react to changes in the graph as they occur and determine whether relevant event structures have emerged.

Multiple design alternatives for implementing the information flow were considered. These alternatives are *dedicated triggers per consumer*, *generic trigger and routing interface*, *conditional routing*, *generic endpoints routing* as well as *discrimination network*. Each alternative presents specific advantages and disadvantages, particularly in terms of performance and usability.

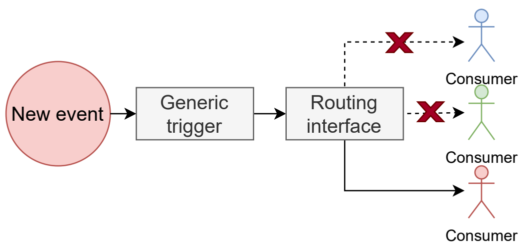
4.4.1 Dedicated triggers per consumer

In this approach, each consumer is assigned a dedicated trigger that listens for specific patterns in the database. This design enables consumers to operate independently, as each trigger is responsible for its own filtering, querying, and routing logic. By attaching a self-contained trigger to the database, a consumer can receive notifications directly, without requiring additional orchestration or coordination with other components. This information flow is depicted in Figure 4.11a.

A key drawback of this approach is the presence of redundant operations, especially as the number of active triggers increases. Since each trigger performs its own filtering independently, identical or overlapping filtering logic may be executed multiple times for the same event. There might also exist several triggers active that filter for the same node type as well as query for the same pattern. This leads to computational overhead and reduced efficiency as the system scales.



(a) Information flow using unique triggers



(b) Information flow for handling routing to multiple consumers in a trigger

Figure 4.11: Dedicated triggers per consumer and Generic trigger and routing interface

4.4.2 Generic trigger and routing interface

The redundancy identified in the previous approach of Section 4.4.1, where each consumer is assigned a separate trigger, can be addressed by restructuring the trigger mechanism to minimize duplicated filtering and querying logic.

To reduce redundancy in pattern filtering and querying, the trigger mechanism can be divided into two components: a *generic trigger*, which handles the common logic for detecting a pattern, and a *routing interface*, which allows users to configure filters on attributes within the aggregated object and specify endpoints for notifications. This separation enables multiple consumers to subscribe to the same pattern while applying different attribute filters, without the system duplicating the underlying filtering or query logic. This structure is shown in Figure 4.11b.

However, this modular approach introduces a trade-off, which is that users must configure both the generic trigger and the routing interface to set up a complete trigger. This added configuration step increases setup complexity and may result in a more time-consuming process for engineers compared to a monolithic approach.

4.4.3 Conditional routing

As an alternative to splitting the trigger logic into two separate modules from Section 4.4.2, a simplified interface can be achieved using a conditional routing mechanism. In this approach, users specify an attribute of interest along with a notification endpoint. This configuration is then embedded directly into a conditional structure within a single generic trigger. Shown in Figure 4.12, this design reduces redundancy by consolidating routing logic and simplifies the setup process for users, who now only need to configure one component.

This approach introduces a key limitation, where the generic trigger must be continuously updated while active to accommodate new consumers or changes in routing logic. This dynamic modification can lead to maintenance challenges, potential performance issues, and increased risk of inconsistencies, especially in high-throughput environments.

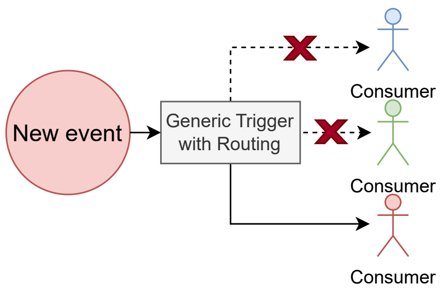


Figure 4.12: Information flow for handling routing to multiple consumers in a trigger

4.4.4 Generic endpoints routing

Using the same core structure as in Figure 4.12, to mitigate the risks associated with conditional routing, triggers can be designed to use generic endpoints. Instead of requiring consumers to specify a custom endpoint, the trigger dynamically constructs the endpoint using information already available within the aggregated object, such as an event ID or consumer identifier. For example, an endpoint might follow the format `myurl/{event_id}`, allowing the system to route notifications correctly based on the content of the event itself.

This approach enables a consumer to receive relevant notifications without the need for a user to register a custom endpoint, as the system can infer the destination from the aggregated data. It simplifies configuration and improves scalability, especially in systems with many consumers.

A potential drawback is that this method relies on the structure and consistency of the aggregated object. If the expected fields (e.g., event_id) are missing, malformed, or ambiguous, it may result in misrouted or failed notifications. Additionally, it assumes that consumers are capable of handling notifications at predictable, structured endpoints.

4.4.5 Discrimination network

An alternative information flow approach in triggers is to decompose the trigger operations into three distinct steps: filtering, querying, and notification. This structure makes it possible to build a *discrimination network*, similar in concept to a *RETE network*.

A RETE network is an efficient pattern-matching algorithm commonly used in rule-based pattern matching [90]. It incrementally processes changes in data by maintaining a network of nodes representing partial patterns, which allows for optimized rule evaluation without repeatedly scanning the entire dataset [90]. By splitting up the logic in this way, several consumers are able to share the same filtering and querying logic. This leads to minimized redundancy in operations regarding each step of the triggering.

As visually presented in Figure 4.13; by adopting a discrimination network, the trigger system can operate filtering, querying and notification, as defined in Section 4.4, in the following way:

1. A trigger is used to filter incoming events based on predefined trigger nodes.
2. If the filter condition is met, specific procedures are invoked to locally query for a matching pattern in the vicinity of the trigger node.
3. If one or more query procedures return a match, additional procedures are executed to generate and send notifications to relevant consumers.

The model is enabled by creating a subgraph of nodes representing procedures, so if a trigger or procedure condition is met, the subgraph is queried to know which procedures to call. This modular approach enhances flexibility, reusability, and performance. It eliminates redundant filtering and query operations. With this structure, a consumer can either define a completely new aggregation rule or reuse an existing one by simply adding specific filters to receive only the information relevant to their product.

4.4.6 Choice of trigger information flow

The primary motivation for the decision-making of the different approaches was the simplicity it offers for users. In the proposed system, the *dedicated trigger per consumer* approach, from Section 4.4.1, was selected, which benefit users with its straightforward setup and minimal configuration effort. Each user could independently define their own trigger logic and notification handling without the need to interface with shared components or routing logic, which aligned well with the project's emphasis on usability whilst also not introducing potential risks with flexibility in the notification stage.

Although the *discrimination network* approach offers theoretically superior performance, modularity, and scalability, its implementation requires a more complex system architecture and deeper integration of modular trigger components. Due to limited time and resources, we found the implementation of this solution to be outside the scope of this project. However, it remains the most promising architecture among stated alternatives and would be a strong candidate for future work, particularly in larger-scale or production-grade systems.

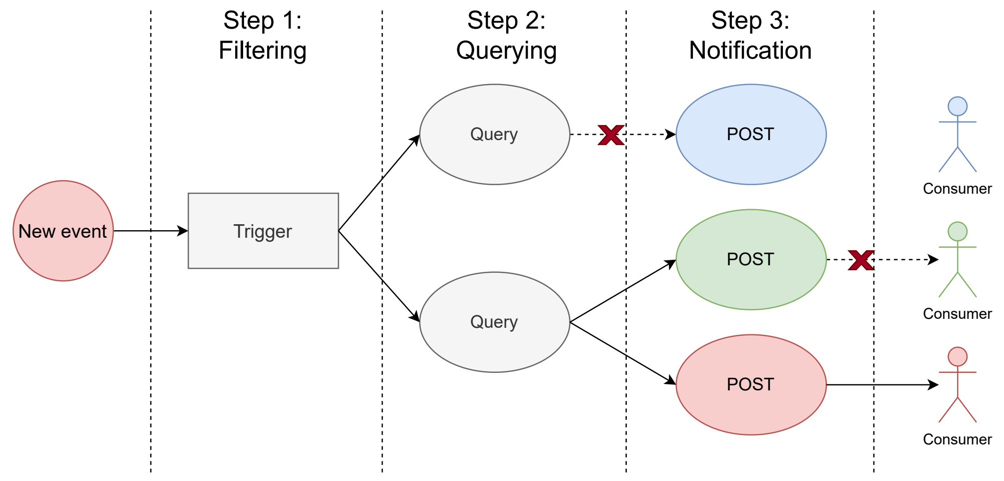


Figure 4.13: Information flow using RETE network

To enable pattern detection in the proposed event-driven systems, defining and managing triggers is central. In graph-based platforms, this process benefits from the natural expressiveness of graph query languages, which allow patterns to be described declaratively in terms of relationships between events. Unlike traditional rule engines or procedural logic, graph queries enable concise and flexible formulations of patterns. The evaluated graph database systems are Neo4j and Memgraph, and below are examples of how triggers can be installed. The structure of how they are used in the proposed system, and in the following examples, stem from the chosen information flow alternative (see Section 4.4.1).

APOC triggers in Neo4j

Neo4j is one of the evaluated graph databases used in the proposed system. In Neo4j, trigger functionality is not available by default but can be created using the APOC library. Each time a node is created, the trigger executes a Cypher query (see Section 2.8.1) that uses the newly inserted node to see if it fulfills a pattern. A query traverses the nodes surrounding the newly inserted node to determine whether a pattern is found. If the query finds a pattern, a notification is finally sent to a consumer. An example of a trigger installation in Neo4j is shown in Figure 4.14, depicting an artifact created pattern, which is explained in Section 4.3.1.

Custom triggers in Memgraph

Memgraph is another evaluated graph database. Memgraph provides native support for triggers using both Cypher and Python. In the example shown in Figure 4.15, Cypher-based triggers are used to achieve the same behavior as in Neo4j (see Section 4.4.6).

Memgraph does not support sending notifications through Cypher natively. To enable this functionality, a custom procedure must be created using Memgraph's procedural API. In the example shown in Figure 4.15, the procedure `trigger.custom_procedure` is used to send a notification to an external endpoint. This procedure must be integrated into Memgraph through its module system, and can look like Figure 4.16.

```
// Call on trigger installation
CALL apoc.trigger.install(
    'neo4j',        // Database name
    'ArtifactCreatedPattern',  // Trigger name
    'WITH [n IN $createdNodes WHERE n.type = "EiffelArtifactCreatedEvent"] AS nodes
    UNWIND nodes AS input
    MATCH (input)-[:FLOW_CONTEXT]->(FCD:Event)
    CALL apoc.load.jsonParams(
        "http://localhost:5000/endpoint",
        {‘Content-Type‘: "application/json"},
        apoc.convert.toJson({ArtC: input.id, FCD: FCD.id})
    ) YIELD value RETURN NULL',
    {phase:"afterAsync"}
);
```

Figure 4.14: Neo4j trigger

```
// Create trigger
CREATE TRIGGER ArtifactCreatedPattern

// When to trigger
ON CREATE
AFTER COMMIT

// What to trigger
EXECUTE
    UNWIND createdVertices AS input
    WITH input
    WHERE input.type = "EiffelArtifactCreatedEvent"
    MATCH (input)-[:FLOW_CONTEXT]->(FCD:Event)
    WITH input.id AS ArtC_id
    WITH FCD.id AS FCD_id
    CALL trigger.custom_procedure(ArtC_id, FCD_id)
    YIELD response
    RETURN response;
```

Figure 4.15: Memgraph trigger

```
@mgp.read_proc
def custom_procedure(ArtC_id: str, FCD_id: str) -> mgp.Record(response=str):
    url = "http://localhost:5000/endpoint"
    headers = {"Content-Type": "application/json"}
    payload = {"ArtC": ArtC_id, "FCD": FCD_id}
    response = requests.post(url, json=payload, headers=headers)
    return mgp.Record(response=response.text)
```

Figure 4.16: Custom procedure for sending notifications in Memgraph

4.5 Event Aggregation with Graph Databases

The graph database acts as the core component for event aggregation. Incoming events are represented as nodes, and their relationships as edges, allowing the system to capture the interconnections between events. This representation enables efficient detection of patterns through declarative queries.

Following the conceptual architecture (see Figure 4.1), to populate the graph database continuously, events are first sent through *preprocessing*, where the raw data is transformed into nodes and edges for insertion into the graph database. Once a node is inserted, the *detect pattern* component and *send notification* component is achieved within trigger logic (see Section 4.4.6) in the graph database that evaluates user-defined patterns against the newly inserted data. If a node fulfills a pattern, the trigger sends a notification to a specified endpoint, enabling follow-up actions by downstream consumers. The structure of the architecture is shown in Figure 4.17.

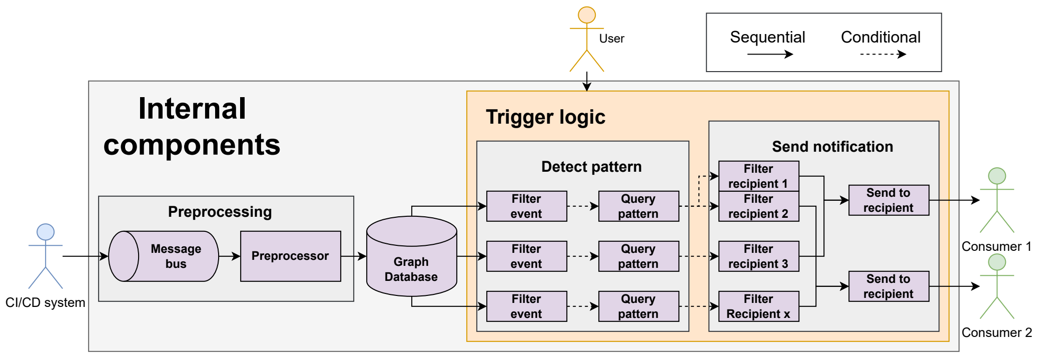


Figure 4.17: Architecture used for graph databases

4.5.1 Preprocessing

The preprocessing is made up of two components, a message bus, as the initial receiver of the system, and a preproccessor, which is formatting the events to a graph-format.

Message bus

The message bus acts as the system's initial receiver, accepting incoming events and making them available for the preprocessor. The chosen tool is RabbitMQ as it aligns with the current implementation in the Eiffel ecosystem, which also uses RabbitMQ for message distribution. RabbitMQ simplifies integration with existing infrastructure and allows for asynchronous, decoupled communication between the data source and preprocessor.

Preprocessor

The preprocessor is a component responsible for transforming incoming JSON objects representing Eiffel events into graph-compatible structures. It listens to the message bus and maps each event into a node and its links to edges, suitable for insertion into the graph database.

To support high-throughput processing to the graph database, the preprocessor uses multiprocessing functionality to parallelize independent event handling and insertion across multiple CPU cores.

To ensure referential integrity, the preprocessor first checks whether the event's dependencies (e.g., links) already exist in the database before insertion. If any dependencies are missing, the event is requeued in the message bus for later processing. This mechanism ensures that no events are inserted with missing links. A flowchart for the different steps is shown in Figure 4.18.

4.5.2 Detect pattern

Pattern detection is carried out within the trigger logic, as outlined in Section 4.4.6. This process consists of two main operations: filtering trigger events and pattern matching, both of which are described below.

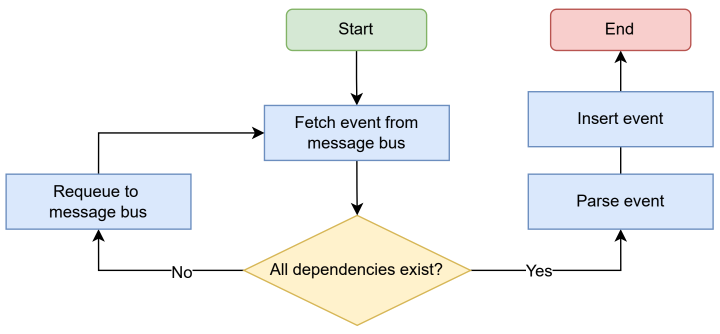


Figure 4.18: Preprocessing logic for an event

Filtering event

When a new event is inserted into the graph database, each active trigger executes a conditional check on the event type of the inserted node to determine whether it is a trigger event (see Section 2.4.1).

Query pattern

If the inserted node is a trigger event for a pattern, the corresponding trigger performs a query to determine whether the new node completes the required structure for that pattern (see Section 4.3.1).

4.5.3 Send notification

To send a notification to a consumer, two operations are performed: *filter recipients*, which identifies which consumers are interested in the aggregated object, and *send notification*, which routes the aggregated object to the appropriate consumer. Similar to pattern detection, both operations are also handled within a dedicated trigger specific to each consumer.

Filter recipients

Consumers are typically only interested in information related to their own products. Therefore, whenever a pattern is detected, the resulting aggregated object must be filtered based on its attributes to determine whether it is relevant to the intended consumer.

Send to recipient

If a user has defined an interest in a pattern, an associated endpoint is registered for receiving notifications. Once the aggregated object has been matched and filtered for the intended consumer, it notifies the consumer's endpoint with the aggregated object. This allows the consumer to react to detected patterns whenever they happen.

4.6 Event Aggregation with Complex Event Processing

The complex event processing engine is responsible for detecting meaningful patterns in real-time event streams. It operates by continuously analyzing sequences of incoming events and evaluating them against user-defined rules.

In alignment with the overall architecture (see Figure 4.1), the system begins with a *preprocessing* step, where events are transformed into a structured format required by the CEP engine, in this case FlinkCEP. Once the events are formatted, the events are sent to the *detect pattern* component, where the CEP engine applies defined aggregation rules. These rules describe sequences or conditions that must be met for a pattern to be recognized (see Section 4.3.2). When such a pattern is detected, the *send notification* component forwards the result to a configured endpoint, allowing external systems to act on the aggregated information.

4.6.1 Preprocessing

The preprocessing for the system with complex event processing at its core starts with a message bus, acting as the receiver. The message bus is used to decouple the input data from the rest of the system, only being fetched when subsequent components are ready.

The preprocessor is responsible for transforming raw event data received from the message bus into a structured format suitable for processing by Flink. Specifically, it converts each event into a *Plain Old Java Object* (POJO), which are simple Java classes with getters and setters [77]. POJOs are needed to ensure that the events are processed in the correct order based on the `time` field of the event (see Figure 2.1).

4.6.2 Detect pattern

In order to detect patterns, the POJOs are sent to FlinkCEP and processed using the Pattern API from the FlinkCEP library to detect patterns. The Pattern API initializes a new instance of the pattern matcher each time it either finds a new start event, the first event in the pattern, or if the event can continue on an existing pattern. Progressing an ongoing pattern will not remove or update the current pattern, but instead keep it and create a new process that handles the pattern that is further along. This will result in an increasing number of matchers being alive at once, and each new POJO event needs to be evaluated against them. To reduce the number of patterns that is alive at any one time it is possible to, for example, use windowing strategies to limit the patterns time to live, be stricter in the order which events have to arrive to fit a certain pattern or in other ways reduce the conditions for pattern matchers to be expanded and created. Using these methods would consequently limit which patterns could be found and no longer meet the requirements. For example, connected events can in practice have weeks between them and, in some cases, years.

4.6.3 Send notification

Once a pattern has been detected, the system prepares a notification that contains information about the matched event sequence. This notification is then routed to a designated downstream consumer. Conceptually, this step involves transforming the matched result into a structured message and forwarding it to an external system that has expressed interest in such patterns.

4.7 Extended Use-cases using Graph Databases

A core objective of this thesis is to enable more flexible, user-defined aggregation patterns, overcoming the structural limitations imposed by Eiffel Intelligence. To illustrate the broader applicability and potential of the proposed system, this section presents two extended use-cases that would be difficult or infeasible to support using Eiffel Intelligence alone. These are pipeline aggregation and legacy support.

4.7.1 Pipeline aggregation

A key advantage of graph databases is their ability to efficiently traverse and query large, interconnected structures. In the context of CI/CD pipelines, this enables users to construct queries that span multiple layers of relationships. A user could for example query from a finished product all the way back to the source code commits, or vice versa. One such use-case is generating a complete view of the events leading up to a particular product version, including which builds, tests, and artifacts contributed to its creation. Conversely, a user might wish to trace forward from a specific commit to determine which product versions include it.

This kind of holistic view is difficult to achieve in Eiffel Intelligence. Its rule-based design focuses on detecting discrete aggregation patterns in isolation and lacks native support for traversing or querying historical event relationships across multiple aggregation levels. Building such queries in Eiffel Intelligence would require significant customization or external tooling.

In contrast, graph databases store events and their links in a navigable graph. This allows for expressive and ad hoc exploration of complex pipelines using standard query languages. As a result, pipeline-level aggregation and traceability, such as root-cause analysis, impact tracking, or historical lineage, becomes both feasible and user-friendly within a single system.

4.7.2 Legacy support

Within Ericsson, two generations of the Eiffel protocol are in use: Generation 1 (E1) and Generation 2 (E2). While E2 is the standardized and externally documented version, many systems still emit E1 events due to historical dependencies and integration constraints.

These two generations differ in their schema, link structure, and metadata representation. Eiffel Intelligence is tightly coupled with the E2 format and cannot easily be extended to support E1 events without major changes to its core logic.

In contrast, the graph-based approach proposed in this thesis provides a natural abstraction layer for handling heterogeneous event formats. Since the system allows for specific preprocessing logic and trigger conditions for the different generations, it becomes possible to normalize or map E1 events into a unified internal representation. Alternatively, E1 and E2 events could have widely different structure, but still be used in the same database, as long as the user knows how the generations might be interconnected when creating their triggers. As a result, both E1 and E2 events can coexist within the same graph database, enabling pattern detection across generations.

This eliminates the need for maintaining two separate aggregation systems and significantly reduces maintenance complexity. Furthermore, it allows for a smoother transition from E1 to E2, as legacy systems can continue to function while newer components adopt the standardized format.

Summary: The method chapter uses design science research to develop and evaluate alternative event aggregation systems. Three systems are built based on: Neo4j, Memgraph, and Flink. Specification of aggregation rules shows that Cypher has the greatest flexibility when it comes to expressiveness. Trigger-based incremental pattern matching is implemented in graph databases in order to detect patterns in event streams, whereas complex event processing continuously analyzes event streams by maintaining and updating partial matches as new events occur. Furthermore, using graph databases enables extended use-cases, such as the ability to aggregate large patterns like entire pipelines, or enable legacy support between protocol versions.

5 Evaluation

This chapter presents the evaluation setup and tests in Section 5.1, and thereafter results from the experimental evaluation in Section 5.2. These are done to answer the following research questions:

(RQ1) What technologies can aggregate events while maintaining sufficient performance and efficient resource usage over time?

- RQ1.1 How do the evaluated technologies perform in terms of throughput and resource usage over time?
- RQ1.2 To what extent do the technologies exhibit consistent performance across identical runs?
- RQ1.3 How accurately do the technologies reproduce expected pattern matches compared to a known ground truth?

(RQ2) How does the number of active patterns affect throughput on the evaluated technologies?

5.1 Setup and Tests

This section provides a detailed overview of the tests that were conducted, the purpose behind each test, the metrics that were chosen and why they were selected, as well as the parameters used during testing. The evaluated tools are Neo4j, Memgraph and Flink, and begins with a scalability test (RQ1.1), followed by a stability test across repeated runs (RQ1.2), an accuracy test comparing detections to a known truth (RQ1.3), and finally a test of how performance is affected by the number of active patterns (RQ2).

General setup

The environment in which the tests were conducted was consistent for all tests. They were carried out on a Red Hat *Virtual Desktop Infrastructure* (VDI) provided by Ericsson. The virtual environment had a CPU with 8 cores, 64 GB of RAM, and 500 GB of storage.

Message bus

For all tests, RabbitMQ version 4.1.0 was used as the message bus. RabbitMQ was started using Docker and initialized as durable to ensure data persistence in case it encountered any issues. The prefetch count was set to 250 events to reduce the risk of the message bus becoming a bottleneck. The queues were reset before each test to ensure a clean start. A script was used to read and publish 10.000 events at a time to the message bus, and if the queue exceeded 50.000 events, the publishing was temporarily postponed. This limit was chosen because it did not allow the tools enough time to empty the queue.

Graph databases

The following graph databases were used: Neo4j community version 2025.01.0 and Memgraph community version 2.14.1. To prepare the test environment, the message bus was first started together with its script to publish data. Then the chosen graph database was started: Memgraph was instantiated through a Docker container, whilst Neo4j was launched directly on the VDI. Both graph databases were reset between tests, as to start from a clean environment. The needed triggers were then installed in the database and an index was created on the event ID field in order to improve query performance and ensure that insert and lookup operations remain efficient as the database grows. Thereafter a local server was started to receive notifications. After the preparations were completed, to start inserting events, a Python program preprocessor was started. The preprocessor's multiprocessing was also set to 32 processes to ensure consistent high CPU usage.

Complex event processing

For the complex event processing tool, Flink version 1.20.0 was used. The Java code for pattern detection, preprocessing, and subscribing to the message bus was compiled into a fat JAR file using Maven. To ensure a clean environment and eliminate any residual state between runs, the Flink cluster was restarted before each job submission. After the JAR file was submitted to the Flink cluster as a job, the running job began receiving events from the message bus once it had initialized. When a pattern was detected, a notification was sent to the terminal for logging.

Eiffel Intelligence

The current event aggregator in Eiffel has both a backend and a frontend. Eiffel Intelligence version 3.2.8 was used for the backend components, and version 3.0.6 was used for the frontend. To set up Eiffel Intelligence, a Docker Compose file was started with all the required services. These are the event repository, the message bus, the frontend, and each backend corresponding to the aggregation rules used. Before running any tests, the system was restarted to ensure a clean environment. The event repository is then filled with the entire dataset used for the test, because the system does not ensure that events are inserted into the database before being handled by the event aggregator, which can lead to missed pattern detections. Next, all needed subscriptions were added to Eiffel Intelligence through the frontend to enable notifications. Finally, a server was initiated to receive these notifications. Once the environment was set up, the Python script responsible for publishing event data to the message bus was started.

Metrics setup

To evaluate and compare the performance of the different tools, several key metrics were monitored: disk usage, RAM usage, CPU utilization, and throughput. These metrics were continuously collected every 10 seconds during the tests. The resource usage was tracked using the psutil library in Python and was measured at the system level, starting from the moment each test began. This ensured that only the additional consumption introduced by the tested tool was captured, without interference from unrelated background processes or prior system activity. Insert speed was measured by calling the RabbitMQ API for the number of newly acknowledged events since the previous measurement. Acknowledged events represent those successfully processed by the tool.

5.1.1 Dataset

We used real-world event data from existing *event repositories* to ensure that the system operates under conditions that closely resemble actual production environments. These event repositories are populated with historical Eiffel event data collected from internal company CI/CD pipelines. Each entry in the database corresponds to a single Eiffel event.

By using authentic event data, the evaluation is grounded in realistic usage patterns, data volumes, and graph structures. This helps ensure that the system is assessed in conditions that closely resemble its intended deployment environment, making the results more representative of real-world behavior.

Although the total number of events across all available repositories exceeds 4 billion, the main data set used for this project comprises approximately 45 million events from a single event repository.

5.1.2 RQ1.1: How do the evaluated technologies perform in terms of throughput and resource usage, over time?

To help answer RQ1.1, it was tested how well the different evaluated tools can process and detect patterns as the number of events increases. The size of the test data was chosen based on practical runtime considerations, and the entire content of one of Ericsson's event repositories, comprising 45 million events, was deemed sufficiently large to yield meaningful results. The goal of this test was to verify how the tools perform while aggregating large amounts of event data, and not how multiple or more complex patterns affect performance, and for that reason the artifact created pattern (see Section 4.3.1) was chosen as it is the smallest pattern.

To evaluate scalability over time, all tracked metrics were used: throughput, CPU usage, memory usage, and disk usage. Throughput reflect the system's ability to maintain a high rate of event processing, which is critical in real-time CI/CD environments. CPU usage helps reveal whether the system makes effective use of available compute resources or suffers from underutilization or bottlenecks. Memory and disk usage were monitored to evaluate how efficiently each system handles increasing data volumes, helping determine whether resource consumption remains sustainable over time. Together, these metrics provide a comprehensive view of each system's performance under load.

5.1.3 RQ1.2: To what extent do the technologies exhibit consistent performance across repeated, identical runs?

To complement the primary scalability evaluation in Section 5.1.2, a secondary testing approach was designed with the purpose of assessing the consistency and reliability of the systems under reproducible and controlled conditions. Although a single large-scale test can reveal how a system performs under heavy load, it does not account for potential variability between runs. By executing a series of smaller, repeated tests under identical configurations, it becomes possible to observe whether the system maintains stable performance, identify any irregularities or outliers, and gain a deeper understanding of its behavior. This approach strengthens the validity of the overall evaluation by investigating if the observed results are isolated anomalies or reflect a consistent performance pattern.

To do this, 30 separate tests were conducted for the evaluated tools, where each test involved streaming 1 million events into each respective system. By isolating each run, the tests allowed for a consistent basis of comparison and enabled the detection of any unexpected performance degradation, caching effects, or system-level anomalies that may emerge across multiple insertions.

To assess the stability of each system, the primary metric used was throughput. This metric captures how well each system can ingest events under identical conditions. For each of the 30 test runs, the average events per second (EPS) was recorded and used as the basis for evaluating performance consistency.

The evaluation focuses on the mean EPS across all runs and the variance between them. A low standard deviation indicates stable and predictable performance, while high variability may suggest sensitivity to internal inconsistencies when handling insertions.

By comparing the distribution of results across all runs, the analysis aims to determine whether the systems deliver not only high throughput but also reliably repeatable behavior, which is an essential property for systems deployed in production environments where predictable performance is critical.

5.1.4 RQ1.3: How accurately do the technologies reproduce expected pattern matches compared to a known ground truth?

For a system to function correctly when handling pattern detection, it is important not only that it can process large volumes of data but also that it can do so accurately by correctly identifying all matching patterns. To evaluate the correctness of pattern detection, the results from the evaluated tools were compared to those produced by Eiffel Intelligence, which has been reliably used internally at Ericsson for several years and is used as the ground truth.

To evaluate the accuracy of pattern detection, the three available patterns in Eiffel Intelligence were used, namely the artifact created pattern, the artifact published pattern, and the confidence level reached pattern (see Section 4.3.1). These patterns were included in all systems to see if the output notifications match that of Eiffel Intelligence.

The dataset used was the first 1.8 million events of the event repository. This number was deemed sufficient to show accuracy of the evaluated systems.

When notifications were sent, the event fulfilling a pattern was logged. The results were then grouped into four categories: True positives, true negatives, false positives and false negatives. This breakdown gave a clear picture of how accurately each tool performed and whether any differences were caused by missed detections or by patterns being incorrectly triggered when they should not have been.

In order to compare the output from the tools to the patterns found by Eiffel Intelligence, a script was used to match the sets of events involved in each notification log.

5.1.5 RQ2: How does the number of active patterns affect throughput on the evaluated technologies?

To help answer RQ2, repeated tests with increasing amounts of patterns active was done for the evaluated tools as well as for Eiffel Intelligence. The motivation behind this approach is to see how the number of patterns active affect the performance of the systems. The dataset used for each test had a volume of 1 million events, which was deemed large enough to give meaningful results.

The patterns used were limited to the three available patterns in Eiffel Intelligence: artifact created pattern, artifact published pattern, and confidence level reached pattern (see Section 4.3.1). To achieve a higher amount of active patterns, each pattern was duplicated in such a way that each one filtered for a unique identifier attribute for known events that would fulfill a pattern. This leads to each trigger (in graph databases) or subscription (in Eiffel Intelligence) to be unique. The distribution of pattern types was uniform, which was chosen because it reflects the current distribution used at Ericsson.

The test was evaluated based on throughput, as this metric directly reflects whether the system can maintain sufficient event processing capacity as the number of active patterns increases. The number of patterns was incrementally increased, starting from zero, until the system's throughput fell below the required threshold of 100 events per second. This approach makes it possible to identify the scalability limits of each technology.

5.2 Results

This chapter presents the results of the tests conducted to evaluate the behavior of Neo4j, Memgraph, and Flink, to be able to answer the two research questions. It includes a series of plots and graphs that visualize the outcome from the tests.

5.2.1 RQ1.1: How do the evaluated technologies perform in terms of throughput and resource usage, over time?

The scalability evaluation revealed that Apache Flink is not suitable as an event aggregator for Eiffel events in its current form. During this test, Flink consistently crashed early due to excessive memory consumption. This behavior is hypothesized to be attributed to Flink's internal data stream handling and chaining mechanisms, which likely buffer large volumes of events in memory as various event types enter in different order, generating new unfinished patterns. This causes a sharp increase in memory usage, which quickly grows to 100 percent. Without fine-tuned backpressure management or early event pruning, Flink cannot sustain long-running aggregation processes at scale for this use case. As the performance is insufficient, no further tests were conducted on Flink.

In contrast, as shown in Figure 5.1, Memgraph exhibited significantly higher insert throughput than Neo4j during the early phase of the test (up to about 5000 seconds, or 10 million events). This aligns with expectations, as Memgraph's architecture is built around an in-memory storage engine that minimizes disk I/O, making it well-suited for high-speed data ingestion. This in-memory graph representation likely contributed to its strong early performance.

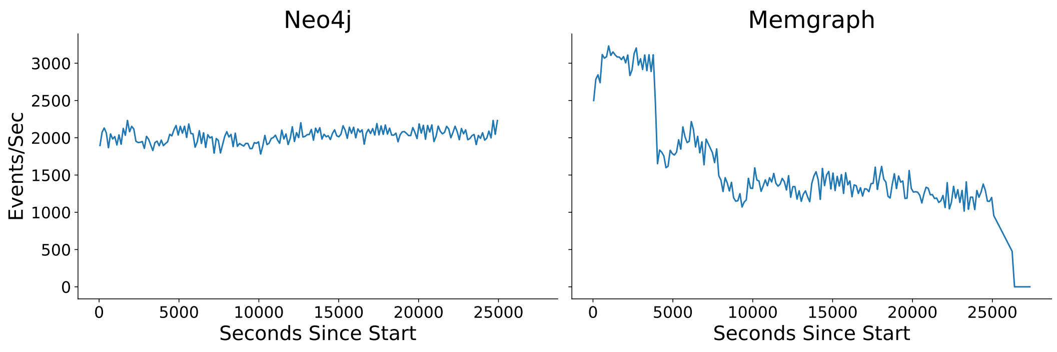


Figure 5.1: Throughput over time for Neo4j and Memgraph

However, after approximately 10 million inserted events, Memgraph's performance dropped sharply. The insert rate declined by about half, falling below that of Neo4j. Interestingly, this decline was accompanied by a reduction in CPU usage, shown in Figure 5.2, suggesting that the system has entered a throttled or degraded operational state. This test was investigated and redone several times, but it resulted in the same phenomenon each time. Although the exact cause of this behavior is unclear, it is hypothesized that internal resource limits or memory management strategies (e.g., garbage collection, snapshot flushing, or lock contention) contributed to the performance degradation.

Moreover, Memgraph crashed entirely after around 37 million events were inserted, when the available system memory was exhausted, which can be seen in Figure 5.3. This highlights a key limitation of its in-memory architecture: while it enables fast ingestion, it also constrains scalability by tying storage capacity directly to available RAM. Once the system reaches memory saturation, it cannot continue operating unless additional memory is provided or time-to-live mechanisms are implemented. Comparatively, Neo4j does demonstrate a constant memory usage over time, which is expected.

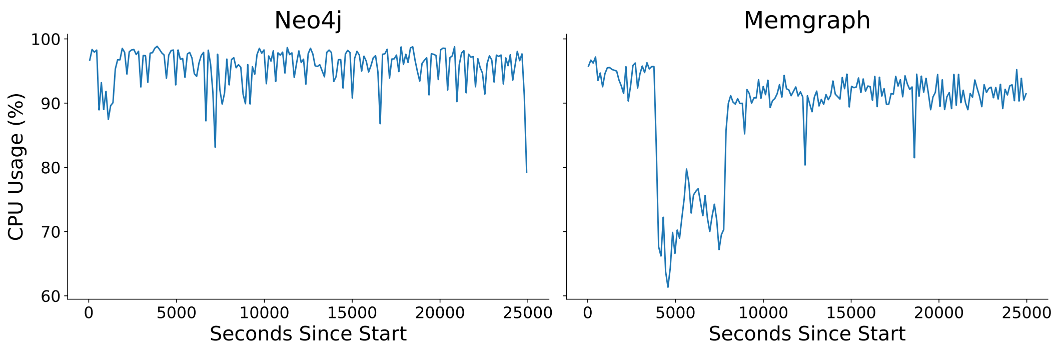


Figure 5.2: CPU usage over time for Neo4j and Memgraph

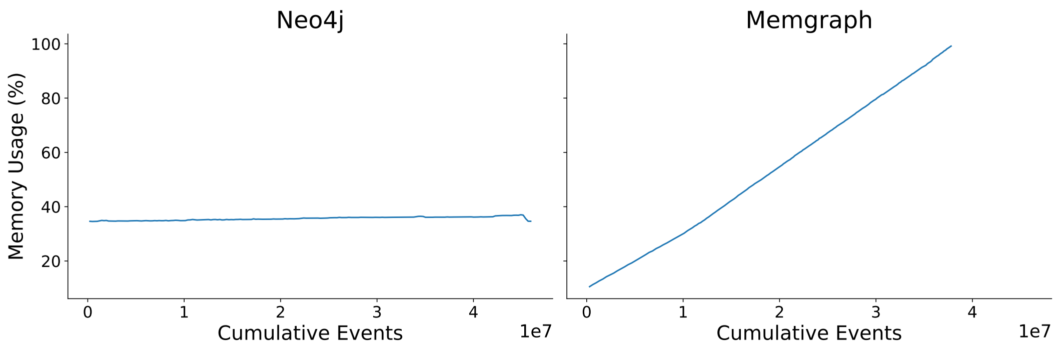


Figure 5.3: Memory usage over cumulative events for Neo4j and Memgraph

In contrast to Memgraph, Neo4j demonstrated stable throughput throughout the entire test, with no visible performance degradation, even beyond 45 million inserted events. This behavior can likely be attributed to its on-disk storage model, which, while slower, ensures that memory constraints is not a hard upper limit. Neo4j's durability come at the cost of throughput, but the scalability results suggest it is more robust for long-term aggregation workloads where data persistence and system stability are critical. Comparison between Memgraph and Neo4j in cumulative events over time is shown in Figure 5.4.

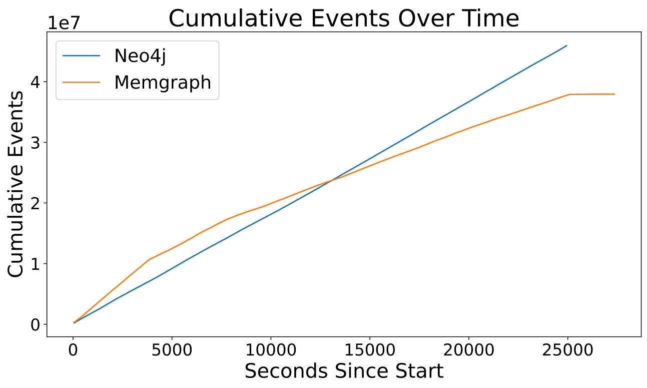


Figure 5.4: Cumulative events over time for Neo4j and Memgraph

In terms of disk usage, an interesting observation emerges from Figure 5.5. Neo4j shows a near-linear and predictable increase in disk consumption as more events are inserted, which aligns with its on-disk storage architecture. This behavior demonstrates efficient disk management and a consistent storage overhead per event, reinforcing Neo4j's suitability for persistent, large-scale workloads.

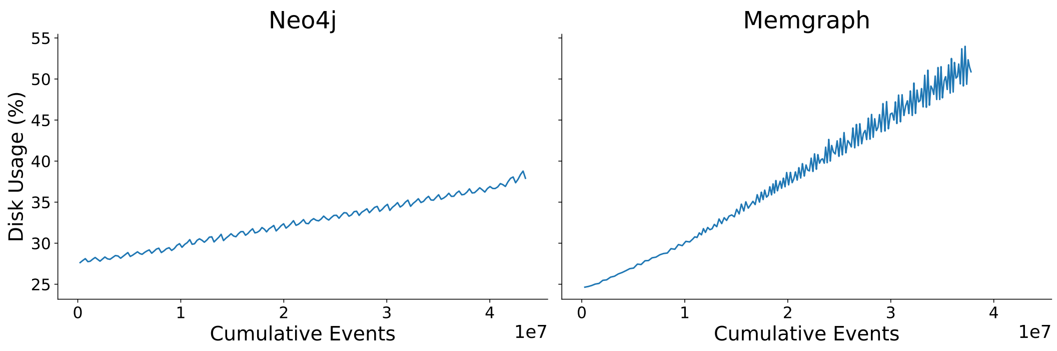


Figure 5.5: Disk usage over cumulative events for Neo4j and Memgraph

Surprisingly, despite being designed as an in-memory database, Memgraph also exhibits a proportional relationship between the number of stored events and disk usage. However, the slope of this relationship is noticeably steeper than that of Neo4j, indicating a higher disk consumption rate per inserted event. While the exact breakdown of disk allocations has not been investigated in detail, it is likely that Memgraph's higher disk usage stems from internal mechanisms such as logging, snapshotting, and auxiliary data structures required to support performance and ensure recoverability. As a result, Memgraph ends up allocating significantly more disk space than Neo4j for the same volume of data, which may be a consideration for deployments with limited storage resources.

RQ1.1: Neo4j demonstrated stable and scalable performance across the large dataset, maintaining consistent throughput, memory, and disk usage. Memgraph showed superior early throughput but entered a degraded state, and finally suffered from eventual crashes due to its in-memory architecture. Flink failed to scale. Overall, Neo4j proved more suitable for long-term pattern detection on large volumes of event data.

5.2.2 RQ1.2: To what extent do the technologies exhibit consistent performance across repeated, identical runs?

To complement the scalability evaluation, a stability test was conducted to assess whether the systems could deliver consistent performance under repeatable conditions. This type of evaluation is important to determine if observed throughput is reliable or influenced by internal variability across runs.

Each database, Neo4j and Memgraph, was evaluated on 30 identical test runs, where 1 million events were inserted per run. The test environment and configuration were reset before each run to ensure isolation and reproducibility. The primary goal was to evaluate whether the systems could maintain consistent throughput over time when subjected to the same workload repeatedly.

Neo4j showed stable performance, with throughput staying consistent across all 30 runs. The coefficient of variation was only 1.7%. Figure 5.6 illustrate that event ingestion was smooth and largely unaffected by internal variability.

Memgraph, while generally performant, exhibited more variability between runs. The coefficient of variation was 7.6%. Although its average throughput was higher than Neo4j, the larger fluctuations between runs suggest increased sensitivity to internal factors. This is further visualized in Figure 5.7, where Neo4j's average throughput distribution between runs is sharply centered, while Memgraph's has a broader spread.

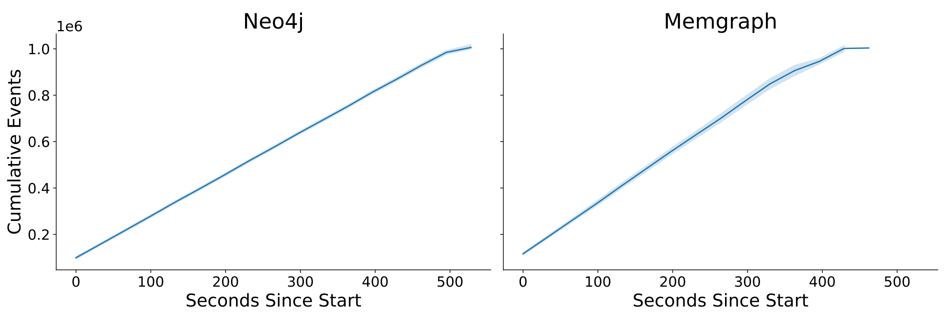


Figure 5.6: Cumulative events over time for Neo4j and Memgraph

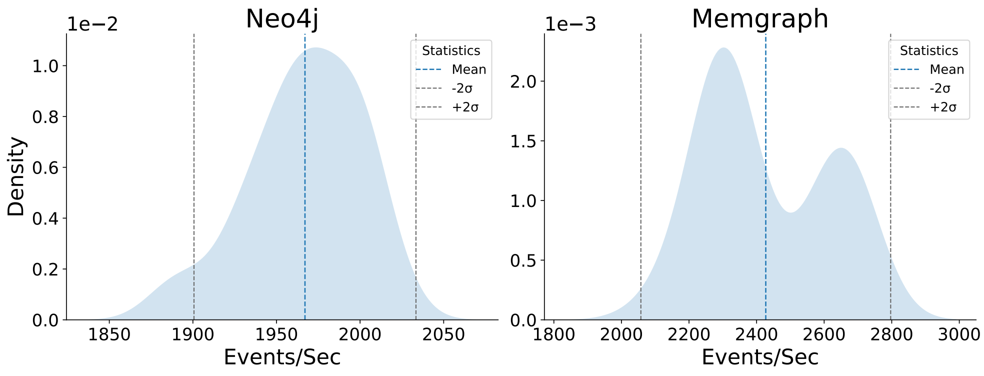


Figure 5.7: Density of throughput for Neo4j and Memgraph

RQ1.2: Neo4j demonstrated stable performance across repeated, identical test runs, maintaining consistent throughput and resource usage. In contrast, Memgraph exhibited greater variability, suggesting sensitivity to internal factors. While Memgraph achieved higher peak throughput, Neo4j's consistent behavior makes it more reliable for environments where predictability is essential.

5.2.3 RQ1.3: Validate accuracy

To verify the correctness of the implemented aggregation mechanisms in Neo4j and Memgraph, a dedicated validity test was conducted. The goal was to assess whether the systems could accurately detect known patterns when compared to Eiffel Intelligence.

The results showed a perfect match between the outputs of the graph databases and Eiffel Intelligence. Both Neo4j and Memgraph correctly identified all valid pattern instances, without generating any false positives or false negatives. Table 5.1 presents a summary of the results for all patterns across the three systems (note that no artifact published patterns were found in the dataset, seen in Table 5.1b).

RQ1.3: Both Neo4j and Memgraph correctly replicated the output of Eiffel Intelligence, detecting all intended patterns without any false positives or negatives. This confirms that the trigger-based mechanism in the graph databases is capable of accurately performing event aggregation. The results validate the correctness of using graph-based systems for pattern detection, showing they can faithfully represent the logic used in the existing solutions of Eiffel Intelligence.

Table 5.1: Accuracy of Neo4j and Memgraph compared to Eiffel Intelligence

(a) Artifact Created Pattern

	Detected	**Passed**
Actual Pattern	2127	0
Not a Pattern	0	1797873

Neo4j

	Detected	**Passed**
Actual Pattern	2127	0
Not a Pattern	0	1797873

Memgraph

(b) Artifact Published Pattern

	Detected	**Passed**
Actual Pattern	0	0
Not a Pattern	0	1800000

Neo4j

	Detected	**Passed**
Actual Pattern	0	0
Not a Pattern	0	1800000

Memgraph

(c) Confidence Level Reached Pattern

	Detected	**Passed**
Actual Pattern	5546	0
Not a Pattern	0	1794454

Neo4j

	Detected	**Passed**
Actual Pattern	5546	0
Not a Pattern	0	1794454

Memgraph

5.2.4 RQ2: How does the number of active patterns affect throughput on the evaluated technologies?

This test was designed to evaluate how increasing the number of active patterns impacts system performance in the evaluated tools. A system that can scale with many concurrent patterns is crucial in real-world CI/CD environments, where several users may be creating aggregation rules. The goal was to assess whether performance, measured primarily by throughput, remains acceptable (>100 EPS) as the number of active patterns increases.

During tests of Memgraph's trigger system under high trigger volume (>100 triggers), the database maintained high transaction throughput (1000-2000 EPS) and continued accepting writes. However, triggers stopped firing consistently, despite transactions being committed. This was observed when the server stopped being notified when increasing the amount of triggers. The database also unexpectedly crashed briefly at times during the tests. The lack of error messages or logs indicates a silent failure of the trigger execution mechanism, compromising system correctness. This test was repeated several times with changes in amount of triggers active, but the actions of the system did not change. This behavior makes Memgraph unsuitable for use cases requiring reliable event-based reaction logic at scale. The reason behind this behavior has not been investigated thoroughly, but it is hypothesized to be trigger-logic piling up in a queue and then being wiped when the database crashes.

Neo4j demonstrated high performance at low trigger counts (0-100), achieving higher average EPS than Eiffel Intelligence. However, its performance degraded non-linearly as more triggers were added. By the time 800 triggers were active, throughput had dropped to approximately 80 EPS, which is below the throughput threshold (100 EPS). This steep performance degradation suggests that Neo4j incurs considerable per-trigger overhead.

In contrast, Eiffel Intelligence starts at a lower throughput and also experiences an initial drop in throughput as the subscription count increases. However, the performance decline slows down as more triggers are added, resulting in a more gradual performance decrease. This is likely a result of Eiffel Intelligence using two steps to detect patterns as described in 2.4.1. The subscriptions are only applied to the aggregated objects created by the aggregation rules, which decreases the number of filtering operations compared to Neo4j and Memgraph, which evaluate every incoming event against all active triggers.

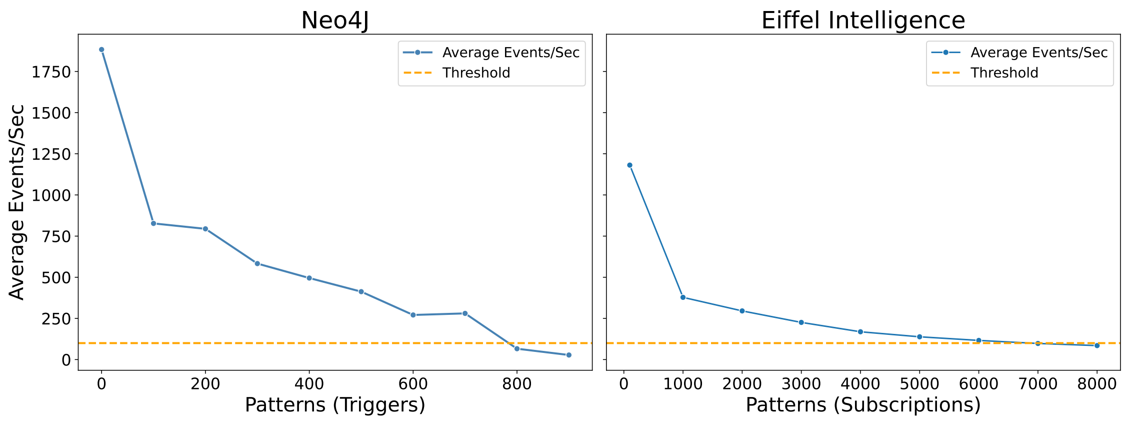


Figure 5.8: Throughput as number of patterns increase for Neo4j and Eiffel Intelligence

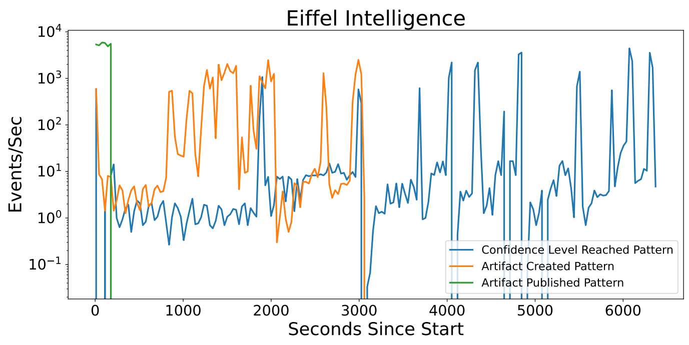


Figure 5.9: Throughput over time for Eiffel Intelligence with 4000 Subscriptions

Figure 5.9 shows Eiffel Intelligence's behavior during pattern detection, demonstrating significant variation in throughput over time. Sharp drops in performance occur during periods when many trigger events arrive, as these initiate the aggregation process and require the system to evaluate all subscriptions against each aggregated object of the same type. When the throughput is high, it indicates that no or a few trigger events are arriving. This explains why the artifact published pattern is processed quickly, as there are no occurrences of this pattern in the data. The throughput for Neo4j is comparatively more stable, as it performs checks regardless of whether matches are found.

RQ2: Neo4j scaled decently, but throughput degraded significantly with each added trigger, falling below target levels at 800 triggers. In comparison, Eiffel Intelligence scaled to 7000 active subscriptions, showing a relatively minor impact from each pattern active. Memgraph is found to be unsuitable as an aggregator at scale, due to triggers becoming unreliable, with occasional crashes, when over 100 triggers.

Summary: This chapter presented the results of the performance and scalability tests for Neo4j, Memgraph, and Flink, compared to Eiffel Intelligence, addressing the research questions. Neo4j demonstrated stable throughput and accuracy across large datasets, while Memgraph showed high early performance but suffered from stability issues and crashes. Flink proved unsuitable due to excessive memory use. While Neo4j's throughput declined faster than Eiffel Intelligence's with more active patterns, however, Neo4j supports more complex patterns and is more user-friendly.

6 Discussion

This chapter explores the practical implications of the results in Section 6.1, focusing on how the findings inform system design, architectural choices, and real-world usage. It is structured around the research questions and highlights trade-offs, limitations, and considerations relevant to applying the evaluated technologies in operational contexts. The chapter concludes with reflections on the evaluation method and its potential impact on the results in Section 6.2.

6.1 Results

The evaluation results raise considerations for the practical use of graph-based systems in event aggregation scenarios, including throughput, resource usage, and pattern impact. Beyond verifying whether the systems can aggregate, the findings help clarify how they behave under operational constraints, what trade-offs they introduce, and how they respond to different patterns.

6.1.1 RQ1: What technologies can aggregate events while maintaining sufficient performance and efficient resource usage over time?

The results from RQ1 indicate that both Neo4j and Memgraph fulfill the functional and performance-related requirements to operate as an event aggregation system using a low number of triggers. They support both the current scenarios and more complex patterns that are not possible with Eiffel Intelligence. Meanwhile, Flink fails to meet the performance requirements. Through targeted testing, the graph database systems demonstrated high throughput, mostly consistent behavior across repeated executions, with some Memgraph runs exceeding regular performance and correct detection of event patterns using trigger-based logic.

Takeaway 1. Apache Flink quickly runs out of available memory, preventing it from maintaining sufficient performance for event aggregation.

Memgraph demonstrated clear advantages in terms of early throughput. Its in-memory design allows for rapid insertion speeds during the initial phase of operation, making it suitable for use cases where high-speed ingestion is critical. However, performance dropped significantly as memory usage increased, and the system eventually became unstable once physical memory limits were reached. This imposes a scalability constraint that must be actively managed. In addition to memory saturation, Memgraph also exhibited aggressive disk usage over time, despite being an in-memory database. As a result, deployments must also include mechanisms to monitor or limit disk usage to avoid exhaustion in high-volume scenarios.

Takeaway 2. Memgraph offers high ingestion speed but requires active memory and disk management. Use of time-to-live (TTL) policies and disk usage monitoring is necessary to maintain stable operation over time.

Neo4j exhibited slightly lower throughput than memgraph. The insert rate remained largely consistent, even at higher event volumes. Disk usage increased as new events were inserted, whilst memory usage was constant. This behavior is a direct result of its on-disk architecture, which decouples data storage from memory.

Finally, the validity test confirmed that both systems are capable of reliably detecting event patterns under realistic conditions. Each pattern was correctly found on live data streams, with no observed false positives or negatives. This confirms that the graph-based trigger mechanism is expressive and reliable enough to serve as the foundation for practical aggregation logic.

A drawback of Neo4j is that it requires storing all events within its database to enable querying for pattern detection, which can be excessive if the goal is specifically pattern matching and the data is already stored elsewhere. Although migrating the data to Neo4j is possible, this may not be desirable. In such cases, Eiffel Intelligence has an advantage, as it can operate without storing the entire dataset.

Takeaway 3. Neo4j provides robust and predictable performance. It handles increasing data volumes gracefully and is well-suited to systems requiring operational stability over time.

In summary, the results show that both Neo4j and Memgraph are viable technologies for event aggregation. They offer different operational trade-offs: Memgraph provides high-speed ingestion but must be paired with mechanisms to limit memory pressure; Neo4j offers stable, long-running operation at the cost of lower throughput. The choice between them depends primarily on workload characteristics and system constraints.

6.1.2 RQ2: How does the number of active patterns affect throughput on the evaluated technologies?

Handling complex event patterns at scale is a critical challenge for event aggregation systems in real-world CI/CD environments. Beyond raw performance, systems must support flexible pattern definitions and adapt quickly to changing requirements while managing multiple patterns simultaneously.

Memgraph demonstrated limitations in scalability when handling multiple concurrent patterns. Although it passed validation tests for individual patterns in RQ 1.3, Memgraph stopped detecting patterns as the number of patterns increased. This suggests that Memgraph is not suitable for handling a large number of patterns, but may be appropriate for scenarios where very high throughput is needed and the number of patterns remains low.

Takeaway 4. Memgraph's inability to support a high number of concurrent subscriptions limits its use in event aggregation scenarios.

Eiffel Intelligence demonstrated a clear advantage in handling a large number of subscriptions while maintaining throughput above the threshold compared to Neo4j. This is due to Eiffel Intelligence creating only one aggregated object per trigger event. This requires one filtering operation for each event entering the system, to see if the entering event is a trigger event. The chosen information flow in Neo4j (see Section 4.4.1), perform this filtering operation for each event on all active triggers. Consequently, Neo4j experiences a greater performance impact as the number of active triggers increases.

The choice to adopt this trigger-per-consumer approach in Neo4j was motivated by the desire to provide a user-friendly and modular trigger creation process, as discussed in Section 4.4.6. While the performance limitations were anticipated during the architectural design, it remains uncertain whether restructuring the internal logic to reduce redundancy would be beneficial. Nonetheless, such restructuring would likely yield significant improvements. One such improvement could involve adopting a more optimized architecture, such as a discrimination network, as described in Section 4.4.5, which would allow shared filtering and querying logic across multiple consumers and theoretically significantly improve scalability.

Takeaway 5. Eiffel Intelligence scales efficiently with high subscription volumes, maintaining performance where Neo4j begins to degrade. Its filtering-first architecture may serve as a design reference for enhancing other aggregation systems.

A trigger in Neo4j offers significantly greater flexibility and can handle almost any pattern independently. In contrast, Eiffel Intelligence requires setting up a separate backend instance to detect a pattern for each new graph structure. Consequently, defining new aggregation rules in Eiffel Intelligence is considerably more complex and requires great expertise, while for Neo4j, it only involves inserting a new trigger. Additionally, Neo4j's graph query language enables a wider range of users to define new patterns with relative ease thanks to its intuitive and expressive design. Furthermore, Neo4j is capable of handling a wider variety of different patterns, such as the ones described in Section 4.3.1 and 4.7, which are impossible for Eiffel Intelligence.

Takeaway 6. Neo4j provides a more flexible and intuitive pattern definition workflow, supporting rapid adaptation of aggregation logic.

In summary, no single system offered a perfect balance between flexibility, scalability, and performance. Instead, each system showed strengths aligned with different priorities: Neo4j excelled in adaptability and ease of use for complex pattern definitions, while Eiffel Intelligence delivered strong performance and scalability under large volumes of subscriptions. Memgraph offered the highest throughput with a low number of active triggers, but it did not operate correctly as the number of patterns increased.

6.2 Threats to Validity

This section discusses potential threats to the reliability, replicability, and validity of the study. The threats are categorized as threats to internal, external, and construct validity.

6.2.1 Internal validity

In Eiffel Intelligence, events are acknowledged before they have been fully processed, which can lead to optimistic results. This is based on the observation that subscription notifications may continue to arrive several seconds after the message queue has been emptied, indicating that internal processing of the final events is still ongoing. Consequently, using acknowledgments as a proxy for processing time in Eiffel Intelligence may slightly underestimate the total time required. This is not an issue for the graph databases, where events are acknowledged only after successful insertion, which implies that all required processing steps have already been completed.

Two of the evaluated tools are implemented in Java, Eiffel Intelligence and Neo4j; hence, they operate on the Java Virtual Machine (JVM). In these cases, it is important to consider that the JVM undergoes an initial warm-up phase, during which performance may be reduced due to just-in-time compilation and runtime optimizations [59]. In this study, no explicit JVM warm-up was conducted before measurements began. However, since the tests are based on sustained workloads and no noticeable performance differences were observed, the impact of JVM warm-up is expected to have negligible effect.

The measurements evaluate the system as a whole rather than isolating the individual tools responsible for the event aggregation. While this provides a realistic evaluation of end-to-end performance, potential bottlenecks in other components may affect the results. Although computational resources are shared among system parts, the extent of their influence on performance metrics is not fully known.

6.2.2 External validity

Flink was excluded after the scalability evaluation, as it did not meet the requirements during testing. There is a possibility that Flink could have been adapted to support the use cases. Doing so would have required further configuration and optimization to reduce the number of simultaneously active pattern instances. However, this was not pursued within the scope of the study.

The evaluation was conducted on internal data provided by Ericsson, which is not publicly available. While this data is the most realistic for testing, as it best represents real usage, it also limits reproducibility. Reproducing the results would likely require using either artificial data or data from a different pipeline, which may have different characteristics, for example, be more densely connected.

6.2.3 Construct validity

For evaluating the Eiffel Intelligence setup, all data was initially published to an event repository and then inserted into its database to ensure that all events existed to enable upstream searches. However, this does not represent real-world usage, where data is sent to both Eiffel Intelligence and the event repository simultaneously. In practice, the lower insertion frequency prevents connected events from appearing immediately one after another, allowing smoother synchronization. This approach was not viable for the evaluation in this thesis, as events were inserted at a significantly higher frequency. Consequently, linked events might not yet be available when Eiffel Intelligence performs searches due to differences in processing speed between the two systems. Therefore, Eiffel Intelligence benefits from pre-existing data in the event repository, resulting in favorable metrics that do not fully reflect real-world performance. However, since Eiffel Intelligence performs the primary function of event aggregation, the evaluation still provides meaningful insights into the core functionality being investigated.

This thesis would have benefited from evaluating a wider range of patterns to better understand how pattern complexity affects performance. In particular, larger patterns that require more complex queries are likely to have a greater impact on system performance than those evaluated. Additionally, the frequency with which patterns are found in the data might impact how triggers affect performance, which suggests the need to test multiple patterns of similar complexity.

Summary: This chapter emphasized the practical implications of the evaluation, highlighting Memgraph's strong early throughput coupled with memory and disk management challenges, Eiffel Intelligence's scalability advantages with large numbers of active patterns thanks to its two-step filtering. Neo4j demonstrated stable and predictable performance with greater flexibility in pattern definition, including the ability to handle complex patterns that Eiffel Intelligence cannot support. In addition, it reviewed the validity and reproducibility of the methodology.

7 Conclusion

This thesis set out to improve the usability, flexibility, and scalability of event aggregation in CI/CD by addressing the limitations of Eiffel Intelligence. By designing and evaluating alternative technologies such as Neo4j, Memgraph, and Apache Flink, it explores how novel approaches using graph databases and declarative query languages compare to existing solutions. This chapter summarizes the key achievements of the work, reflects on how the objectives were accomplished (Section 7.1), and concludes with a discussion of broader implications and potential directions for future research (Section 7.2).

7.1 Overview of Objectives and Contributions

This section summarizes the main objectives of the thesis and how they were achieved. It also highlights the key contributions made through design, implementation, and evaluation of the event aggregation techniques.

7.1.1 Objective 1: Declarative specification of aggregation patterns

Objective Ob1 aimed to enable declarative specification of event aggregation patterns using expressive graph query constructs. By proposing the use of graph databases and applying graph query languages, we successfully simplified the creation of aggregation patterns, making the process more intuitive.

7.1.2 Objective 2: Prototype implementation and validation

The aim of Objective Ob2 was to implement a prototype system for event aggregation and validate its functionality against realistic CI/CD scenarios. We replayed historical Eiffel event data from real-world pipelines to assess the system's ability to successfully process and aggregate events, proving its practical applicability to accurately detect patterns.

7.1.3 Objective 3: Performance evaluation of aggregation techniques

The goal of Objective Ob3 was to provide a comparative evaluation of alternative aggregation approaches under increasing event loads. By simulating varying throughput scenarios, we compared system performance in terms of processing speed and resource consumption. Revealing that graph databases performed well initially but slowed as patterns increased; the CEP solution crashed under load.

7.1.4 Contributions

We propose a new architecture based on graph databases that supports multiple aggregation rules within a single instance in Contribution Co1, reducing deployment complexity and enhancing scalability. Beyond these improvements, the use of graph databases has the following added values:

- **Pipeline aggregation**: The same database can function both as an event aggregator and as an analytics engine for historical event data.
- **Legacy support**: It can handle backwards compatibility between versions of event protocols, enabling organizations to adopt a centralized aggregation solution.

We implemented and tested three distinct systems under varying workloads in Contribution Co2, offering a comparative perspective on how graph databases and CEP engines perform in practical CI/CD scenarios.

In Contribution Co3, one of the key features of our solution is to define aggregation rules declaratively using a graph query language, making pattern specification more expressive, reducing limitations in which patterns can be defined. An added value of declaratively defining aggregation rules with graph query languages is that it also makes it more intuitive for the users interacting with the system, compared to the current solution.

We developed custom triggers in Neo4j and Memgraph that enable incremental and reactive detection of event patterns as they arrive in Contribution Co4, offering a novel mechanism for live event stream processing.

7.2 Future Work

While this thesis confirms the viability of centralized and scalable trigger-based event aggregation using declarative graph queries, several limitations were revealed that motivate further research and development.

7.2.1 Performance optimization at scale

A key challenge identified is the performance degradation that occurs as the number of active aggregation rules increases. To address this, future work should explore the use of a discrimination network, a well-established concept from rule-based systems such as RETE (see Section 4.4.5). By splitting up the logic needed for pattern detection and notification as a decision tree, common filtering and query logic across rules can be evaluated once and reused across multiple patterns. This would theoretically significantly reduce redundant computation, improve trigger evaluation efficiency, and enable the system to handle a larger rule set without aggressive performance degradation.

7.2.2 Usability and developer experience

Another important area for future work is usability. This thesis builds on the assumption that declarative graph query languages are more intuitive and maintainable than the current rule definitions used in Eiffel Intelligence, which rely on JMESPath. While no formal user evaluation were conducted, limited informal feedback was gathered through discussions with engineers during development. These interactions suggested that even users with limited graph database experience found the system's pattern definition process approachable. However, these observations were not systematically evaluated and are therefore not reported as part of the results.

Conducting a usability study would help validate or challenge this assumption. The motivation stems from the need to understand how users interact with the system when defining and managing aggregation rules, and whether the switch to declarative graph-based rules leads to measurable improvements in usability. Gathering empirical data from such a study would guide future design decisions, such as whether to introduce declarative graph query languages as aggregation rules, or not.

Bibliography

[1] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. “Efficient pattern matching over event streams”. In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 2008, pp. 147–160.

[2] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. “Distributed Evaluation of Subgraph Queries Using Worst-case Optimal and Low-Memory Dataflows”. In: *Proc. VLDB Endow.* 11.6 (2018), pp. 691–704.

[3] Renzo Angles. “The Property Graph Database Model”. In: *AMW*. Vol. 2100. CEUR Workshop Proceedings. CEUR-WS.org, 2018.

[4] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. “Foundations of modern query languages for graph databases”. In: *ACM Computing Surveys (CSUR)* 50.5 (2017), pp. 1–40.

[5] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. “Foundations of Modern Query Languages for Graph Databases”. In: *ACM Comput. Surv.* 50.5 (2017), 68:1–68:40.

[6] Apache Flink. *Apache Flink Documentation (Nightly)*. Accessed: 2025-04-23. 2025. URL: `https://nightlies.apache.org/flink/flink-docs-master/docs/`.

[7] S.A.I.B.S. Arachchi and Indika Perera. “Continuous Integration and Continuous Delivery Pipeline Automation for Agile Software Project Management”. In: *2018 Moratuwa Engineering Research Conference (MERCon)*. 2018, pp. 156–161. DOI: `10.1109/MERCon.2018.8421965`.

[8] Bradley R. Bebee, Rahul Chander, Ankit Gupta, Ankesh Khandelwal, Sainath Mallidi, Michael Schmidt, Ronak Sharda, Bryan B. Thompson, and Prashant Upadhyay. “Enabling an Enterprise Data Management Ecosystem using Change Data Capture with Amazon Neptune”. In: *ISWC (Satellites)*. Vol. 2456. CEUR Workshop Proceedings. CEUR-WS.org, 2019, pp. 189–192.

[9] Gábor Bergmann, Ákos Horváth, István Ráth, and Dániel Varró. “A benchmark evaluation of incremental pattern matching in graph transformation”. In: *International Conference on Graph Transformation*. Springer. 2008, pp. 396–410.

[10] Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler. “Practice of Streaming Processing of Dynamic Graphs: Concepts, Models, and Systems”. In: *IEEE Trans. Parallel Distributed Syst.* 34.6 (2023), pp. 1860–1876.

[11] Maciej Besta, Robert Gerstenberger, Emanuel Peter, Marc Fischer, Michal Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. “Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries”. In: *ACM Comput. Surv.* 56.2 (2024), 31:1–31:40. DOI: 10.1145/3604932.

[12] R. Bhargavi, Vijay Vaidehi, P. T. V. Bhuvaneswari, P. Balamuralidhar, and M. Girish Chandra. “Complex Event Processing for object tracking and intrusion detection in Wireless Sensor Networks”. In: *ICARCV*. IEEE, 2010, pp. 848–853.

[13] Bibek Bhattarai, Hang Liu, and H. Howie Huang. “CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching”. In: *SIGMOD Conference*. ACM, 2019, pp. 1447–1462.

[14] Björn Bringmann and Siegfried Nijssen. “What Is Frequent in a Single Graph?” In: *PAKDD*. Vol. 5012. Lecture Notes in Computer Science. Springer, 2008, pp. 858–863.

[15] Mihai Capota, Tim Hegeman, Alexandru Iosup, Arnau Prat-Pérez, Orri Erling, and Peter A. Boncz. “Graphalytics: A Big Data Benchmark for Graph-Processing Platforms”. In: *Proceedings of the Third International Workshop on Graph Data Management Experiences and Systems, GRADES 2015, Melbourne, VIC, Australia, May 31 - June 4, 2015*. Ed. by Josep Lluís Larriba-Pey and Theodore L. Willke. ACM, 2015, 7:1–7:6. DOI: 10.1145/2764947.2764954.

[16] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. “Apache Flink™: Stream and Batch Processing in a Single Engine”. In: *IEEE Data Eng. Bull.* 38.4 (2015), pp. 28–38.

[17] Otávio M de Carvalho, Eduardo Roloff, and Philippe OA Navaux. “A Survey of the State-of-the-art in Event Processing”. In: *Proceedings of the 11th Workshop on Parallel and Distributed Processing (WSPPD)*. 2013, p. 35.

[18] Stefano Ceri, Anna Bernasconi, Alessia Gagliardi, Davide Martinenghi, Luigi Bellomarini, and Davide Magnanimi. “PG-Triggers: Triggers for Property Graphs”. In: *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024*. Ed. by Pablo Barceló, Nayat Sánchez-Pi, Alexandra Meliou, and S. Sudarshan. ACM, 2024, pp. 373–385. DOI: 10.1145/3626246.3653386.

[19] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Frederick Reiss, and Mehul A. Shah. “TelegraphCQ: Continuous Dataflow Processing”. In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*. Ed. by Alon Y. Halevy, Zachary G. Ives, and AnHai Doan. ACM, 2003, p. 668. DOI: 10.1145/872757.872857.

[20] Chialin Chang, Bongki Moon, Anurag Acharya, Carter Shock, Alan Sussman, and Joel Saltz. “Titan: a high-performance remote-sensing database”. In: *Proceedings 13th international conference on data engineering*. IEEE. 1997, pp. 375–384.

[21] Sadaf Charkhabi, Peyman Samimi, Sikha Bagui, Dustin Mink, and Subhash C. Bagui. “Node Classification of Network Threats Leveraging Graph-Based Characterizations Using Memgraph”. In: *Comput.* 13.7 (2024), p. 171.

[22] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. “G-Miner: an efficient task-oriented graph mining system”. In: *EuroSys*. ACM, 2018, 32:1–32:12.

[23] Jane Cleland-Huang, Orlena Gotel, Jane Huffman Hayes, Patrick Mäder, and Andrea Zisman. “Software traceability: trends and future directions”. In: *FOSE*. ACM, 2014, pp. 55–69.

[24] Eiffel Community. *Eiffel*. `https://eiffel-community.github.io/`. Accessed: 2025-05-05. 2024.

[25] Gianpaolo Cugola and Alessandro Margara. “Processing flows of information: From data stream to complex event processing”. In: *ACM Comput. Surv.* 44.3 (2012), 15:1–15:62. DOI: `10.1145/2187671.2187677`.

[26] Gianpaolo Cugola and Alessandro Margara. “The Complex Event Processing Paradigm”. In: *Data Management in Pervasive Systems*. Data-Centric Systems and Applications. Springer, 2015, pp. 113–133.

[27] István Dávid, István Ráth, and Dániel Varró. “Foundations for Streaming Model Transformations by Complex Event Processing”. In: *Softw. Syst. Model.* 17.1 (2018), pp. 135–162.

[28] Alin Deutsch, Nadime Francis, Alastair Green, Keith W. Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Fred Zemke. “Graph Pattern Matching in GQL and SQL/PGQ”. In: *SIGMOD Conference*. ACM, 2022, pp. 2246–2258.

[29] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. “TigerGraph: A Native MPP Graph Database”. In: *CoRR* abs/1901.08248 (2019). arXiv: `1901.08248`.

[30] EsperTech. *Esper Reference Documentation*. Accessed: 2025-05-28. 2025. URL: `https://esper.espertech.com/release-8.0.0/reference-esper/html/`.

[31] Emanuele Falzone, Riccardo Tommasini, Emanuele Della Valle, Petra Selmer, Stefan Plantikow, Hannes Voigt, Keith Hare, Ljubica Lazarevic, and Tobias Lindaaker. “Semantic foundations of seraph continuous graph query language”. In: *arXiv preprint arXiv:2111.09228* (2021).

[32] Wenfei Fan, Jianzhong Li, Shuai Ma, Hongzhi Wang, and Yinghui Wu. “Graph Homomorphism Revisited for Graph Matching”. In: *Proc. VLDB Endow.* 3.1 (2010), pp. 1161–1172.

[33] Wenfei Fan, Xin Wang, and Yinghui Wu. “Incremental graph pattern matching”. In: *ACM Trans. Database Syst.* 38.3 (2013), p. 18.

[34] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. “Cypher: An Evolving Query Language for Property Graphs”. In: *SIGMOD Conference*. ACM, 2018, pp. 1433–1445.

[35] Ruediger Gad, Martin Kappes, Juan Boubeta-Puig, and Inmaculada Medina-Bulo. “Employing the CEP paradigm for network analysis and surveillance”. In: *AICT 2013* (2013), pp. 204–210.

[36] Emden R. Gansner and Stephen C. North. “An open graph visualization system and its applications to software engineering”. In: *Softw. Pract. Exp.* 30.11 (2000), pp. 1203–1233.

[37] Nikos Giatrakos, Alexander Artikis, Antonios Deligiannakis, and Minos N. Garofalakis. “Uncertainty-Aware Event Analytics over Distributed Settings”. In: *DEBS*. ACM, 2019, pp. 175–186.

[38] GitLab. *Get started with GitLab CI/CD*. Accessed: 2025-06-18. 2025. URL: `https://docs.gitlab.com/ci/`.

[39] Alejandro Grez, Cristian Riveros, and Martín Ugarte. “Foundations of complex event processing”. In: *CoRR, abs/1709.05369* (2017).

[40] Andrey Gubichev and Manuel Then. “Graph Pattern Matching - Do We Have to Reinvent the Wheel?” In: *Second International Workshop on Graph Data Management Experiences and Systems, GRADES 2014, co-loated with SIGMOD/PODS 2014, Snowbird, Utah, USA, June 22, 2014*. Ed. by Peter A. Boncz and Josep Lluís Larriba-Pey. CWI/ACM, 2014, 8:1–8:7. DOI: `10.1145/2621934.2621944`.

[41] Gregory Z. Gutin. “Acyclic Digraphs”. In: *Classes of Directed Graphs*. Springer Monographs in Mathematics. Springer, 2018, pp. 125–172.

[42] Ralf Hartmut Güting. “GraphDB: Modeling and Querying Graphs in Databases”. In: *VLDB*. Morgan Kaufmann, 1994, pp. 297–308.

[43] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. “Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together”. In: *SIGMOD Conference*. ACM, 2019, pp. 1429–1446.

[44] Safiollah Heidari, Yogesh Simmhan, Rodrigo N. Calheiros, and Rajkumar Buyya. “Scalable Graph Processing Frameworks: A Taxonomy and Open Challenges”. In: *ACM Comput. Surv.* 51.3 (2018), 60:1–60:53.

[45] Iman M. A. Helal and Ahmed Awad. “Online correlation for unlabeled process events: A flexible CEP-based approach”. In: *Inf. Syst.* 108 (2022), p. 102031.

[46] Jürgen Hölsch, Tobias Schmidt, and Michael Grossniklaus. “On the Performance of Analytical and Pattern Matching Graph Queries in Neo4j and a Relational Database”. In: *Proceedings of the Workshops of the EDBT/ICDT 2017 Joint Conference (EDBT/ICDT 2017), Venice, Italy, March 21-24, 2017*. Ed. by Yannis E. Ioannidis, Julia Stoyanovich, and Giorgio Orsi. Vol. 1810. CEUR Workshop Proceedings. CEUR-WS.org, 2017. URL: `https://ceur-ws.org/Vol-1810/GraphQ%5C_paper%5C_01.pdf`.

[47] Jürgen Hölsch, Tobias Schmidt, and Michael Grossniklaus. “On the Performance of Analytical and Pattern Matching Graph Queries in Neo4j and a Relational Database”. In: *EDBT/ICDT Workshops*. Vol. 1810. CEUR Workshop Proceedings. CEUR-WS.org, 2017.

[48] Susan Horwitz and Thomas W. Reps. “The Use of Program Dependence Graphs in Software Engineering”. In: *ICSE*. ACM Press, 1992, pp. 392–411.

[49] Ante Javor. *Memgraph vs. Neo4j: A Performance Comparison*. Accessed: 2025-05-09. 2024. URL: `https://memgraph.com/blog/memgraph-vs-neo4j-performance-benchmark-comparison`.

[50] Marc Kaepke and Olaf Zukunft. “A Comparative Evaluation of Big Data Frameworks for Graph Processing”. In: *Innovate-Data*. IEEE, 2018, pp. 30–37.

[51] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. “Benchmarking Distributed Stream Data Processing Systems”. In: *ICDE*. IEEE Computer Society, 2018, pp. 1507–1518.

[52] Robin Keskisärkkä. “Complex Event Processing under Uncertainty in RDF Stream Processing”. PhD thesis. Linköping University, Sweden, 2021.

[53] Timothy Kinsman, Mairieli Santos Wessel, Marco Aurélio Gerosa, and Christoph Treude. “How Do Software Developers Use GitHub Actions to Automate Their Workflows?” In: *CoRR* abs/2103.12224 (2021).

[54] Donald E. Knuth. *The Stanford GraphBase - a platform for combinatorial computing*. ACM, 1993. ISBN: 978-0-201-54275-2.

[55] Eleni Kougioumtzi, Antonios Kontaxakis, Antonios Deligiannakis, and Yannis Kotidis. “Towards creating a generalized complex event processing operator using FlinkCEP: architecture & benchmark”. In: *15th ACM International Conference on Distributed and Event-based Systems, DEBS 2021, Virtual Event, Italy, June 28 - July 2, 2021*. Ed. by Alessandro Margara, Emanuele Della Valle, Alexander Artikis, Nesime Tatbul, and Helge Parzyjegla. ACM, 2021, pp. 188–189. DOI: `10.1145/3465480.3467841`.

[56] Pradeep Kumar and H. Howie Huang. “GraphOne: A Data Store for Real-time Analytics on Evolving Graphs”. In: *ACM Trans. Storage* 15.4 (2020), 29:1–29:40. DOI: `10.1145/3364180`.

[57] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, Ying Zhang, Zhengping Qian, and Jingren Zhou. “Distributed Subgraph Matching on Timely Dataflow”. In: *Proc. VLDB Endow.* 12.10 (2019), pp. 1099–1112.

[58] Heng Lin, Xiaowei Zhu, Bowen Yu, Xiongchao Tang, Wei Xue, Wenguang Chen, Lufei Zhang, Torsten Hoefler, Xiaosong Ma, Xin Liu, Weimin Zheng, and Jingfang Xu. “ShenTu: processing multi-trillion edge graphs on millions of cores in seconds”. In: *SC*. IEEE / ACM, 2018, 56:1–56:11.

[59] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. “Don’t Get Caught in the Cold, Warm-up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 383–400.

[60] David C Luckham. *Event processing for business: organizing the real-time enterprise*. John Wiley & Sons, 2011.

[61] Zhuojing Ma, WangYang Yu, Xiaojun Zhai, and Menghan Jia. “A Complex Event Processing-Based Online Shopping User Risk Identification System”. In: *IEEE Access* 7 (2019), pp. 172088–172096.

[62] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. “GraphZero: A High-Performance Subgraph Matching System”. In: *ACM SIGOPS Oper. Syst. Rev.* 55.1 (2021), pp. 21–37.

[63] Memgraph. *Memgraph Documentation*. `https://memgraph.com/docs`. Accessed: 2025-05-12. 2025.

[64] Neo4j. *Awesome Procedures On Cypher*. `https://neo4j.com/labs/apoc/`. Accessed: 2025-06-03. 2025.

[65] Neo4j. *Cypher Introduction*. `https://neo4j.com/docs/cypher-manual/current/introduction/`. Accessed: 2025-04-12. 2025.

[66] Neo4j. *Neo4j documentation*. `https://neo4j.com/docs/`. Accessed: 2025-04-12. 2025.

[67] Neo4j. *What is Cypher*. `https://neo4j.com/docs/getting-started/cypher/`. Accessed: 2025-04-12. 2025.

[68] Guadalupe Ortiz, Adrian Bazan-Muñoz, Winfried Lamersdorf, and Alfonso García de Prado. “Evaluating the integration of Esper complex event processing engine and message brokers”. In: *PeerJ Comput. Sci.* 9 (2023), e1437.

[69] Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee. “A Design Science Research Methodology for Information Systems Research”. In: *J. Manag. Inf. Syst.* 24.3 (2008), pp. 45–77.

[70] Jaroslav Pokorný. “Graph Databases: Their Power and Limitations”. In: *Computer Information Systems and Industrial Management - 14th IFIP TC 8 International Conference, CISIM 2015, Warsaw, Poland, September 24-26, 2015. Proceedings*. Ed. by Khalid Saeed and Wladyslaw Homenda. Vol. 9339. Lecture Notes in Computer Science. Springer, 2015, pp. 58–69. DOI: `10.1007/978-3-319-24369-6\_5`.

[71] Jaroslav Pokornỳ. “Graph databases: their power and limitations”. In: *IFIP International Conference on Computer Information Systems and Industrial Management*. Springer. 2015, pp. 58–69.

[72] Miao Qiao, Hao Zhang, and Hong Cheng. “Subgraph Matching: on Compression and Computation”. In: *Proc. VLDB Endow.* 11.2 (2017), pp. 176–188.

[73] T. Ramalingeswara Rao, Soumya Kanti Ghosh, and Adrijit Goswami. “Mining user-user communities for a weighted bipartite network using spark GraphFrames and Flink Gelly”. In: *J. Supercomput.* 77.6 (2021), pp. 5984–6035.

[74] Daniel Ritter, Luigi Dell’Aquila, Andrii Lomakin, and Emanuele Tagliaferri. “OrientDB: A NoSQL, Open Source MMDMS”. In: *Proceedings of the The British International Conference on Databases 2021, London, United Kingdom, March 28, 2022*. Ed. by Holger Pirk and Thomas Heinis. Vol. 3163. CEUR Workshop Proceedings. CEUR-WS.org, 2021, pp. 10–19. URL: `https://ceur-ws.org/Vol-3163/BICOD21%5C_paper%5C_3.pdf`.

[75] D Robins. “Complex event processing”. In: *Second International Workshop on Education Technology and Computer Science. Wuhan*. Citeseer. 2010, pp. 1–10.

[76] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph databases: new opportunities for connected data*. O’Reilly Media, Inc., 2015.

[77] Brian Sam-Bodden. *Beginning POJOs: Lightweight Java Web Development Using Plain Old Java Objects in Spring, Hibernate, and Tapestry*. Apress, 2006.

[78] Konstantinos Semertzidis and Evaggelia Pitoura. “Time Traveling in Graphs using a Graph Database”. In: *Proceedings of the Workshops of the EDBT/ICDT 2016 Joint Conference, EDBT/ICDT Workshops 2016, Bordeaux, France, March 15, 2016*. Ed. by Themis Palpanas and Kostas Stefanidis. Vol. 1558. CEUR Workshop Proceedings. CEUR-WS.org, 2016. URL: `https://ceur-ws.org/Vol-1558/paper21.pdf`.

[79] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L. Willke, Jeffrey S. Young, Matthew Wolf, and Karsten Schwan. “GraphIn: An Online High Performance Incremental Graph Processing Framework”. In: *Euro-Par*. Vol. 9833. Lecture Notes in Computer Science. Springer, 2016, pp. 319–333.

[80] John Ferguson Smart. *Jenkins - The Definitive Guide: Continuos Integration for the Masses: also Covers Hudson*. O’Reilly, 2011.

[81] Daniel Ståhl, Kristofer Hallén, and Jan Bosch. “Achieving traceability in large scale continuous integration and delivery deployment, usage and validation of the Eiffel framework”. In: *Empir. Softw. Eng.* 22.3 (2017), pp. 967–995.

[82] John Stegeman. *What Is a Knowledge Graph?* `https://neo4j.com/blog/knowledge-graph/what-is-knowledge-graph/`. Accessed: 2025-05-26. 2024.

[83] Ben Stopford. *Designing event-driven systems*. O’Reilly Media, Incorporated, 2018.

[84] Sriskandarajah Suhothayan, Kasun Gajasinghe, Isuru Loku Narangoda, Subash Chaturanga, Srinath Perera, and Vishaka Nanayakkara. “Siddhi: a second look at complex event processing architectures”. In: *SC-GCE*. ACM, 2011, pp. 43–50.

[85] Shixuan Sun, Xibo Sun, Bingsheng He, and Qiong Luo. “RapidFlow: An Efficient Approach to Continuous Subgraph Matching”. In: *Proc. VLDB Endow.* 15.11 (2022), pp. 2415–2427.

[86] Toyotaro Suzumura, Shunsuke Nishii, and Masaru Ganse. “Towards large-scale graph stream processing platform”. In: *23rd International World Wide Web Conference, WWW ’14, Seoul, Republic of Korea, April 7-11, 2014, Companion Volume*. Ed. by Chin-Wan Chung, Andrei Z. Broder, Kyuseok Shim, and Torsten Suel. ACM, 2014, pp. 1321–1326. DOI: `10.1145/2567948.2580051`.

[87] Gábor Szárnyas, Jack Waudby, Benjamin A. Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter A. Boncz. “The LDBC Social Network Benchmark: Business Intelligence Workload”. In: *Proc. VLDB Endow.* 16.4 (2022), pp. 877–890.

[88] Nesime Tatbul. “The Aurora/Borealis/streambase codelines: a tale of three systems”. In: *Making Databases Work*. Vol. 22. ACM Books. ACM / Morgan & Claypool, 2019, pp. 321–332.

[89] Travis CI. *How developers build simple, trustworthy CI/CD pipelines*. Accessed: 2025-06-18. 2025. URL: `https://www.travis-ci.com/`.

[90] Gergely Varró and Frederik Deckwerth. “A Rete Network Construction Algorithm for Incremental Pattern Matching”. In: *ICMT*. Vol. 7909. Lecture Notes in Computer Science. Springer, 2013, pp. 125–140.

[91] Gergely Varró, Andy Schürr, and Dániel Varró. “Benchmarking for Graph Transformation”. In: *VL/HCC*. IEEE Computer Society, 2005, pp. 79–88.

[92] Gergely Varró, Dániel Varró, and Andy Schürr. “Incremental graph pattern matching”. In: *Electronic Communications of the EASST: Graph and Model Transformation* 4 (2006), p. 2006.

[93] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. “A comparison of a graph database and a relational database: a data provenance perspective”. In: *Proceedings of the 48th annual ACM Southeast Conference*. 2010, pp. 1–6.

[94] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. “RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on A Single Machine”. In: *OSDI*. USENIX Association, 2018, pp. 763–782.

[95] Xi Wang, Qianzhen Zhang, Deke Guo, and Xiang Zhao. “A survey of continuous subgraph matching for dynamic graphs”. In: *Knowledge and Information Systems* 65.3 (2023), pp. 945–989.

[96] Eugene Wu, Yanlei Diao, and Shariq Rizvi. “High-performance complex event processing over streams”. In: *SIGMOD Conference*. ACM, 2006, pp. 407–418.

[97] Rongjian Yang, Zhijie Zhang, Weiguo Zheng, and Jeffrey Xu Yu. “Fast Continuous Subgraph Matching over Streaming Graphs via Backtracking Reduction”. In: *Proc. ACM Manag. Data* 1.1 (2023), 15:1–15:26.

[98] Wenlu Yang, Alzennyr Da Silva, and Marie-Luce Picard. “Computing data quality indicators on Big Data streams using a CEP”. In: *IWCIM*. IEEE, 2015, pp. 1–5.

[99] Ping Zhang and Gary Chartrand. *Introduction to graph theory*. Vol. 2. 2.1. Tata McGraw-Hill New York, 2006.

[100] Xu Zhu. “Complex event detection for commodity distribution Internet of Things model incorporating radio frequency identification and Wireless Sensor Network”. In: *Future Gener. Comput. Syst.* 125 (2021), pp. 100–111.