

GOProQ: A Graphical Query Language for Object-Centric Process Analysis

Gyunam Park^{1,2}, Jan Niklas Adams³, and Daniel Schuster⁴

¹ Eindhoven University of Technology
g.park@tue.nl

² Fraunhofer Institute for Applied Information Technology (FIT)

³ Process and Data Science, RWTH Aachen University

⁴ Process Intelligence Solutions

Abstract. Service-oriented architectures facilitate distributed business processes where multiple services collaborate to achieve business goals. However, traditional process mining approaches rely on single-case event logs that fail to capture the complex interactions between services and their associated data objects. Object-Centric Event Logs (OCELs) address this challenge by modeling service processes as graphs of interconnected events and objects, providing a foundation for comprehensive service analysis. While existing querying approaches offer capabilities for traditional event data, they lack the expressiveness needed for analyzing object-centric service interactions. This paper introduces GOProQ, a graphical query language specifically designed for querying object-centric service processes. We identify key requirements for service-oriented process querying, define a formal language that addresses these requirements, and implement an interactive tool for service analysts. Our evaluation across multiple real-world service datasets demonstrates GOProQ’s expressiveness and performance for service monitoring in complex service-oriented environments.

Keywords: Service-Oriented Process Mining · Service Composition Analysis · Object-Centric Event Logs · Service Monitoring · Service Process Querying

1 Introduction

Process mining provides tools and techniques for discovering, analyzing, and enhancing business processes using event data. Traditionally, process mining relied on sequence-based single-case event logs standardized by the XES format [1]. However, this simplistic representation fails to capture the complexity of modern service-oriented processes where multiple objects (such as orders, items, and deliveries) interact simultaneously within service compositions and orchestrations.

To address these limitations, Object-Centric Event Logs (OCELs) have been introduced, representing processes as interconnected graphs of events rather than linear sequences. This approach more naturally reflects the reality of complex service interactions in modern distributed systems.

In service-oriented computing, the ability to query cross-service interactions is essential for monitoring service quality, ensuring compliance with service-level agreements (SLAs), and optimizing service compositions [17,12]. Despite the importance of object-centric service analysis, querying such data remains challenging. Traditional process querying

methods cannot adequately handle the multi-object interactions. For example, analyzing “service executions where orders involve items from multiple suppliers, with payments processed before all deliveries complete” requires understanding relationships between orders, items, suppliers, payments, and deliveries simultaneously, something that single-case approaches cannot express naturally.

Existing process querying methods are predominantly designed for traditional flattened event logs [13,9,11] and general process repositories [15,14,3], with limited applicability to service-oriented OCELs. While some approaches use general-purpose query languages like Cypher [7,8], the lack of specialized query frameworks hampers service analysts to extract meaningful insights for service monitoring and optimization.

To fill this gap, we investigate how querying can be adapted to leverage the multi-object structure of OCELs in service-oriented environments. Specifically, we propose GOProQ, a novel graphical query language tailored for object-centric service process data. Graphical query languages align with the structural properties of the data, reducing query complexity and enhancing user experience in service analysis contexts [18].

This paper makes the following contributions:

- We introduce a formal definition of a graphical query language designed for object-centric service event data, addressing the limitations of existing textual query languages for service monitoring and analysis.
- We implement an interactive application that allows service administrators and analysts to define, execute, and visualize graphical queries on OCELs.
- We evaluate the proposed query language using real-world service-oriented datasets, demonstrating its feasibility for analyzing complex service interactions.

The remainder of this paper is structured as follows. Sect. 2 reviews the state-of-the-art in service process querying and object-centric event data analysis. Next, Sect. 3 introduces preliminaries regarding object-centric event logs and service process executions. Then, Sect. 4 establishes the requirements for an object-centric service process query language, motivating our design choices. Sect. 5 defines the syntax and semantics of the proposed graphical query language based on these requirements. Sect. 6 presents a comprehensive evaluation of the language’s expressiveness and performance for service analysis, and Sect. 7 concludes the paper.

2 Related Work

In this section, we review existing approaches to process querying [16], with a focus on their applicability to object-centric event data in service-oriented environments.

Service-Oriented Process Querying Languages Several frameworks and specialized languages have been developed for querying service process data. The Process Instance Query Language (PIQL) [15] provides a comprehensive approach to querying service process instances, offering capabilities for filtering, pattern matching, and aggregation. However, PIQL’s design focuses on traditional service process instances, making it less suitable for the graph structure of object-centric service executions that capture complex inter-service interactions. The Data-Aware Process Oriented Query Language (DAPOQ) [14] extends querying capabilities by incorporating data perspectives into process analysis. While DAPOQ enables correlation between service events and data elements, its process model assumes case sequences rather than the interconnected service object relationships found in service-oriented OCELs. BP-SPARQL [3] represents a significant advancement in graph-based service process querying. It utilizes SPARQL

to query service process knowledge graphs, enabling complex pattern matching and aggregation across service interactions. While BP-SPARQL shares our approach’s graph representation, it was not specifically designed for the unique characteristics of service-oriented OCELs, where multiple object types interact simultaneously within complex service orchestrations and choreographies.

Graph-based Querying Techniques Graph-based techniques exploit the intuitive visual representation of graphs to model and query service-oriented process data, mirroring the interconnected nature of modern microservice architectures. Ly et al. [13] propose Compliance Rule Graphs (CRGs) based on first-order predicate logic to instantiate and verify service compliance rules. Knuplesch et al. [11] extend CRGs into Extended Compliance Rule Graphs (eCRGs), incorporating additional service process perspectives such as time, data, and resources for comprehensive service monitoring. Hildebrandt et al. [9] introduce Dynamic Condition Response Graphs (DCR Graphs) as a means to capture and enforce service compliance constraints visually. Despite their effectiveness in modeling service compliance, these approaches are tailored to traditional service event logs and fail to account for the complex relationships among different service objects inherent in object-centric service event data, particularly in microservice-based architectures.

Object-Centric Service Process Querying Approaches The unique challenges of querying object-centric service event data have led to the development of specialized frameworks for service-oriented systems. Esser and Fahland [7,8] introduce event knowledge graphs and implement a querying framework using Neo4j and Cypher to handle object-centric event logs in service environments. While this approach offers a flexible means of querying using the general-purpose query language, i.e., Cypher, it is not specifically designed for the analysis of object-centric service event data in complex service orchestrations. Such Cypher queries perform well for simple queries but struggle with more complex inter-service queries due to the lack of specialization for object-centric service process analysis and monitoring of service-level agreements.

Model Checking and Temporal Logic Approaches Model checking approaches have been extensively applied to process verification, with tools like SPIN [10] and NuSMV [4], providing frameworks for verifying temporal properties over finite state machines. These approaches excel at analyzing single execution threads or global system states, where state transitions represent discrete system changes with clear branching choices, as demonstrated in classical applications to communication protocols and concurrent systems [6]. However, the semantic foundations of model checking are fundamentally misaligned with object-centric process querying requirements. While model checking tools like SPIN model system behaviors through state transitions with explicit choices and branching decisions, object-centric process executions represent recorded interactions between multiple service objects with independent lifecycles, where edges capture temporal precedence based on shared objects rather than state transitions. The multi-object nature of service-oriented architectures creates additional complexity that traditional model checking approaches struggle to address efficiently, as evidenced by the state explosion problem when modeling multiple interacting entities [5]. These semantic differences necessitate specialized approaches for object-centric process querying that can natively handle multi-object interactions and actual execution analysis.

3 Preliminaries

In this section, we introduce the core concepts underlying our query language, specifically focusing on Object-Centric Event Logs (OCELs) and the notion of service process executions. We introduce service process executions as the central abstraction for querying object-centric service event data.

Object-Centric Event Logs Traditional event logs represent service activities from a single-case perspective, typically capturing events related to a single service process instance (or case). In contrast, OCELs record events involving multiple interacting service objects of various types, which better reflect the reality of service orchestrations and choreographies in modern architectures.

Formally, an OCEL is a tuple: $L = (E, O, \pi_{obj}, \pi_{act}, \pi_{time})$, where E is the set of service events; O is the set of service objects, with each object $o \in O$ assigned a type $\pi_{type}(o)$; $\pi_{obj}: E \rightarrow \mathcal{P}(O)$ maps each service event to a set of associated service objects; $\pi_{act}: E \rightarrow \mathcal{A}$ assigns an activity label to each service event, where $\mathcal{A}$ is the universe of service activities; $\pi_{time}: E \rightarrow \mathcal{T}$ maps each service event to a timestamp, enabling service-level agreement monitoring and temporal analysis.

Tab. 1 shows an event log $L_1 = (E_1, O_1, \pi_{obj}, \pi_{act}, \pi_{time})$ from a service-oriented order fulfillment system where $E_1 = \{e_1, e_2, e_3, \dots\}$ and $O_1 = \{o_1, o_2, i_1, i_2, d_1, c_1, \dots\}$. Each service event $e_i \in E_1$ is associated with a set of service objects through the mapping π_{obj} . For example, $\pi_{obj}(e_1) = \{o_1, i_1, i_2, i_3, c_1\}$ indicates that the service event e_1 involves the order service object o_1 , item service objects i_1, i_2, i_3 , and customer service object c_1 , representing a cross-service interaction typical in microservice architectures. π_{act} assigns service activities to events, e.g., $\pi_{act}(e_1) = \text{Place Order}$, while π_{time} assigns timestamps to events, e.g., $\pi_{time}(e_1) = 21.10.02:26:11$, which is crucial for monitoring service-level agreements and service performance.

Table 1: Example for an object-centric service event log

Event ID	Service Activity	Timestamp	Service Object Types			
			Order	Item	Delivery	Cust.
e_1	Place Order	21.10.02:26:11	o_1	i_1, i_2, i_3		c_1
e_2	Confirm Order	21.10.02:26:12	o_1	i_1, i_2, i_3		c_1
e_3	Pick Item	21.10.02:30:55		i_1		
e_4	Pick Item	21.10.02:37:50		i_2		
e_5	Pick Item	21.10.02:57:50		i_3		
e_6	Load Cargo	21.10.03:02:26		i_1, i_2	d_1	
e_7	Fuel Car	23.10.14:30:54			d_1	
e_8	Send Invoice	23.10.16:25:46	o_1			c_1
e_9	Load Cargo	23.10.17:02:26		i_3	d_2	
e_{10}	Receive Payment	24.10.12:22:17	o_1			c_1
e_{11}	Receive Payment	24.10.14:55:44	o_1			c_1
e_{12}	Start Route	24.10.19:56:03		i_1, i_2	d_1	
e_{13}	Start Route	24.10.21:54:03		i_1, i_2	d_1	
e_{14}	End Route	26.10.09:56:03		i_3	d_2	
e_{15}	End Route	26.10.15:34:03		i_3	d_2	
e_{16}	Place Order	21.10.11:36:57	o_3	i_4		c_2
...	...	...	...	...	...	...

Given event log $L = (E, O, \pi_{obj}, \pi_{act}, \pi_{time})$, the precedence constraints between service events $\prec_L$ capture the temporal dependencies between the events based on shared service objects, representing service dependencies and interactions. Formally, $\prec_L = \{(e, e') \in E \times E \mid e \neq e', \exists o \in O: o \in \pi_{obj}(e) \wedge o \in \pi_{obj}(e') \wedge \pi_{time}(e) < \pi_{time}(e') \wedge \nexists e'' \in E: o \in \pi_{obj}(e'') \wedge \pi_{time}(e) < \pi_{time}(e'') < \pi_{time}(e')\}$. For instance, since both e_1 (**Place Order**) and e_2 (**Confirm Order**) involve the service object o_1 , and $\pi_{time}(e_1) < \pi_{time}(e_2)$ without any event in between, we have $(e_1, e_2) \in \prec_{L_1}$, representing a direct service dependency.

Service Process Executions The primary goal of our query language is to query service process executions rather than individual service events. A service process

execution is a directed graph of events representing the execution of a service-oriented process. The edges describe the precedence constraints between service events given by service objects, capturing service interactions. This model generalizes traditional case-based service execution, which is a sequence of events. A sequence is a special case of a graph, just as a monolithic service is a special case of a microservice architecture.

Formally, a service process execution for a subset of service objects $O' \subseteq O$ is $p = (E', K)$, where (i) $E' \subseteq E$ is the set of service events associated with the service objects in O' , i.e., $\forall e \in E' [\pi_{obj}(e) \cap O' \neq \emptyset]$ and (ii) $K \subseteq (E' \times E') \cap \prec_L$ is the set of precedence constraints, capturing the temporal order of service events based on shared service objects, representing service dependencies.

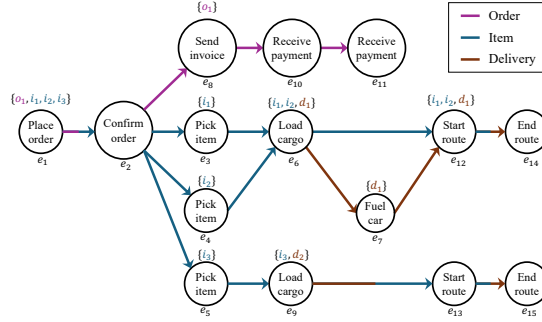


Fig. 1: Service process execution p_1 extracted from L_1 in Tab. 1, representing cross-service interactions. Note that some service object types/objects are omitted for brevity. c_1 is involved in all the service events.

Fig. 1 shows a service process execution $p_1 = (E', K)$ from the service event log L_1 in Tab. 1. $E' = \{e_1, \dots, e_{15}\}$ is the set of service events included in the process execution. Each service event in E' is associated with a subset of service objects O' , such as orders (o_1), items (i_1, i_2, i_3), deliveries (d_1, d_2), and customers (c_1), representing interactions between different microservices in the system. The edges K define the precedence constraints between service events based on shared service objects and their temporal order, capturing the choreography of the service execution.

Service process executions are derived from object-centric event logs. One possible approach treats the service event log as a bipartite graph where service events and service objects form two distinct node sets, with edges connecting events to their associated objects. Service process executions are then identified by finding connected components in this bipartite graph—events that share service objects are grouped into the same execution, representing cohesive service interactions within a distributed architecture. For instance, in Tab. 1, service process execution p_1 (shown in Fig. 1) includes events e_1 through e_{15} because they all share service objects and form a connected component. Conversely, service event e_{16} belongs to a separate process execution as it is associated with a different order service object (o_3) and customer (c_2) that do not interact with the objects in p_1 , thus representing an independent service process instance.

4 Requirements for Object-Centric Process Querying

In this section, we establish requirements for a query language specifically designed for object-centric service process data. We analyze the structural characteristics of object-centric service process executions to identify fundamental querying dimensions, then

define necessary expressiveness and performance requirements to guide our language design for service-oriented systems.

Service process executions extracted from OCEs have unique structural characteristics that impact querying requirements. As introduced in Sect. 3, a service process execution is represented as a graph, where nodes correspond to service events and edges represent precedence constraints between service events, mirroring the interconnected nature of modern service orchestrations.

The basic principle of querying service-oriented graphs involves interrogating nodes and edges (i.e., structural elements) using their associated labels [2]. In object-centric service process executions, these labels carry rich semantic information:

- Node labels include: (1) service activity names (e.g., `Load cargo`) and (2) service object types (e.g., `Item` and `Delivery`).
- Edge labels capture temporal service relationships: (1) between service activities (e.g., `Pick item` and `Load cargo`) and (2) between service object types (e.g., `Item` and `Delivery`), representing service dependencies.

These structural characteristics give rise to a two-dimensional framework for service process querying, as illustrated in Tab. 2. The horizontal dimension represents structural elements (nodes and edges), while the vertical dimension represents label types (service activities and service object types). Each cell in this matrix represents a specific query capability that our language must support for effective service monitoring and analysis.

Table 2: Dimensions of querying service process executions. The query language must support all query types represented in this matrix to comprehensively address object-centric service process analysis needs.

Structural Elements	Label Types	
	Service Activity	Service Object Type
Node (Service Event)	Query nodes based on service activity presence, including start/end service activities.	Query nodes based on their associated service object types.
Edge (Service Precedence Constraint)	Query edges based on temporal relationships between service activities.	Query edges based on temporal relationships between associated service object types.

Required Expressiveness for Service-Oriented Querying Based on the querying dimensions in Tab. 2 and real-world service analysis needs, we define four key expressiveness requirements:

- **Support for all query dimensions:** The language must enable queries across both structural elements (nodes and edges) and label types (service activities and service object types), covering all cells in the matrix shown in Tab. 2.
- **Service-level cardinality constraints:** Beyond simple presence queries, the language must support cardinality constraints across all dimensions. This includes: Counting occurrences of service activities (e.g., “service executions with at least three payment activities”); Quantifying service object types (e.g., “orders with more than ten items”); Specifying frequency of temporal service relationships (e.g., “service shipment followed by delivery at least twice for SLA monitoring”).
- **Temporal relationship expressiveness:** The language must express both direct temporal succession (directly-follows) and indirect succession (eventually-follows) service relationships. This distinction is crucial for analyzing complex service flows where intermediate service activities may occur between key service steps, particularly in microservice orchestrations.
- **Query composition:** Service-level requirements typically involve combinations of simpler service patterns. The language must support logical operators (AND, OR, NOT) to compose queries across all dimensions into meaningful service patterns for comprehensive service monitoring and optimization.

Performance Considerations for Service Analysis Object-centric service event logs are typically larger and more complex than traditional service logs, imposing significant performance challenges for service monitoring and analysis. We identify three critical performance requirements:

- **Service data scalability:** The language implementation must scale to handle real-world OCELs with thousands of service process executions, each containing numerous service events and service objects across microservice architectures.
- **Query optimization:** Evaluation strategies must prioritize less computationally expensive operations for efficient service monitoring. For example, filtering by service object type cardinality before checking temporal service patterns can significantly reduce evaluation time, enabling near real-time service analysis.
- **Interactive service response time:** For practical service monitoring, query evaluation should provide results within reasonable timeframes for real-time monitoring needs, depending on service dataset size and query complexity.

5 GOProQ: Graphical Query Language for Object-Centric Service Process Querying

To address the requirements established in Sect. 4, we propose a graphical query language, GOProQ, specifically designed for service-oriented environments. In the following, we introduce the formal syntax and semantics of GoProQ and illustrate its application through concrete service monitoring examples.

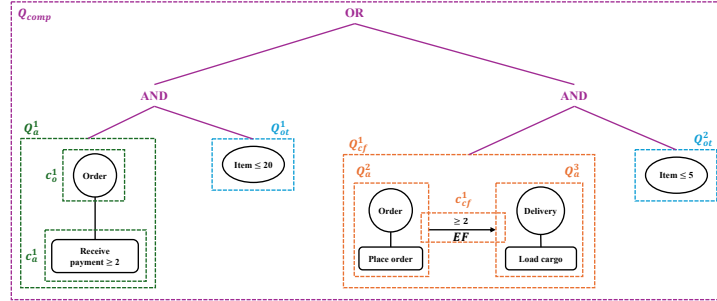


Fig. 2: An example graph query demonstrates the composition of multiple query types to find service process executions with either: (1) an order with fewer than 20 items that have been paid for twice, monitoring potential duplicate payment services, or (2) an order that was loaded into two delivery services when the number of items is fewer than 5, indicating possible inefficient service resource allocation. Querying orders with multiple payment services can reveal duplicate efforts in billing services, while identifying orders loaded into multiple cargo services with few items can highlight optimization opportunities in logistics service planning and service-level agreement compliance.

5.1 GOProQ: Syntax for Service Process Querying

GOProQ is structured around three fundamental query types essential for service analysis: *activity query*, *object type query*, and *control-flow query*.

Activity Query An activity query targets specific service activities within service process executions, checking for their presence, start, or end. Formally, an activity query is defined as:

$$Q_a = (c_o, c_a), \text{ where}$$

- c_o is the service object type component:

$$c_o = (ot) \quad (\text{single service object type}), \text{ or}$$

$$c_o = (ot, \odot, n) \quad (\text{service object cardinality})$$
 where $ot \in \mathcal{OT} \cup \{\text{ANY}\}$ is a service object type, and $n \in \mathbb{N}_0$ is a count of service objects.
- c_a is the service activity component, which can be:

$$c_a = (l_1) \quad (\text{single service activity}),$$

$$c_a = (\text{S}, \{l_1, \dots, l_n\}) \quad (\text{start service activity}),$$

$$c_a = (\text{E}, \{l_1, \dots, l_n\}) \quad (\text{end service activity}),$$

$$c_a = (\Delta, \{l_1, \dots, l_n\}) \quad (\text{quantified service activities}), \text{ or}$$

$$c_a = (l_1, \odot, n) \quad (\text{service activity cardinality})$$
 where $l_1, \dots, l_n \in \mathcal{A}$ are service activity labels, $\Delta \in \{\text{ANY}, \text{ALL}\}$ is a service quantifier, $\odot \in \{\geq, \leq, =\}$ is a comparison operator, and $n \in \mathbb{N}_0$ specifies the service activity count for SLA verification.

In Fig. 2, an activity query $Q_a^1 = (c_o^1, c_a^1)$ with $c_o^1 = (\text{Order})$ and $c_a^1 = (\text{Receive payment}, \geq, 2)$. This checks for the presence of at least 2 occurrences of the payment service activity **Receive payment** for at least one object of type *Order*, helping to identify potential duplicate payment service executions.

Object Type Query Object type query checks for the existence of service objects of a specified type within the service process execution, enabling monitoring of service resource allocation:

$$Q_{ot} = (c_{ot}), \text{ where}$$

$$c_{ot} = (ot) \quad (\text{simple service object type}), \text{ or}$$

$$c_{ot} = (ot, \odot, n) \quad (\text{service object type with cardinality})$$

with $ot \in \mathcal{OT}$ representing service object types, and $n \in \mathbb{N}_0$ representing the service object count for capacity planning.

In Fig. 2, a service object query $Q_{ot}^1 = (\text{Item}, \leq, 20)$. This checks whether there are not more than 20 service objects of type *Item* in the service process execution, providing insights for service capacity planning and resource allocation.

Control-Flow Query A control-flow query Q_{cf} captures temporal and cardinality relationships between service activities in a service process execution, crucial for service orchestration analysis. It is formally defined as:

$$Q_{cf} = (Q_1, Q_2, T, c_{cf}), \text{ where}$$

- Q_1 and Q_2 are activity queries that define the service activities involved in the service control-flow relationship.
- $T \in \{\text{DF}, \text{EF}\}$ specifies the type of temporal service relationship:
 - **DF** (Directly-Follows): The service activity specified in Q_1 must be directly followed by the service activity specified in Q_2 , representing immediate service dependencies.
 - **EF** (Eventually-Follows): The service activity specified in Q_1 must eventually be followed by the service activity specified in Q_2 , representing service choreographies where intermediate services may execute.
- c_{cf} is a service constraint component, which can be:

- $(\odot_1, n_1)$: A service object cardinality constraint, where $\odot_1 \in \{\leq, \geq, =\}$ is a comparison operator, and $n_1 \in \mathbb{N}_0$ denotes the number of service objects in the process execution that satisfy the control-flow relationship.
- $(\odot_2, n_2)$: A service relationship cardinality constraint, where $\odot_2 \in \{\leq, \geq, =\}$ is a comparison operator, and $n_2 \in \mathbb{N}_0$ specifies the frequency of the temporal relationship for each service object, useful for SLA monitoring.
- $(\odot_1, n_1, \odot_2, n_2)$: A combined service constraint that specifies both the service object cardinality $(\odot_1, n_1)$ and the service relationship cardinality $(\odot_2, n_2)$.

In Fig. 2, $Q_{cf}^1 = (Q_a^2, Q_a^3, \text{EF}, c_{cf}^1)$ with activity query $Q_a^2 = ((\text{Order}), (\text{Place order}))$, activity query $((\text{Delivery}), (\text{Load cargo}))$, $c_{cf}^1 = (\geq_1, 2)$. This checks if, in the service process execution, the service activity **Place order** eventually follows the service activity **Load cargo** for at least two order-delivery service pairs, helping to identify unusual service orchestration patterns.

Query Composition To express more complex service requirements, queries can be composed using logical operators such as conjunction (**AND**), disjunction (**OR**), and negation (**NOT**) to create comprehensive service monitoring patterns. For instance, Fig. 2 illustrates a composed query Q_{comp} .

The composed query illustrated in Fig. 2 demonstrates the advantages of our graphical approach over existing query languages for object-centric service analysis. This query involves multiple object types with cardinality constraints, activity frequency requirements, and temporal relationships—patterns that are challenging to express in existing approaches. In BPSPARQL, this query would require complex nested subqueries with multiple **GROUP BY** clauses for cardinality counting, explicit temporal logic for the eventually-follows relationship, and intricate **FILTER** expressions to handle multi-object constraints. Using Cypher, the query would need multiple **MATCH** clauses with separate aggregation steps for each object type, manual temporal relationship handling, and complex **WHERE** conditions. In contrast, our graphical approach captures the same semantics through intuitive visual composition of four query components (two activity queries, one object type query, and one control-flow query) connected by logical operators.

5.2 GProQ: Formal Semantics for Service Analysis

The semantics of the query language are defined using the evaluation function $\Phi : \mathbb{U}_Q \times \mathbb{U}_P \rightarrow \{\text{true}, \text{false}\}$, where $\mathbb{U}_Q$ is the universe of queries and $\mathbb{U}_P$ is the universe of service process executions.

Activity Query Semantics The evaluation function for an activity query $Q_a = (c_o, c_a)$ is defined as:

$$\Psi(Q_a, p) \iff \Psi_1(c_o, O_p^{ot}) \wedge \Psi_2(c_a, O_p^{ot}), \text{ where}$$

- O_p^{ot} is the set of service objects of type ot in the service process execution p , i.e., $O_p^{ot} = \{o \in O_p \mid \pi_{type}(o) = ot\}$, where O_p is the set of service objects in p .
- $\Psi_1(c_o, O_p^{ot})$ ensures that the specified number of service objects of type ot exist in the service process execution. Formally, given $c_o = (ot, \odot, n)$,

$$\Psi_1(c_o, O_p^{ot}) \iff |O_p^{ot}| \odot n$$

If $c_o = (ot)$, we assume $\odot = \geq$ and $n = 1$.

- $\Psi_2(c_a, O_p^{ot})$ verifies whether the specified service activity or set of service activities satisfies the constraints in the trace of each service object $o \in O_p^{ot}$. Formally,

$$\Psi_2(c_a, O_p^{ot}) \iff \forall o \in O_p^{ot} [\delta(o, c_a)], \text{ where}$$

$\delta(o, c_a)$ evaluates the service activity condition for the trace of service object o as:

$$\delta(o, c_a) \iff \begin{cases} \exists e \in E(o) [\pi_{act}(e) = l_1], & \text{if } c_a = (l_1) \\ \pi_{trace}(o)(1) \in \{l_1, \dots, l_n\}, & \text{if } c_a = (\mathbf{S}, \{l_1, \dots, l_n\}) \\ \pi_{trace}(o)(|\pi_{trace}(o)|) \in \{l_1, \dots, l_n\}, & \text{if } c_a = (\mathbf{E}, \{l_1, \dots, l_n\}) \\ \Delta = \mathbf{ANY} \Rightarrow \exists l_i \in \{l_1, \dots, l_n\}, \exists e \in E(o) [\pi_{act}(e) = l_i], & \text{if } c_a = (\Delta, \{l_1, \dots, l_n\}) \\ \Delta = \mathbf{ALL} \Rightarrow \forall l_i \in \{l_1, \dots, l_n\}, \exists e \in E(o) [\pi_{act}(e) = l_i], & \text{if } c_a = (\Delta, \{l_1, \dots, l_n\}) \\ |\{e \in E(o) | \pi_{act}(e) = l_1\}| \odot n, & \text{if } c_a = (l_1, \odot, n) \end{cases}$$

where $E(o)$ is the set of service events associated with service object o and $\pi_{trace}(o)$ represents the sequence of service activities associated with a service object o , e.g.,

$$\pi_{trace}(o_1) = \langle \mathbf{Place\ Order}, \dots, \mathbf{Receive\ Payment} \rangle.$$

Consider the activity query $Q_a^1 = (c_o^1, c_a^1)$ in Fig. 2, where $c_o^1 = (\mathbf{Order})$ and $c_a^1 = (\mathbf{Receive\ payment}, \geq, 2)$. This query checks whether there are at least two occurrences of the payment service activity **Receive payment** for service objects of type **Order**, potentially indicating duplicate payment service executions. For the service process execution p_1 in Fig. 1: (i) $O_{p_1}^{\mathbf{Order}} = \{o_1\}$, (ii) $\Psi_1(c_o^1, p_1): |\{o_1\}| \geq 1$ is true, and (iii) $\Psi_2(c_a^1, p_1)$: For service object o_1 , the payment service activity **Receive payment** occurs in service events e_{10} and e_{11} ; Thus, $|\{e_{10}, e_{11}\}| \geq 2$ holds. Hence, Q_a^1 evaluates to *true* for p_1 , indicating potential duplicate payment processing.

Object Type Query Semantics The evaluation function for an object type query $Q_{ot} = (ot, \odot, n)$ checks whether the service process execution p satisfies the service object type and cardinality constraints specified in Q_{ot} , enabling service capacity monitoring. Formally,

$$\Psi(Q_{ot}, p) \iff |O_p^{ot}| \odot n, \text{ where}$$

O_p^{ot} is the set of service objects of type ot in the service process execution p .

Consider the object type query $Q_{ot}^1 = (\mathbf{Item}, \leq, 20)$ in Fig. 2. This query checks whether there are fewer than 20 service objects of type **Item**, important for service capacity planning. For the service process execution p_1 in Fig. 1: (i) $O_p^{\mathbf{Item}} = \{i_1, i_2, i_3\}$ and (ii) $|\{i_1, i_2, i_3\}| \leq 20$. Hence, Q_{ot}^1 evaluates to *true* for p_1 , indicating compliance with capacity constraints.

Control-Flow Query Semantics The evaluation function for a control-flow query $Q_{cf} = (Q_1, Q_2, T, c_{cf})$ evaluates if a service process execution satisfies (i) the activity queries Q_1 and Q_2 and (ii) the temporal service constraint T with the service cardinality constraints c_{cf} , essential for service orchestration monitoring. Formally,

$$\Psi(Q_{cf}, p) \iff \Psi(Q_1, p) \wedge \Psi(Q_2, p) \wedge \Omega(\Lambda(T, p), c_{cf}), \text{ where}$$

- the function $\Lambda(T, p)$ produces a mapping of service object pairs (o_i, o_j) to the corresponding service event pairs (e_i, e_j) that satisfy the temporal service relationship T :

$$\Lambda(T, p) = \{((o_i, o_j), \{(e_i, e_j) | e_i \in E(o_i, l_1), e_j \in E(o_j, l_2),$$

$$\pi_T(e_i, e_j)\}) | o_i \in O_{ot_1}, o_j \in O_{ot_2}\}, \text{ where}$$

- l_1 and l_2 are the service activity labels, and ot_1 and ot_2 are the service object types, specified in Q_1 and Q_2 ,

- O_p^{ot} represents the set of service objects of type ot in the service process execution p ,
- $E(o,a)$ is the set of service events associated with service object o and labeled with service activity a , and
- the temporal service condition $\pi_T(e_i, e_j)$ is defined as follows:

$$\pi_T(e_i, e_j) \iff \begin{cases} e_i \prec_L e_j, & \text{if } T = \text{DF}, \\ \exists e_1, \dots, e_n \in E, n \geq 2, \\ \quad [e_1 = e_i, e_n = e_j, \forall k \in [1, n-1]: (e_k, e_{k+1}) \in \prec_L], & \text{if } T = \text{EF}. \end{cases}$$

- the function Ω evaluates the extracted service event pairs against the specified service constraints c_{cf} :

$$\Omega(\Lambda(T, p), c_{cf}) \iff \begin{cases} |\text{dom}(\Lambda(T, p))| \odot_1 n_1, & \text{if } c_{cf} = (\odot_1, n_1), \\ \min(|\{ \Lambda(T, p)((o_i, o_j)) | \\ \quad (o_i, o_j) \in \text{dom}(\Lambda(T, p)) \}|) \odot_2 n_2, & \text{if } c_{cf} = (\odot_2, n_2), \\ \Omega(\Lambda(T, p), (\odot_1, n_1)) \wedge \Omega(\Lambda(T, p), (\odot_2, n_2)), & \text{if } c_{cf} = (\odot_1, n_1, \odot_2, n_2). \end{cases}$$

Consider the control-flow query $Q_{cf}^1 = (Q_a^2, Q_a^3, \text{DF}, c_{cf}^1)$ in Fig. 2 where $Q_a^2 = ((\text{Order}), (\text{Place order}))$, $Q_a^3 = ((\text{Delivery}), (\text{Load cargo}))$, and $c_{cf}^1 = (\geq_1, 2)$. This query checks whether the service activity **Place order** directly follows the service activity **Load cargo** for at least two order-delivery service pairs, revealing potential service orchestration anomalies. For the service execution p_1 in Fig. 1: (i) $\Psi(Q_a^2, p_1)$: *true*, as service object o_1 performs the service activity **Place order**, (ii) $\Psi(Q_a^3, p_1)$: *true*, as service objects d_1 and d_2 perform the service activity **Load cargo**, and (iii) $\Lambda(\text{DF}, p_1)$: Service event pair (e_1, e_6) satisfies the **Directly-Follows** service condition for o_1 and d_1 , and similarly for d_2 . Since the service cardinality constraint $c_{cf}^1 = (\geq_1, 2)$ is met, Q_{cf}^1 evaluates to *true* for p_1 , indicating potential service orchestration patterns requiring optimization.

Query Composition Semantics Formally, for two queries $Q_1, Q_2 \in \mathbb{U}_Q$, the composed query semantics are defined as:

$$\Psi(Q_{comp}, p) \iff \begin{cases} \Psi(Q_1, p) \wedge \Psi(Q_2, p), & \text{if } Q_{comp} = (Q_1 \wedge Q_2) \\ \Psi(Q_1, p) \vee \Psi(Q_2, p), & \text{if } Q_{comp} = (Q_1 \vee Q_2) \\ \neg \Psi(Q_1, p), & \text{if } Q_{comp} = \neg Q_1 \end{cases}$$

Consider the composed query Q_{comp} in Fig. 2, which combines multiple service patterns to identify either duplicate payment services or inefficient logistics service allocation:

$$Q_{comp} = (Q_a^1 \wedge Q_{ot}^1) \vee (Q_{cf}^1 \wedge Q_{ot}^2)$$

This is evaluated as follows:

$$\begin{aligned} \Psi(Q_{comp}, p_1) &\iff (\Psi(Q_a^1, p_1) \wedge \Psi(Q_{ot}^1, p_1)) \vee (\Psi(Q_{cf}^1, p_1) \wedge \Psi(Q_{ot}^2, p_1)) \\ \Psi(Q_{comp}, p_1) &\iff (true \wedge true) \vee (true \wedge true) \\ \Psi(Q_{comp}, p_1) &\iff true \end{aligned}$$

This evaluation confirms that the service process execution exhibits both potential duplicate payment services and inefficient logistics service resource allocation, highlighting opportunities for service optimization.

6 Evaluation

To evaluate the proposed query language for service-oriented environments, we implemented a graphical user interface tool for querying service-based OCELs. Users can import service OCEL files, compose queries, and visualize the resulting service process executions. The source code and user manual of the tool are publicly available at <https://github.com/gyunamister/GOProQ>.

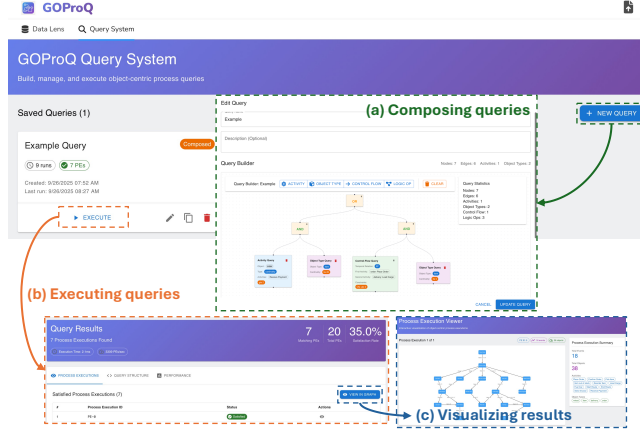


Fig. 3: A screenshot of the tool for object-centric process querying

This section presents a comprehensive evaluation of our query language, focusing on how it addresses the service-oriented requirements established in Sect. 4. We evaluate both the expressiveness of the language for service analysis through our implementation in the GOProQ tool and assess its performance feasibility for service monitoring using a series of experimental evaluations across diverse service datasets.

Table 3: Overview of service-oriented datasets

Service OCEL	#Service Events	#Service Executions ^a	#Service Objects	Service Object Types
Loan Application (Large)	392,018	31,305	74,139	application, offer
Loan Application (Small)	64,770	7,198	14,396	application, offer
Recruiting ^d	6,980	1,289	15,05	applications, offers
Travel Reimbursement ^e	18,246	375	1,761	Task, permit id
Procure-To-Payment ^d	24,854	16,822	74,489	MATNR, MBLNR
Order-To-Cash ^d	239,391	48	796	item, order, delivery

6.1 Evaluation of Expressiveness Requirements

The GOProQ tool implementation demonstrates that our query language satisfies the expressiveness requirements for service analysis established in Sect. 4. Specifically:

- **Support for all query dimensions:** The tool successfully implements query constructs that cover all cells in the query dimensions matrix (Tab. 2). It supports activity queries, object type queries, and temporal service relationship queries between both service activities and service object types, providing comprehensive coverage of the defined service dimensions.
- **Service-level cardinality constraints:** The implementation supports service cardinality constraints across all dimensions, allowing users to specify service monitoring conditions such as “service processes with at least three payment service activities” or “orders with more than ten items,” enabling effective SLA monitoring.

- **Temporal relationship expressiveness:** Both Directly-Follows (DF) and Eventually-Follows (EF) service relationships are implemented in the tool, enabling service analysts to distinguish between immediate service succession and indirect temporal service relationships in complex microservice orchestrations.
- **Query composition:** The graphical interface of GProQ allows users to visually compose queries using logical operators (AND, OR, NOT), enabling the construction of complex service patterns that combine multiple simpler service conditions for comprehensive service monitoring and analysis.

6.2 Evaluation of Performance Requirements

Experimental Setup We conducted experiments using the six service-oriented OCEL datasets. Each dataset represents different service environments, from financial services (Loan Application) to logistics services (Order-To-Cash). The hardware specifications are as follows: Intel Core i5-12600K processor (10 cores, 16 threads, base frequency 3.7 GHz, turbo frequency 4.9 GHz), 32 GB DDR4 RAM (3600 MHz), and Samsung SSD 980 Pro (1 TB, sequential write speed 7,000 MB/s, read speed 5,000 MB/s).

Computation Time of Individual Queries To evaluate the performance requirements for service monitoring established in Sect. 4, we first assess the computation times of individual query types. Tab. 4 presents the results for object type, activity, and control-flow queries. Object type queries demonstrate the shortest computation times, as they operate on individual service objects without considering temporal service relationships. Activity queries, such as *Presence*, *Start*, and *End*, exhibit slightly higher computation times but remain computationally efficient, with variations averaging only 1.3%. Control-flow queries, particularly *Eventually-Follows* (EF), have the longest computation times due to the need to explore paths between service events, which is critical for complex service orchestration analysis. In the Travel Reimbursement and Procure-To-Payment datasets, EF queries could not be computed within a reasonable time frame for real-time service monitoring. This limitation is attributed to the presence of service process executions with thousands of nodes and edges (e.g., 8,481 service nodes and 8,611 service dependencies in Travel Reimbursement).

Table 4: Computation time for different query types across datasets

Service Dataset	Service Object Type	Presence	Start/End	DF	EF
Loan Application (Large)	0.2261s	12.4943s	12.6322s	16.6797s	20.7484s
Loan Application (Small)	0.0102s	0.5119s	0.5023s	0.7624s	1.2207s
Recruiting	0.0077s	0.0159s	0.0155s	0.0384s	0.0485s
Travel Reimbursement	0.0023s	0.1575s	0.1600s	0.9523s	-
Procure-To-Payment	0.0151s	0.2380s	0.2304s	1.5349s	-
Order-To-Cash	0.0009s	0.0884s	0.0918s	0.0892s	0.4315s

These results directly address the service data scalability and service response time requirements identified in Sect. 4. While most queries execute efficiently across the test datasets, we identify clear performance boundaries, particularly for complex temporal queries on large service process executions, which has implications for real-time service monitoring capabilities.

Reordering Strategy To address the query optimization requirement from Sect. 4, we developed a reordering strategy based on the insights from individual query evaluations. The strategy prioritizes the evaluation of less computationally intensive queries to filter non-matching service process executions early, thereby reducing the overall computation time for effective service monitoring. The reordering strategy comprises the following steps:

- Evaluate individual queries first:
 - Execute object type queries first, as they are the least computationally expensive for quick service filtering.
 - Execute activity queries next, prioritizing those with fewer associated service objects (e.g., *Item* before *Order*). Queries involving **ANY** are executed last as they are less restrictive in service filtering.
 - Finally, execute control-flow queries, with **DF** queries preceding **EF** queries due to their stricter temporal constraints and lower computational cost.
- Handle composed queries with logical operators:
 - Execute **NOT** queries first to exclude non-matching service executions early.
 - Execute **AND** queries next, as all sub-queries must be satisfied for compliance.
 - Execute **OR** queries last, as they require evaluating all possible sub-queries.

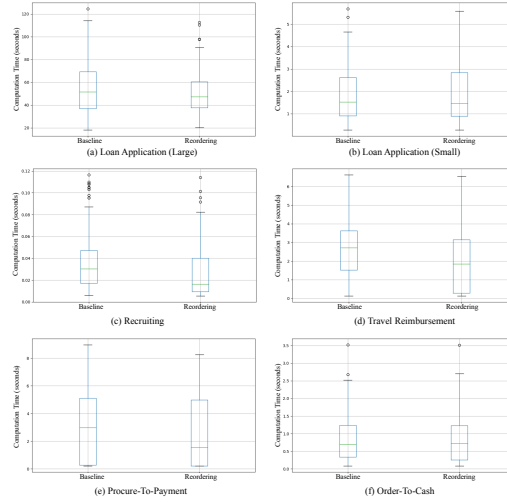


Fig. 4: Query computation times of the baseline and reordering approaches, showing improved efficiency for service monitoring through optimization

Fig. 4 compares the reordering strategy with a baseline approach that evaluates queries in random order. The reordering strategy consistently reduces computation times, highlighting its effectiveness for responsive service monitoring. For instance, the median computation time for the recruiting service dataset is reduced by 57.1%, enabling more responsive service analysis. Using the reordering strategy, the queries on all datasets except Loan Application (Large) take less than 3 seconds. This demonstrates the practical feasibility of the query language for most real-life service monitoring scenarios, directly addressing the interactive service response time requirement from Sect. 4.

7 Conclusion

In this paper, we introduced GProQ. Our approach enables expressive queries on activities, object types, and their relationships within service-oriented architectures. We implemented the query language as an interactive tool and evaluated its feasibility and performance for service analysis using six publicly available service OCEL datasets. The results demonstrate the practical usability of the language for service monitoring and orchestration analysis, with computation times that are manageable for most service scenarios.

The approach has limitations in the context of service-oriented computing. First, direct service-to-service relationships not mediated through activities were not explored. While GOProQ supports querying shared service events, user-defined service relationships such as directly-follows or eventually-follows between service objects remain unexplored, which could be valuable for complex microservice dependency analysis. Second, the computation time for complex queries remains a challenge, particularly for large-scale service environments with extensive process executions, where query response times may exceed practical limits for real-time service monitoring. Third, while our graphical approach reduces cognitive load for typical service monitoring queries, very complex multi-layer queries can become cognitively demanding.

References

1. van der Aalst, W.M.P.: *Process Mining - Data Science in Action*. Springer (2016)
2. Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J.L., Vrgoc, D.: Foundations of modern query languages for graph databases. *ACM Comput. Surv.* **50**(5), 68:1–68:40 (2017)
3. Beheshti, A., Benatallah, B., Motahari-Nezhad, H.R., Ghodratnama, S., Amouzgar, F.: BP-SPARQL: A query language for summarizing and analyzing big process data. In: Polyvyanyy, A. (ed.) *Process Querying Methods*, pp. 21–48. Springer (2022)
4. Cimatti, A., et al.: Nusmv 2: An opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) *CAV*. vol. 2404, pp. 359–364 (2002)
5. Clarke, E.M.: Model checking - my 27-year quest to overcome the state explosion problem. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) *LPAR*. vol. 5330, p. 182 (2008)
6. Clarke, E.M., Grumberg, O., Kroening, D., Peled, D.A., Veith, H.: *Model checking* (2018)
7. Esser, S., Fahland, D.: Multi-dimensional event data in graph databases. *J. Data Semant.* **10**(1-2), 109–141 (2021)
8. Fahland, D.: Process mining over multiple behavioral dimensions with event knowledge graphs. In: van der Aalst, W.M.P., Carmona, J. (eds.) *Process Mining Handbook*, vol. 448, pp. 274–319. Springer (2022)
9. Hildebrandt, T.T., Mukkamala, R.R., Slaats, T.: Nested dynamic condition response graphs. In: Arbab, F., Sirjani, M. (eds.) *FSEN*. vol. 7141, pp. 343–350. Springer (2011)
10. Holzmann, G.J.: *The SPIN Model Checker - primer and reference manual* (2004)
11. Knaplesch, D., Reichert, M., Kumar, A.: A framework for visually monitoring business process compliance. *Inf. Syst.* **64**, 381–409 (2017)
12. Lin, J., Chen, P., Zheng, Z.: Microscope: Pinpoint performance issues with causal graphs in micro-service environments. In: Pahl, C., Vukovic, M., Yin, J., Yu, Q. (eds.) *ICSOC*. vol. 11236, pp. 3–20. Springer (2018)
13. Ly, L.T., Rinderle-Ma, S., Dadam, P.: Design and verification of instantiable compliance rule graphs in process-aware information systems. In: Pernici, B. (ed.) *CAiSE*. vol. 6051, pp. 9–23. Springer (2010)
14. de Murillas, E.G.L., Reijers, H.A., van der Aalst, W.M.P.: Data-aware process oriented query language. In: Polyvyanyy, A. (ed.) *Process Querying Methods*, pp. 49–83. Springer (2022)
15. Pérez-Álvarez, J.M., Díaz, A.C., Parody, L., Quintero, A.M.R., Gómez-López, M.T.: Process instance query language and the process querying framework. In: Polyvyanyy, A. (ed.) *Process Querying Methods*, pp. 85–111. Springer (2022)
16. Polyvyanyy, A. (ed.): *Process Querying Methods*. Springer (2022)
17. Schuster, D., Martini, M., van Zelst, S.J., van der Aalst, W.M.P.: Control-flow-based querying of process executions from partially ordered event data. In: Troya, J., et al. (eds.) *ICSOC*. vol. 13740, pp. 19–35. Springer (2022)
18. da Silva, E.A., do Nascimento Fidalgo, R., Ferro, M., Franco, N.: Visual query languages to design complex queries: a systematic literature review. *Softw. Syst. Model.* **22**(4), 1217–1249 (2023)