FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION FOR HIGHER EDUCATION
NATIONAL RESEARCH UNIVERSITY «HIGHER SCHOOL OF ECONOMICS»

Faculty of Computer Science

The School of Software Engineering

Jasurbek Tursunov

Name, Surname of the Student

Метод обнаружения анти-паттернов в журналах событий на основе правил

Project topic title in Russian

Method for Rule-based Anti-Pattern Detection in Event Logs

Project topic title in English

Version of Master Thesis Report (ultimate form)

Field of study 09.04.04 «Software engineering»

Program: System and Software Engineering

Student's signature Date of submission: 29.02.2024

Jasurbek Tursunov

Supervisor of the Graduate Work: Sergey Shershakov, PhD, assoc. prof., School of Software Engineering

Moscow, 2024

CONTENTS

ABSTRACT

Detecting anti-patterns [1] in event logs [2] plays a crucial role in ensuring the reliability and efficiency of processes in organizations. This study proposes a novel rule-based [3] anti-pattern detection method for event logs aimed at identifying deviations from the desired process behavior that may indicate inefficiency, errors, or compliance issues. The proposed approach uses predefined rules derived from domain knowledge and expert opinions that are applied to event log data to identify patterns indicating anti-patterns. By experimenting with real sets of event log data, the effectiveness and scalability of the method in accurately identifying different types of anti-patterns will be demonstrated. The proposed approach not only improves the understanding of process behavior, but also provides useful information for improving processes and managing compliance. The current development contributes to the improvement of anti-pattern detection methods in intelligent process analysis and offers practical solutions for organizations seeking to optimize their operational processes.

TERMS AND ABBREVIATIONS

Terms:

1. **Anti-pattern** is a common solution to a problem that generates decidedly negative consequences, often contrary to the intended purpose.
2. **Event Logs** store a chronological record of significant events that occur within a system or application.
3. **Rule-based System** is a software system in which decision making is based on a set of rules that determine the logic of the system's operation.
4. **System** in the context of this work means software being developed based on the method of detecting antipatterns in logs using rule cards.

Abbreviations:

1. **IS** – Information System.
2. **DSL** – Domain System Language.
3. **SQL** – Structured Query Language.
4. **DPI** – Deep Packet Inspection.
5. **SOA** – Service-Oriented Architecture.
6. **SODA** – Service Oriented Detection for Antipatterns.
7. **SOFA** – Service Oriented Framework for Antipatterns.
8. **QoS** – Quality of Service.

INTRODUCTION

In modern information systems, detecting antipatterns in event logs is crucial to ensure operational efficiency, system reliability, and user satisfaction. The purpose of this work is to solve this problem by proposing a method for detecting various antipatterns in event logs using predefined rule systems. These rules are described using *Domain System Language (DSL)* [4]. A DSL is a programming language with a higher level of abstraction optimized for a specific class of problems. A DSL uses the concepts and rules from the field or domain. That is, the user does not need to describe the rules in terms of low-level programming language in order to search for antipatterns.

The main motivation for this work is taken from real-world scenarios, such as the problems faced by the *European Travel Computer Reservation System (CRS)* [5]. The architecture of the information system considered in this context adheres to the *Service-Oriented Architecture (SOA)* approach, which is a sign of a modular, flexible, easily extensible service. SOA organizes software components into reusable services [6]. As a result, we get that IS consists of many interconnected business domains that are responsible for different product functionality. In turn, each domain consists of processes. Processes from different domains can transfer data between themselves. Each process or group of processes implements a service within a single domain. These services are private and unavailable for interaction with other domains. Given the well-implemented architecture of the system and all its complexity, this system faced problems related to efficiency and performance, which were partially explained by the appearance of anti-patterns resulting from the incorrect implementation of domains, processes and services. Such errors can compromise the stability and performance of the system. The proposed method uses predefined rules set by the end user to proactively detect such antipatterns, thereby reducing the risk of system failures. By automating the detection process, this method provides a proactive approach to identifying and eliminating potential problems before they escalate, which ultimately ensures the integrity and functionality of information systems. But to find such a vulnerability in the system, you need to consult the event log. This log stores all information about each interaction of local services, processes, and business domains. Due to the large volume of data and their complexity, manual analysis using filters, groupings and other methods is almost impossible. Automation is needed to find anti-patterns. In this context, anti-patterns represent deviations from the expected behavior of the system, signaling potential risks to the functionality and reliability of the system. Automating the detection of these inconsistencies using a rules-based approach simplifies the monitoring process and allows timely intervention to prevent system failures and financial losses.

The proposed method promises to increase the stability of information systems by providing systematic and effective means of identifying and eliminating antipatterns. By using a rules-based approach, this method can be adapted and applied in various information systems, offering a scalable solution to the problem of detecting inconsistencies in event logs.

In the end, we get a product that accepts a set of rules described using DSL and a system event log as input. The search algorithm itself interprets the rules into Python code to form a query in the form of SQL [7]. Then a search will be performed in the logs. The result will be given to the end user in the form of a system report. In the first chapter of this work, you will be able to get acquainted with the main purpose of the work and the tasks set for its implementation. An analysis and comparison of existing solutions in this area will be carried out, and the target audience of the product will be determined. The second chapter will be more about the technical component. It will demonstrate the basic concept of the work. Each component of the proposed product will be considered separately.

1 ANALYSIS OF DOMAIN

1.1 Project background

In the realm of information systems, the ability to efficiently manage and analyze operational data is crucial for maintaining system integrity and performance. The European Travel Computer Reservation System (CRS), a sophisticated platform based on Service-Oriented Architecture (SOA), serves as a running example of such a system. SOA ensures that the system remains flexible, scalable, and modular, with each service tailored to specific operational functions. This architecture is essential for reservation systems that handle real-time user requests and must integrate various services seamlessly [8].

The data for this project is derived from the event logs generated by the CRS. These logs record a wide array of system activities including errors, warnings, and other system messages. The comprehensive nature of these logs makes them an invaluable resource for detecting patterns and potential inefficiencies within the system operations.

The core objective of this project is to develop a method for detecting anti-patterns within these event logs. Anti-patterns, in this context, refer to repetitive errors or flaws in system architecture and logic that can lead to decreased performance and service quality degradation. To achieve this, we employ rules defined using a Domain Specific Language (DSL). DSL allows for the articulation of complex conditions and criteria that are specific to the domain of reservation systems. These rules are systematically applied to scan the logs, identifying any deviations from optimal operational patterns.

By focusing on the detection of anti-patterns, this project aims not only to enhance the operational efficiency of the European Travel Computer Reservation System but also to ensure higher quality of service and reliability of other companies. Through the continuous monitoring and analysis of event logs, potential issues can be addressed proactively, thereby mitigating risks associated with system failures and downtime.

The following section, Motivation and Problem Statement, will delve deeper into the specific challenges and motivational factors driving the need for an anti-pattern detection method within the CRS logs.

1.2 Motivation and Problem Statement

The motivation behind this project stems from the critical need to enhance the operational efficiency and reliability of products like CRS. As a sophisticated reservation platform operating on a Service-Oriented Architecture, CRS handles an immense volume of transactions daily. Each transaction generates data that is captured in event logs, which, if analyzed effectively, can provide deep insights into the system's operational health.

Challenges in Current Log Analysis Approaches:

1. **Volume and Complexity of Data:** The vast amount of log data generated by CRS is both a valuable resource and a significant challenge. The sheer volume and complexity of this data often obscure underlying patterns, making manual analysis impractical and error-prone.

2. **Detection of Anti-Patterns:** Anti-patterns in CRS can manifest as redundant processes, inefficient transactions, or recurring errors. These issues not only degrade the user experience but also strain resources, leading to higher operational costs and reduced system responsiveness.

3. **Dynamic Nature of Services:** The modular and interconnected structure of CRS services means that inefficiencies in one module can propagate errors across the entire system. Identifying the root causes of such systemic issues is critical for maintaining system integrity.

Objective of Anti-Pattern Detection:

The primary goal of this project is to develop a robust method for detecting anti-patterns in the event logs of CRS. By leveraging rules defined in a Domain Specific Language (DSL), this method will enable the automated identification of problematic patterns that indicate inefficiencies or errors in system operations. The objectives include:

1. **Improving System Performance:** By systematically identifying and addressing

anti-patterns, the CRS can operate more efficiently, reducing the time and resources spent on resolving issues.

2. **Enhancing Service Quality:** Effective anti-pattern detection will minimize downtime and errors, thereby improving the overall user experience and reliability of the CRS.

3. **Scalability and Adaptability:** The proposed method will be designed to easily integrate with existing systems and be adaptable to handle increasing amounts of data as CRS evolves.

It is expected that the implementation of this method of eliminating patterns will significantly increase the efficiency of the CRS. This will not only reduce the frequency and severity of system failures, but also pave the way for better analytics and automation capabilities in the future. By addressing the root causes of operational inefficiency, CRS can provide high throughput and improved service delivery, which is crucial in a competitive environment of computer reservation systems. The next chapter will be devoted to the target audience, who will be able to integrate the proposed solution into their finished product.

1.3 Intended Audience

The development and implementation of the anti-pattern detection method designed for the European Travel Computer Reservation System (CRS) is envisaged to serve a broad spectrum of stakeholders within and potentially beyond the organization. This section delineates the primary audiences who stand to benefit from the project's capabilities and innovations.

If we consider this work within the framework of CRS, then the target audience will be different departments of the company. These include: *System Architects and Developer, Operations Management Team, Quality Assurance Professionals, Customer Service Representatives, IT Security Analysts, Senior Executives*. Although the teams are different, they all have the same goal: to use the information obtained as a result of pattern detection to improve and optimize the product. But each team benefits in its own way. For example, a team of engineers will be able to implement a more reliable and scalable, easily managed system. The operational management team will receive improved monitoring of the system's performance. The quality assurance team will increase the reliability and performance of the system, which will lead to improved end-user experience. And of course, senior managers will be able to make informed decisions in accordance with the strategic goals of the company.

If we consider the target audience not within the framework of the CRS, then they may be other organizations in the field of information technology. Their goal will be to implement an anti-pattern detection method to improve the performance and reliability of their own systems. Research institutes and communities can also be the end users of this solution. They will be able to analyze and confirm the effectiveness of the methodology as a universal system analysis tool. And if successful, they will promote this solution in the field of system analytics and pattern detection, contributing to wider adoption and innovation.

The target audience of this anti-pattern detection project is a wide range of stakeholders, ranging from employees performing internal functions within European tourism systems to potential third-party developers in various industries. By adapting the results of the project to the diverse needs of these groups, the initiative not only improves internal operations, but also sets standards for best practices in the field of system maintenance and analytics in various sectors. To understand how competitive the proposed solution is, the next chapter will analyze existing solutions and compare them.

1.4 Analysis of Existing Systems

Anti-pattern detection in event logs is an emerging field in software engineering, specifically in Service Oriented architecture (SOA). This section will review the main and most significant tool and method that has made significant contributions to this field, developed by Francis Palma, Mathieu Nayrolles, Naouel Moha, and others [9]. This material reflects an approach to the definition and identification of anti-patterns, which has unique advantages and a technological basis.

An article by Palma et al. entitled **"SOA Antipatterns: an approach to their specification and detection"** is devoted to solving problems related to the detection and definition of antipatterns in service-oriented systems (SBS) using service-oriented architecture (SOA) [10]. The approach and systems involved in the study will be described in detail below

- the SODA approach (Service Oriented Detection for Anti patterns) and the SOFA approach (Service Oriented Framework for Antipatterns).

1.4.1 Overview of the SODA Approach and SOFA Framework

SODA Approach

1. SODA stands for Service Oriented Detection for Antipatterns, an innovative approach designed to specify and detect antipatterns in SBSs.

2. It utilizes a domain-specific language (DSL) to allow high-level specification of antipatterns, which are common poor solutions in the architecture of SBSs that degrade their design and quality of service (QoS).

3. SODA supports both static and dynamic analyses of SBSs. Static analysis involves examining structural properties of the system, while dynamic analysis involves runtime properties associated with QoS.

SOFA Framework

1. SOFA supports the SODA approach by providing a robust environment to implement the detection algorithms derived from the antipattern specifications.

2. The framework facilitates the automation of detection algorithm generation, making the process less error-prone and more efficient.

3. SOFA utilizes advanced modeling techniques and tools like Ecore and Acceleo to ensure the rule cards that define antipatterns are properly parsed and implemented.

Applications and Validations

The paper details the application of the SODA approach and the SOFA framework in two distinct systems:

1. **Home-Automation System:** A smaller SBS with 13 services, developed independently by two Masters students.

2. **FraSCAti:** An open-source implementation of the Service Component Architecture (SCA) standard, which is considerably larger, comprising over 100 services.

The validation of the SODA approach demonstrated **high precision (90%)** and **recall (97.5%)** across these systems, indicating that the approach is highly effective in real-world scenarios.

Extensibility and Performance

1. The approach is noted for its extensibility; the DSL can be expanded to include new metrics and antipattern definitions as required by emerging needs in SBS design.
2. The performance analysis highlighted that the computation times for the detection of antipatterns are reasonably low, facilitating quick feedback for systems in development and production environments.

This detailed examination of the work by Francis Palma et al. on "SOA Antipatterns: An Approach for Their Specification and Detection" showcases a comprehensive and specialized effort in formalizing the detection and specification of antipatterns in service-oriented systems. The use of a domain-specific language (DSL) for specifying antipatterns highlights the complexity of their approach, which might require significant time investment from users to learn and effectively apply the syntax. This complexity underpins the necessity for users to possess a deep understanding of DSL design to fully leverage the framework's capabilities.

In contrast, work aims to streamline the user experience by utilizing JSON for rule specification, which can simplify the entry barrier for users unfamiliar with complex DSL syntax. Additionally, while Palma's framework incorporates elements of neural networks, adding both strengths and potential unpredictabilities, our approach, focusing on SQL implementations, seeks to provide a more straightforward and perhaps more predictable method for rule-based anti-pattern detection. This divergence in methodologies highlights the practical applicability and accessibility of your research compared to the existing frameworks.

1.5 Requirements Analysis

To ensure the successful development and deployment of the anti-pattern detection system for CRS and for other companies, it is critical to define clear functional and non-functional requirements [11]. This analysis will help in structuring the development process and ensuring that the final product meets the desired specifications.

1.5.1 Functional Requirements

Functional requirements define what the system should do. They include tasks, data manipulation and processing, as well as other specific functionality that helps users perform their actions.

1. **Data organization:** The system must systematize the collected logs into a structured database that provides effective query and further analysis.
2. **Development of the DSL interpreter:** Develop an interpreter for Domain System Language (DSL) that converts business rules into executable conditions in a programming language.
3. **Anti-pattern detection:** Implementation of an algorithm for detecting anti-patterns in structured data using previously defined rules.
4. **Interface Functionality:** Provides a user-friendly interface that allows users to define new rules, modify existing ones, and view reports.
5. **Reporting:** The output of the proposed product should be a report showing the presence of antipatterns in the system.

1.5.2 Non-functional Requirements

Non-functional requirements specify how the system performs certain functions and include system properties such as performance benchmarks, usability guidelines, and scalability factors. The following are critical non-functional requirements for the anti-pattern detection system:

1. **Performance:**
    - The system must process logs and detect anti-patterns in real-time with minimal latency.
    - It should handle high volumes of data and intensive query execution without degradation of system performance.
2. **Scalability:**
    - The system must scale effectively to accommodate increasing amounts of data and more complex queries as system usage grows.
3. **Reliability:**
    - High availability and minimal downtime are critical.
    - The system should perform consistently under varying loads and when processing different types of anti-patterns.
4. **Usability:**
    - The interface must be intuitive and require minimal training for new users.
    - Comprehensive documentation and accessible support resources should be provided.
5. **Maintainability:**
    - The system should be easy to update and maintain, with modular components that can be independently enhanced or replaced.
    - It should support efficient debugging and troubleshooting processes.
6. **Integration:**
    - The system should integrate seamlessly with existing CRS infrastructure without significant modifications.
    - It should support integration with other systems and external data sources as needed.

The above functional and non-functional requirements are intended to guide the development of an antipattern detection system. By meeting these requirements, the system will not only increase the efficiency of integrated systems, but also ensure its reliability, scalability and ease of use. This comprehensive analysis of the requirements sets a clear framework for the next section, which will form the purpose of the work and its subtasks that need to be completed in order to achieve the final goal.

1.6 Goals and Objectives

The primary goal of this project is to develop a comprehensive method for rule-based anti-patterns detection in the event logs. This initiative is aimed at enhancing operational efficiency and reliability by systematically identifying and addressing system inefficiencies and errors. Below are the specific objectives that guide the development and implementation of this detection method:

1. **Log Acquisition and Database Presentation:** Acquire logs from information system events into a database to facilitate access and analysis.
2. **Identification of Common Anti-Patterns:** Identify and define a basic set of common anti-patterns to establish a benchmark for operational inefficiency detection.
3. **Development of a DSL Interpreter:** Create an interpreter [12] for Domain System Language (DSL) to enable the translation of rules from high-level interpretation to programming language.
4. **Query Development:** Develop a querying mechanism that applies DSL-based rules to efficiently search and analyze data within the event database.
5. **User Interface Creation:** Design a user-friendly external interface that allows users to easily configure, operate, and extract insights from the detection system.
6. **Performance Metrics and Effectiveness Analysis:** Measure and evaluate the metrics and effectiveness of the detection method to ensure reliability and operational improvement.

After achieving these goals, we will receive a final project that will help us significantly improve error detection and management, thereby improving operational performance, reducing downtime and increasing user satisfaction with various information systems. This solution will not only benefit CRS, but also has the potential for wider application in other similar systems. The next chapter of this work will be devoted to the technical component of the product being developed.

2 METHODOLOGY AND SYSTEM DESIGN

2.1 System Design

Fig.1 shows the basic concept of the proposed system, which is designed to detect anti-patterns in event logs using rules. The system works through a multi-step process, integrating several key components together to produce useful log-based information. The system design [13] is described as follows based on the provided diagram:

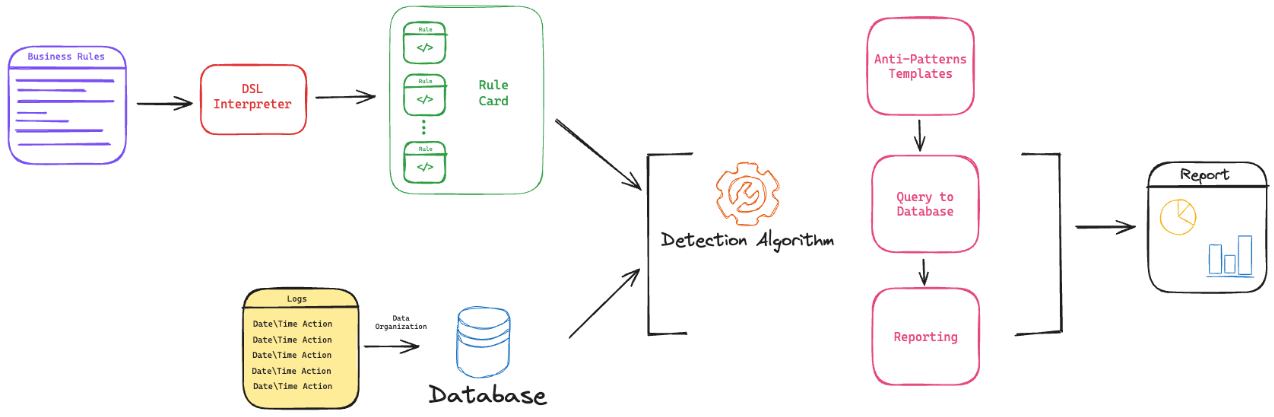


Figure 1: The basic concept of work.

Component overview

1. **Business Rules:** Business rules [14] are one of 2 *input parameters* of the system, which describes the rules at a high level for which the system may have performance vulnerabilities.

2. **DSL Interpreter:** The DSL interpreter is one of the main and important components of the system, responsible for translating high-level business rules into rule cards described using a programming language. These rules define patterns that are considered indicators of potential anti-patterns in the system.

3. **Rule Cards:** Each rule card is an encapsulated set of instructions received from the DSL interpreter. They are intended to be applied directly to a database for query and analysis purposes.

4. **Logs and Databases:** The second *input element* of the system is the information system logs. These logs are the most common ones, which contain information such as date, time, payload, response status and much more. This data is structured into a database to optimize data retrieval and query execution.

5. **Detection algorithm:** The heart of the system is the detection algorithm. This complex component uses a set of rule cards to scan a database, identifying and classifying anti-patterns based on predefined patterns.

6. **Anti-Pattrens Templates:** These patterns provide predefined, generic queries to identify deviations from normal patterns. They are essential for recognizing a variety of antipatterns, from simple anomalies to complex sequences that indicate deeper systemic problems.

7. **Query to Database:** This component is important because it searches the database for antipatterns.

8. **Reporting:** The last component of the system is the reporting module. It generates detailed reports and visualizations of detected antipatterns, presenting them in a format that is easy for users to interpret and understand.

Data Flow

Antipattern detection process begins with the DSL Interpreter translating business rules into rule cards(see section 2.3.4). These rule cards are then utilized by the Detection Algorithm to query the database for potential anti-patterns. Upon successful detection, the relevant data is compiled into reports that are both informative and easily digestible by the end-users.

System Interaction

Each component is designed to interact seamlessly with the others. For instance, the rule cards are created in a format that is directly executable by the Detection Algorithm, ensuring smooth data flow. The database is structured to support the complex queries generated by the

Detection Algorithm, and the reporting tools are integrated to provide immediate feedback on detected anti-patterns.

User Interface

Design of the developed system includes a user-friendly interface that allows for the easy creation and modification of business rules, initiation of the detection process, and viewing of reports. It is designed to be intuitive, requiring minimal technical expertise to operate.

Architecture of the developed system facilitates efficient detection of anti-patterns in event logs, transforming complex business rules into actionable queries and insights. With a robust database, intelligent detection algorithm and user-centric reporting tools, the system can improve operational efficiency and help in proactive system management.

2.2 System Architecture

Architecture[15] of the developed system for event logs is a comprehensive framework designed to facilitate the seamless translation of business rules into useful information for system optimization. The architecture is modular, with each component serving a specific function and working in harmony with the others. The next section will be devoted to the architectural principles of operation.

Architectural principles

The project follows several key architectural principles:

1. **Modularity:** Each component, such as the DSL interpreter, rule map, detection algorithm, and reporting module, is a separate unit that can be updated independently.
2. **Scalability:** The system can handle growing volumes of event logs and complex queries without significant changes to its architecture.
3. **Interoperability:** Each module is designed to work with different data sources and formats, ensuring compatibility with other systems.

Technology Stack

The system will use a combination of technologies such as SQL for database operation, an internal language such as Python [16] for the detection algorithm, and an external environment. The technology stack will be discussed in more detail a little later, in the following sections.

System architecture is the design of a scalable and interoperable system. It is designed to effectively solve the problems of detecting anti-patterns in event logs and serve as a reliable tool for system analysts and administrators to maintain optimal system performance.

2.3 Methodology

2.3.1 Service-Oriented Architecture

Service-Oriented Architecture (SOA) is a design paradigm that organizes software components into reusable services, which can be accessed and combined to fulfill specific business functions [6]. In SOA, services are loosely coupled and interact through standard communication protocols, making it easier to integrate disparate systems and adapt to changing business requirements.

SOA has gained popularity in big information systems due to several key factors. Firstly, its *modular and reusable nature* allows organizations to break down complex functionalities into smaller, manageable components, facilitating easier development, maintenance, and evolution of large-scale systems. Secondly, SOA promotes *seamless integration* by providing standardized interfaces for communication and interaction between disparate systems and technologies, enabling efficient data exchange and collaboration. Additionally, SOA architectures offer scalability and flexibility, allowing systems to scale horizontally to accommodate growing volumes of data and users while remaining agile and responsive to changing business needs. The capability for dynamic service discovery and composition further enhances agility by enabling real-time assembly of complex business processes from reusable services [8].

The IS as the running example has an SOA architecture. CRS is a distributed software containing various client and server components. Fig.2 visualizes a more detailed structure of the product. As can be seen from the figure, at a high level we have business domains that implement separate business logic and do not interact with each other in any way. For example, there are business domains “booking” and “accounting". Processes are grouped into **process groups (PG)**, and services are grouped into **technical domains (TD)**. Returning to the previous example, there are PG and TD named “booking". Processes and services located in the same domain can interact with each other. But from outside, only process is available. That is, we get that processes in one domain can communicate with processes in another domain. Technically, all processes and services are implemented as interfaces and packages of a certain programming language. Each process and service sends and receives messages (corresponding to OperationName), and each specific message is logged as a separate trace in the log.

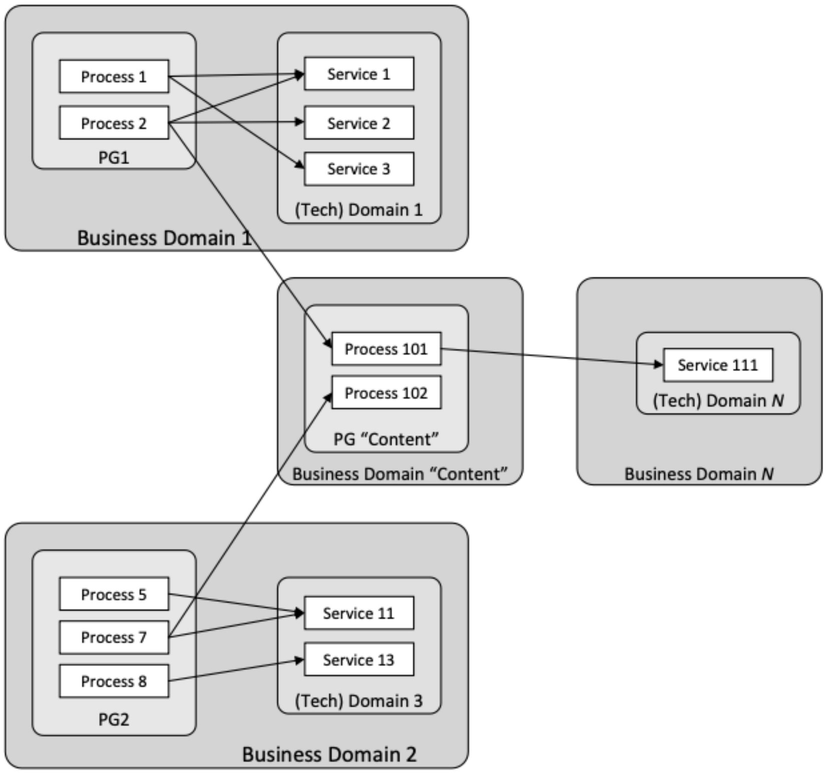


Figure 2: “Domains-Processes-Services” architecture scheme.

We have a set of logs that track the relationships between these processes and services and contain many different aspects of the data. By studying certain data representations, we can look at the system from different perspectives. Each particular view can represent some aspect of the system. We highlight aspects such as the control flow aspect, the data perspective, the organizational aspect and the infrastructural context.

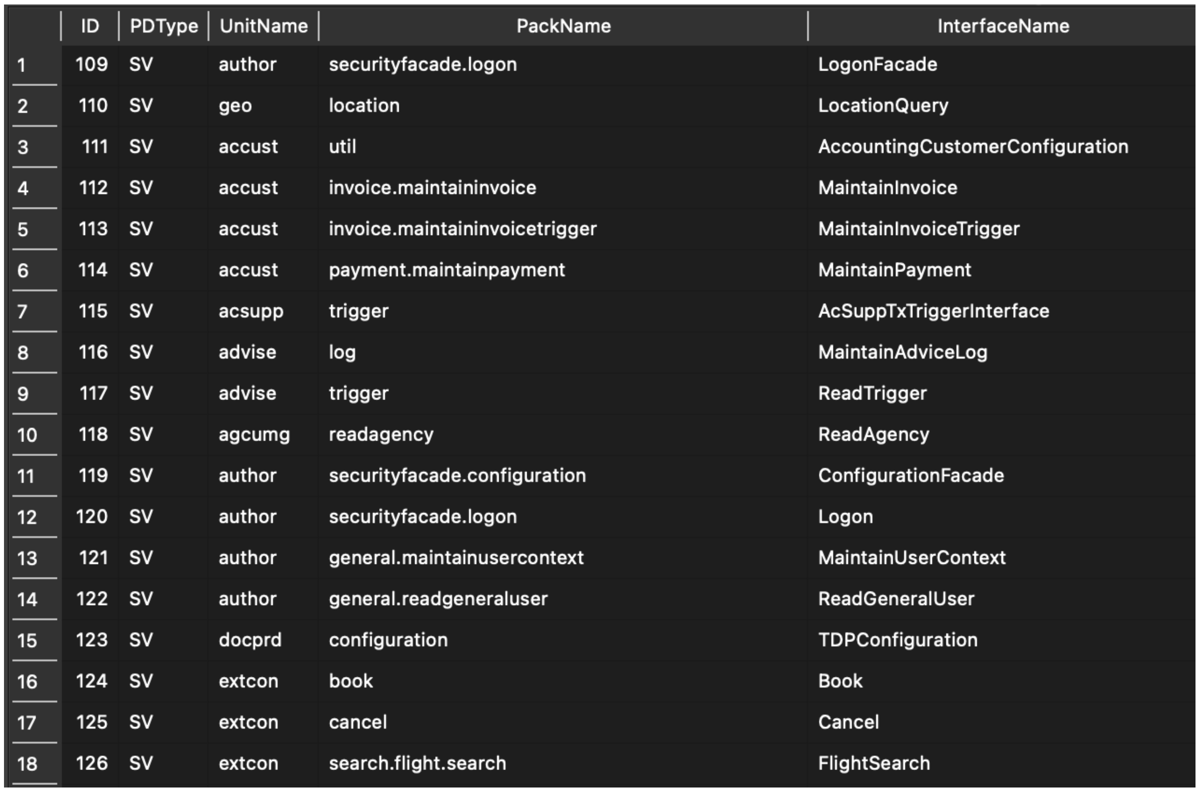
In this section, we have considered what an information system based on SOA consists of. We looked at a basic example of how business domains, processes, and services interact with each other, and confirmed the fact that each interaction of these components is logged in an event log that we can use for analysis. More details about the contents of the logs will be mentioned in the next section.

2.3.2 Event Logs and Tools

Informally, event log is a chronologically ordered record of events occurring within a system or process, often used for troubleshooting and analysis purposes [2]. It can include various types of events, such as errors, warnings, information messages, and user actions. Each event is typically time stamped and includes additional information such as the source of the event and any relevant data associated with the event.

Subsystems of CRS support the ability to track the interaction of all necessary processes and services. The initial call of any process, for example, made by an application with an extended client interface, is accompanied by the allocation of a special call identifier (*also called InvID*). Both **processes (PR)** and **services (SV)** receive a **request message (RQ)** as input and the returned **response message (RS)** or **exception (SE)** as output. These messages are logged. At the log level, traces are recorded in XML format [17]. For the convenience of working with data, these events have been overwritten [18] in SQLite3 format [19]. An example of such an event is given in the Fig.3.

Each trace is included in a tracing event element that contains several sections describing the trace and including additional data that can be used for more in-depth analysis. For our first step, we are interested in the following data: **UnitName, PackName, Interface-Name**. Then we will also look at two very important fields. First, the timestamp of the event. In describing the second, we should mention that the trace event contains a payload specified in a form used by processes and services to exchange data with each other. The useful data is presented as a piece of data and can be used to analyze the model created



	ID	PDType	UnitName	PackName	InterfaceName
1	109	SV	author	securityfacade.logon	LogonFacade
2	110	SV	geo	location	LocationQuery
3	111	SV	accust	util	AccountingCustomerConfiguration
4	112	SV	accust	invoice.maintaininvoice	MaintainInvoice
5	113	SV	accust	invoice.maintaininvoicetrigger	MaintainInvoiceTrigger
6	114	SV	accust	payment.maintainpayment	MaintainPayment
7	115	SV	acsupp	trigger	AcSuppTxTriggerInterface
8	116	SV	advise	log	MaintainAdviceLog
9	117	SV	advise	trigger	ReadTrigger
10	118	SV	agcumg	readagency	ReadAgency
11	119	SV	author	securityfacade.configuration	ConfigurationFacade
12	120	SV	author	securityfacade.logon	Logon
13	121	SV	author	general.maintainusercontext	MaintainUserContext
14	122	SV	author	general.readgeneraluser	ReadGeneralUser
15	123	SV	docprd	configuration	TDPConfiguration
16	124	SV	extcon	book	Book
17	125	SV	extcon	cancel	Cancel
18	126	SV	extcon	search.flight.search	FlightSearch

Figure 3: A table containing service interaction events.

using this log from a data perspective.

The TracingEvents table has a lot of interesting event data besides the payload. Below are all the fields that are contained in the table:

- **ID**: A unique identifier for the interface.
- **PDType**: The type of protocol or data format used.
- **UnitName**: The name of the unit associated with the interface.
- **PackName**: The package name associated with the interface.
- **InterfaceName**: The name of the interface.
- **ID**: A unique identifier for the event.
- **Interface_ID**: The identifier linking the event to a specific interface.
- **OperationName**: The name of the operation or action associated with the event.

- **InvID**: The identifier associated with the event invocation.
- **InvNodeName**: The name of the node where the event invocation occurred.
- **InvIP**: The IP address associated with the event invocation.
- **TransContext_ID**: The identifier linking the event to a transaction context.
- **AppServerContext_ID**: The identifier linking the event to an application server context.
- **EventType**: The type of event, such as error, warning, or informational.
- **EventTimestamp**: The timestamp indicating when the event occurred.
- **PayloadSize**: The size of the payload associated with the event.
- **PayloadCRC32**: The cyclic redundancy check (CRC32) value for the payload.
- **EventSeqNum**: The sequence number of the event within the log.

One of the suggested anti-pattern search options is payload analysis among TracingEvents. That is, we will determine whether the size of her data and her hash amount change from call to call. You can also analyze the data for overloading services. When services interact, one service communicates with another until the first one receives the necessary information from the second. Sometimes, with incorrect software implementation of such interaction of services, calls are overloaded, which can have a bad effect on efficiency, and in the worst case, on its inactivity. To find such antipatterns in the logs, it is proposed to analyze the events for the number of calls.

Such proposed solutions describe only the general concept of what the anti-pattern detection method is focused on and how it will work. In order for the proposed method to be universal and convenient for the end user, it is proposed to describe the rules by which anti-patterns will be searched in different information systems. For more information about what such rules are, how to create and describe them, see the next section

2.3.3 Rule-Based Systems and Domain-System Languages

In ISs, a rule-based system is a paradigm for encoding and executing logical rules to automate decision-making or problem-solving tasks [3]. These systems rely on a set of rules expressed in a formal language that defines conditions and actions. Each rule typically consists of an antecedent (condition) and a consequent (action), where the action is executed if the condition is satisfied. Rule-based systems are widely used in various domains, including business process management, expert systems, and workflow automation.

One of the key advantages of rule-based systems is their declarative nature, which separates the logic of the rules from the control flow of the program. This enables domain experts to define and modify rules without extensive programming knowledge, making rule-based systems accessible and adaptable to changing requirements. Additionally, rule-based systems facilitate transparency and traceability, as the logic governing decision-making is explicitly defined in human-readable form.

Domain-System languages (DSLs) are programming languages adapted to meet the specific needs and objectives of a specific domain or problem space [4]. Unlike general-purpose programming languages, which are designed to be universal and applicable in various domains, DSLs are optimized to express concepts, abstractions, and operations within a specific domain.

In rule-based systems, DSLs are often used to define the syntax and semantics of rules depending on the domain. These DSLs provide a higher level of abstraction that closely aligns with users domain experience, allowing them to express rules in a natural and intuitive way. By encapsulating domain-specific concepts and designs, DSLs enhance the readability, maintainability, and usability of rule-based systems. DSLs in rule-based systems can offer specialized constructs to define conditions, actions, constraints, and relationships related to a specific subject area. These DSLs can also provide mechanisms for modularization, layout, and reuse of rule components, making it easier to develop and manage complex rule sets.

In this section, we have gained a basic understanding of systems with rules, as well as about DSL, which is responsible for the implementation of such a system. The next section will describe the basic principles of the proposed method of detecting anti-patterns in logs.

2.3.4 DSL Interpreter

Describing Rules Using DSL

Describing business rules usually does not require deep technical knowledge. It is enough to have an idea of the IS for which the rules will be described. Therefore, when compiling the syntax of the rules described using DSL, this important point was taken into account.

The syntax for this rule is shown in Listing 1. Each rule begins with the *rule* keyword, followed by the rule name in quotation marks. The *body* of the rule is described in curly braces and consists of a condition and an action. Conditions are specified with *when* keyword. This block checks the conditions under which the rule should be activated. Actions are specified with *then* keyword. This block defines the actions that must be performed if the conditions of the rule are met. In our case, this is the *makeReport()* function, which generates a report.

Listing 1: Syntax of a rule described using DSL

```
rule "Long_Response_Time_Rule" {
  when
    ServiceA.RT > 100;
  then
    makeReport();
}
```

When describing the conditions for executing a rule, you can use various arithmetic and logical operations (>, <, ==, !=). The *left* operand must consist of a single expression separated by a dot. To the left of the dot, the name of the service, module, process or domain in which you want to search is described. To the right of the dot, you must indicate one specification (see section 3.2). The right operand can be an nteger, float, string, or boolean value. Using the example shown in Listing 1, we specify a rule in which the response time of service A is more than 100 (milliseconds). The concept of a rules card using DSL represents several rules, an example of one of which we looked at above. Listing 2 shows an example of a rule card.

Currently, the system being developed can only generate a report. Therefore, each rule only has a *makeReport()* event in the *then* block. A report will not be generated for every rule. It will be done for the entire rule card.

Listing 2: Syntax of a rule card using DSL

```
rule "Long_Response_Time_Rule" {
  when
    ServiceA.RT > 100;
  then
    makeReport();
}

rule "High_Coupling_Rule" {
  when
    ServiceB.CPL = true;
  then
    makeReport();
}

rule "Low_Availability_Rule" {
  when
    ServiceC.A = false;
  then
    makeReport();
}
```

Translating Rules into Rule Card

DSL Interpreter performs the function of converting rule cards described using DSL into a rule card in JSON, which will be used by the developed system to search for antipatterns in logs (see section 3.3). Why do you need such a card? Firstly, a card described using DSL is not convenient from a technical point of view. In order not to encounter consistency problems in the developed system, the rules will be described in JSON. On the card shown in Listing

2, you can see fairly simple rules, which individually do not provide useful information for developers. That is, if the rule states that *service A* has a *long response time* and that *service B* has *high coupling*, then such a statement will not give us cause for alarm. The combination of such rules is precisely a sign of an *antipattern*. That is, if we add the *low availability* rule for service B to the rules described above, then we get a ready-made *BottleneckService* antipattern. DSL Interpreter not only performs the conversion function from DSL to JSON, but also supplements the rules with any other suitable rules if they ultimately form a ready-made antipattern. Why did this role fall on Interpreter? The end user does not need to know about the concept of an antipattern, its types and what it consists of. To make the user's life easier, the process of initialization and generation of a ready-made rules card, which formally will contain information about the antipattern, will be performed in DSL Interpreter.

2.4 Algorithm for Translating a Rule Card into an SQL Query

This subsection outlines an algorithm designed to translate a rule card, provided in JSON format, into an SQL query. The purpose of this algorithm is to generate an SQL query that identifies specific anti-patterns in log data, based on the rules defined within the rule card.

Input and Output

The input to the algorithm is a JSON object representing the rule card. An example is provided in section 3.3.2. The output of the algorithm is an SQL query that searches for the anti-pattern specified by the rule card. For the example input(Fig. 10), the output would be like in Listing 3.

Listing 3: SQL Query for NobodyHome Antipattern

```sql
SELECT OperationName, EventType, count(*) as nir, avg(PayloadSize) as nmi
from FullData3
WHERE EventType == "RS"
Group by OperationName,  EventType
HAVING nir <= ordi_value AND nmi > ordi_value
ORDER by nir, nmi DESC
```

The IS being developed contains a list of antipatterns that the system can find from logs. Taking into account the fact that different information systems, which in the future will implement the product being developed, have different log structures. In order for the system to function correctly and find antipatterns in the logs, before the first launch of the system, it is necessary to initialize the names of key parameters in the logs. For example, the parameter storing the payload can be called *Payload*, *PayloadSize* or *Volume*. To avoid confusion, the end user must configure the system being developed in advance.

Algorithm Overview

The algorithm consists of the following steps:

1. **Parse the JSON Input:**
    - Extract the primary rule card name.
    - Extract the list of rules.
2. **Initialize SQL Components:**
    - Initialize the `SELECT` clause to specify the columns to be selected.
    - Initialize the `FROM` clause to specify the table involved.
    - Initialize the `WHERE` clause to specify the conditions based on the rules.
3. **Generate Conditions Based on Rules:**
    - For each rule, generate the corresponding SQL condition based on the rule's `metric_id` and `ordi_value`.
    - Handle complex conditions where multiple rules are combined using logical operators (e.g., `INTER` for intersection).
4. **Combine Conditions:**
    - Combine individual rule conditions using appropriate logical operators.
5. **Construct the SQL Query:**

- Combine all the SQL components into a single SQL query.

This algorithm ensures that the SQL query accurately reflects the conditions specified in the rule card, enabling effective identification of anti-patterns in the logs.

2.5 Workflow of the Development System

This section will describe the basic principle of the method proposed in this paper for detecting anti-patterns in event logs. Earlier, we learned what an event log is and what logs it consists of. We looked at an example of data from the railway ticket booking system. We also figured out what a rule-based system is, and what tools are needed for this. Fig.4 shows the general scheme of the system, which was obtained on the basis of all the knowledge obtained just above.

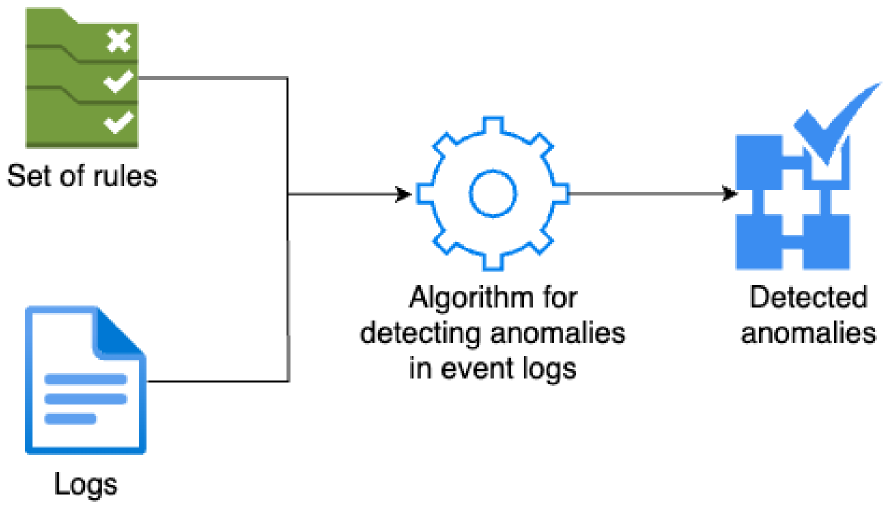


Figure 4: The basic principle of the system operation.

The operation of the system can be described in more detail as follows: a set of rules and logs is provided as input to the antipattern detection method. The rules are described using a DSL. The number of rules is limited. Logs need to be transferred and grouped into DB-based format such as *SQ3EventLog*. These parameters are then passed as input to the antipattern detection method. A method based on these rules executes a complex SQL query against a database. Next, the database is analyzed for the content of antipatterns. At the end of the search based on the results obtained by the method, the system issues a report on the work done, which will describe in detail the search results for each rule.

This system is implemented as a framework in Python [16]. This choice of programming language is associated with the convenient and flexible ability to integrate into various information systems. The next section will be devoted to the software implementation of the product being developed.

3 SYSTEM IMPLEMENTATION

3.1 Tools and Technologies

The development of the anti-pattern detection system was underpinned by a selection of technologies chosen for their reliability, flexibility, and performance. This section outlines the technologies employed and the rationale for their use.

JSON

JSON is used to describe the rules and metrics of antipatterns. This provides a number of advantages over using a domain-specific language (DSL), such as the one used by Francis Palma and co-authors.:

- **Accessibility and Familiarity:** JSON is widely used and understood in the software development community due to its simplicity and compatibility with various data formats. It's easier for developers who are already familiar with JSON to adopt and integrate it into existing systems without the need to learn a new syntax, as would be required with a DSL.
- **Interoperability:** JSON is a language-independent data format, making it highly interoperable across different platforms and programming environments. This flexibility allows our rule-based detection system to be more easily integrated with other applications and systems, which might not be the case with a specialized DSL.
- **Ease of Use:** JSON's structure is both human-readable and machine-parsable, facilitating easier debugging, testing, and maintenance of the rule definitions. Developers can quickly understand and modify the JSON-described rules without navigating the complexities of a more esoteric DSL syntax.
- **Tooling and Support:** There is robust tooling available for JSON processing, including parsers and libraries across all major programming languages, which can significantly reduce the time and effort required to implement and manage the system.

- **Scalability:** JSON's lightweight nature makes it well-suited for applications that require high performance and scalability. Managing and updating JSON files is generally less resource-intensive compared to processing DSLs, which can sometimes involve more complex parsing mechanisms.

Python

At the heart of our system's DSL interpreter and detection algorithm lies Python, a language chosen for several key reasons:

- **Versatility:** Python's extensive range of libraries and frameworks enables rapid development of complex applications.
- **Readability:** The language's clear syntax promotes a maintainable and comprehensible codebase, essential for the project's long-term viability.
- **Community Support:** A robust community and extensive documentation ensure that solutions for potential issues are easily accessible.
- **Integration Capabilities:** Python's ability to work seamlessly with other technologies and services made it an optimal choice for a system requiring interaction with a variety of components and data formats.

SQL

SQL was selected for database querying due to its powerful data retrieval and manipulation capabilities, being particularly well-suited for managing structured data in relational databases. The system's need for complex query execution and large-volume data handling rendered SQL an essential component of the technology stack.

- **Efficiency:** SQL's data manipulation and querying proficiency allows for rapid searches across extensive datasets, pivotal for real-time anti-pattern detection.
- **Universality:** As a standard that is widely adopted, SQL ensures database operations are compatible with various database management systems.

- **Scalability:** The scalability of SQL databases aligns with the requirement to handle an expanding volume of event log data.

Integration and Deployment

The integration of the Python logic with SQL querying was facilitated through a dedicated interface, ensuring seamless communication between the application layer and the database. The architecture was designed to support deployment in diverse environments, accommodating both on-premises and cloud-based infrastructures.

The synergistic use of Python and SQL forms a solid foundation for the anti-pattern detection system. The flexibility and user-friendliness of Python, combined with the robust data handling capabilities of SQL, result in a powerful yet adaptable system, ready to meet the dynamic requirements of the European Travel Computer Reservation System (CRS).

3.2 Antipatterns and Specifications

In the context of work, antipatterns refer to common, repetitive incorrect practices or design flaws in the architecture of service-based systems that can lead to reduced performance, reduced usability, increased complexity, or other negative consequences. In fact, they represent "bad solutions" to common problems where better solutions are known to exist. In this work, antipatterns generally accepted by various scientific communities [9] were used, such as:

1. *Bottleneck Service* is a service that is highly used by other services or clients. It has a **high incoming and outgoing coupling**. Its **response time** can be **high** because it may be used by too many external clients, for which clients may need to wait to get access to the service. Moreover, its **availability** may also be low due to the traffic.
2. *Chatty Service* corresponds to a set of services that exchange a lot of **small data of primitive types**, usually with a **Data Service** antipattern. The Chatty Service is also characterized by a **high number of method invocations**. Chatty Services chat a lot with each other [20].

3. *Data Service* in Object-Oriented [21] systems, corresponds to a service that contains **mainly accessor methods**, i.e., getters and setters. In the distributed applications, there can be some services that may only perform some simple information retrieval or data access to such services. Data Services contain usually **accessor methods** with **small parameters of primitive types**. Such service has a **high** data **cohesion**.

4. *Duplicated Service*, introduced by IBM, corresponds to a set of **highly similar services**. Because services are implemented multiple times as a result of the silo approach, there may have **common** or **identical methods** with the **same names and–or parameters** [22].

5. *LostData Service* antipattern appears when data essential for the completion of business processes is misplaced or lost during transitions between services. It typically indicates poor data handling or integration practices within the system.

6. *Multi Service* also known as *God Object* corresponds to a service that implements a **multitude of methods** related to different business and technical abstractions. This service aggregates too many methods into a single service, such a service is not easily reusable because of the **low cohesion** of its methods and is often **unavailable** to end-users because of it is **overloaded**, which may also induce a **high response** time [20].

7. *Nobody Home* corresponds to a service, defined but actually never used by clients. Thus, the methods from this service are **never invoked**, even though it may be **coupled** to other services. Yet, it still requires deployment and management, despite of its non-usage [23].

8. *Service Chain* [21] in Object-Oriented systems corresponds to a **chain of services**. The Service Chain appears when clients request consecutive service invocations to fulfill their goals. This kind of **dependency** chain reflects the subsequent **invocation** of services.

9. *The Knot* is a set of **very low cohesive** services, which are **tightly coupled**. These services are thus less reusable. Due to this complex architecture, the availability of these services may be **low**, and their **response time high** [24].

10. *Tiny Service* is a small service with **few methods**, which only implements part of an abstraction. Such service often requires **several coupled** services to be used together, resulting in higher development complexity and **reduced usability**. In the extreme case, a Tiny Service will be limited to **one method**, resulting in many services that implement an overall set of requirements [20].

The antipatterns described above are quite complex, especially for the end user. To simplify this task, specifications (metrics) were introduced [25], through which these anti patterns will be described. The metric suite encompasses both static and dynamic metrics. The static metric suite includes (but is not limited to) the following metrics: number of methods declared (NMD), number of incoming references (NIR), number of outgoing references (NOR), coupling (CPL), cohesion (COH), average number of parameters in methods (ANP), average number of primitive type parameters (ANPT), average number of accessor methods (ANAM), and average number of identical methods (ANIM). The dynamic metric suite contains: number of method invocations (RMI), number of transitive methods invoked (NTMI), response time (RT), and availability (A). As a novelty for the current work, new indicators were added: the number of encapsulated services (NS), the total number of parameters (NP), the number of interfaces (NI) and the number of auxiliary methods (NUM), the payload volume (PLD).

In this section, the basic antipatterns and their specifications have been identified, which will be used to find vulnerabilities in the system based on logs. The next section will be devoted to the rules card, namely to its definition, to its composition.

3.3 Rule Cards

3.3.1 Grammar

The rule card is the output of the DSL Interpreter, which converted and expanded the rule card described using the DSL into JSON. In developing the principle of drawing up such a card, it was important to take this point into account, because it should be easy to compose, understand, and read, not only by the author of the card, but also by any other third-party person. Based on existing work [9], where the rules were described using a Backus-Naur Form, for development and support, which requires a lot of resources and effort, it was decided to simplify the task and define the card using JSON. The advantages of using JSON were defined earlier in the Tool and Technologies section.

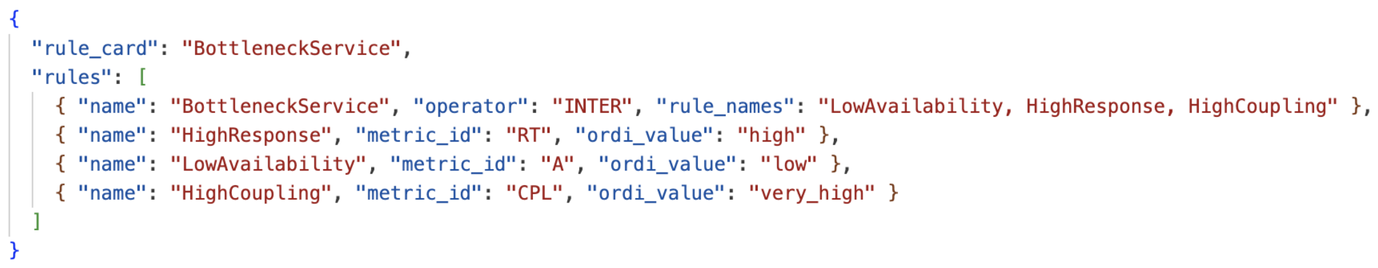
The Fig.5 shows the structure of the description of the rules card. The syntax of the rules card is quite easy to understand. It consists of 2 parameters: the name of the *rule_ card* card and the *rules* array itself. Each rule consists of a *name* and one *specification*. The specification can be a metric described in the *Antipatterns* section. A relationship or an operator can also serve as a specification. If we consider the specification of the metric in more detail, we can see that it comes in 2 types: with an indication of *ordi_ value* and with an indication of comparator *num_ value*. *ordi_ value* and *num_ value* show how much the metric has a strong infusion on the process. That is, for example, if we have the RT high specification, it means that the response time of the service is high, or if we have the PLD equal 0 specification, it means that the useful load transmitted by the service is zero.

```
rule_card      = rule card name
rules          = array of rules [rule]
rule           = name metric | relationship | operator
operator       = INTER | UNION | DIFF | INCL | NEG
metric         = id_metric ordi_value | id_metric comparator num_value
id_metric      = NMI | NRI | NOR | CPL | COH | ANP | ANPT | RT | A | PLD
ordi_value     = very_high | high | medium | low | very_low
comparator     = equal | less | less_equal | greater | greater_equal
relationship   = relationType FROM ruleName cardinality TO ruleName cardinality
relationType   = assoc | compos
cardinality    = one | many | one_or_many | num_value NUMBER_OR_MANY
rule_card, ruleName is String
num_value is Double
```

Figure 5: Grammar of Rule Cards

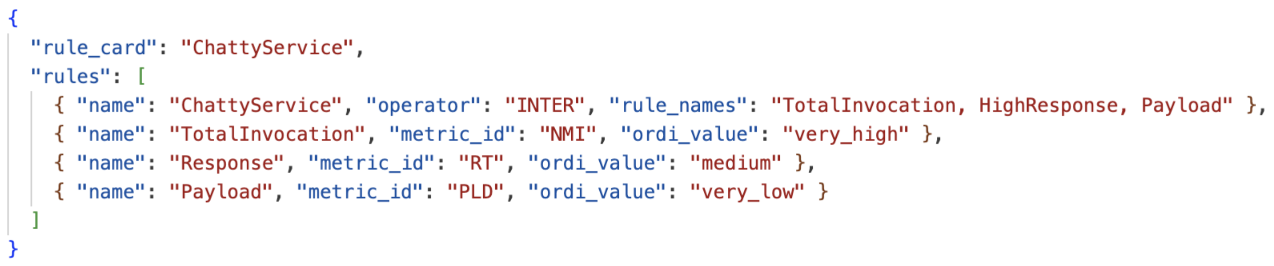
The following Figures 6, 7, 8, 9, 10 will provide examples of anti-pattern descriptions using the rules card.

3.3.2 Examples



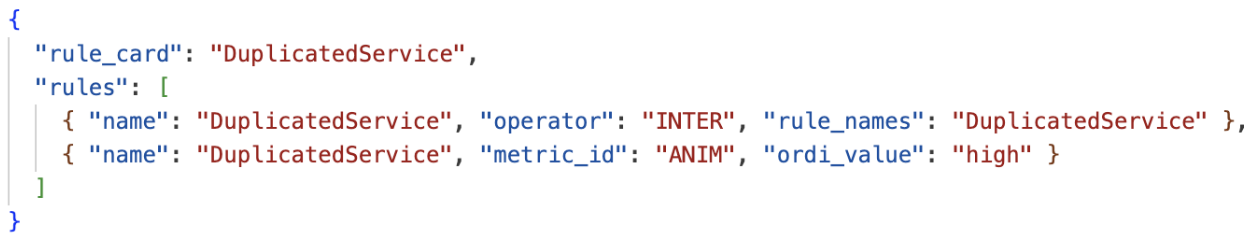
```
{
  "rule_card": "BottleneckService",
  "rules": [
    { "name": "BottleneckService", "operator": "INTER", "rule_names": "LowAvailability, HighResponse, HighCoupling" },
    { "name": "HighResponse", "metric_id": "RT", "ordi_value": "high" },
    { "name": "LowAvailability", "metric_id": "A", "ordi_value": "low" },
    { "name": "HighCoupling", "metric_id": "CPL", "ordi_value": "very_high" }
  ]
}
```

Figure 6: "Bottleneck Service" Rule Card



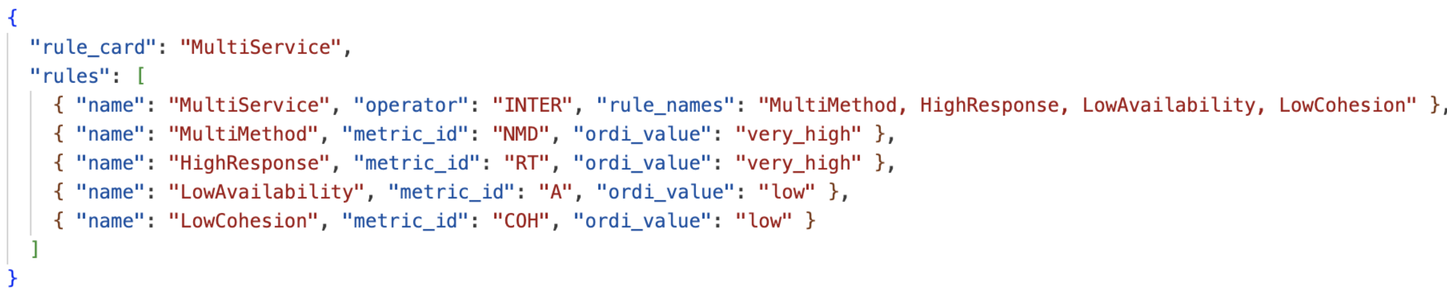
```
{
  "rule_card": "ChattyService",
  "rules": [
    { "name": "ChattyService", "operator": "INTER", "rule_names": "TotalInvocation, HighResponse, Payload" },
    { "name": "TotalInvocation", "metric_id": "NMI", "ordi_value": "very_high" },
    { "name": "Response", "metric_id": "RT", "ordi_value": "medium" },
    { "name": "Payload", "metric_id": "PLD", "ordi_value": "very_low" }
  ]
}
```

Figure 7: "Chatty Service" Rule Card



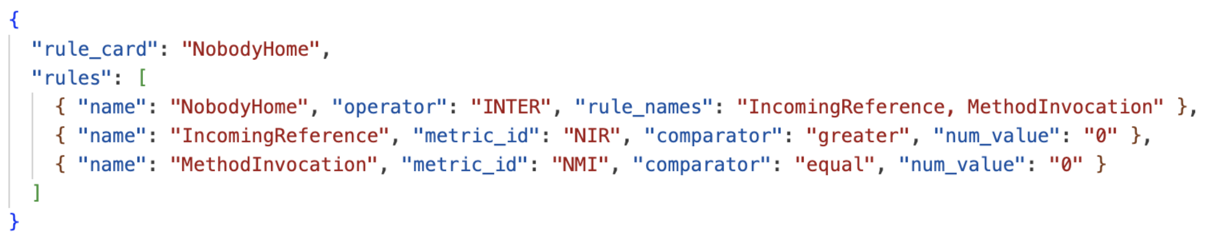
```
{
  "rule_card": "DuplicatedService",
  "rules": [
    { "name": "DuplicatedService", "operator": "INTER", "rule_names": "DuplicatedService" },
    { "name": "DuplicatedService", "metric_id": "ANIM", "ordi_value": "high" }
  ]
}
```

Figure 8: "Duplicate Service" Rule Card



```
{
  "rule_card": "MultiService",
  "rules": [
    { "name": "MultiService", "operator": "INTER", "rule_names": "MultiMethod, HighResponse, LowAvailability, LowCohesion" },
    { "name": "MultiMethod", "metric_id": "NMD", "ordi_value": "very_high" },
    { "name": "HighResponse", "metric_id": "RT", "ordi_value": "very_high" },
    { "name": "LowAvailability", "metric_id": "A", "ordi_value": "low" },
    { "name": "LowCohesion", "metric_id": "COH", "ordi_value": "low" }
  ]
}
```

Figure 9: "Multi Service" Rule Card



```
{
  "rule_card": "NobodyHome",
  "rules": [
    { "name": "NobodyHome", "operator": "INTER", "rule_names": "IncomingReference, MethodInvocation" },
    { "name": "IncomingReference", "metric_id": "NIR", "comparator": "greater", "num_value": "0" },
    { "name": "MethodInvocation", "metric_id": "NMI", "comparator": "equal", "num_value": "0" }
  ]
}
```

Figure 10: "Nobody Home Service" Rule Card

Let's look at the rule card for the *Bottleneck* antipattern that shown on Fig.6. The array of rules consists of 4 elements. The first rule is the *operator* specification. The example indicates that filtering will be done using the intersection of the listed rules *LowAvailability, HighResponse, HighCoupling*. The remaining 3 elements of the array are defined using the *mertic_id* specification. In this example, it is indicated that the Bottleneck antipattern is characterized by high response time, low availability and very high coupling of internal services.

As another example, consider the *Nobody Home* antipattern shown in Fig.10. This antipattern is described using 3 rules: the main 2 rules are described using the *metric_id* and comparator specifications. In this example, we have that it is typical for this antipattern that the *IncomingReference* number will be greater than zero and the *MethodInvocation* number will be zero. The third rule shows us that it is necessary to *intersect* the results of these 2 rules in order to get the final result.

In this section, the grammar of the rules cards was described, its structure and several examples of such cards were analyzed. By following this grammar, you can define a rule card for many other anti-patterns. The next section will be devoted to the implementation of the finished product. All its constituent components will be considered.

3.4 Implementation Structure

3.4.1 Overview

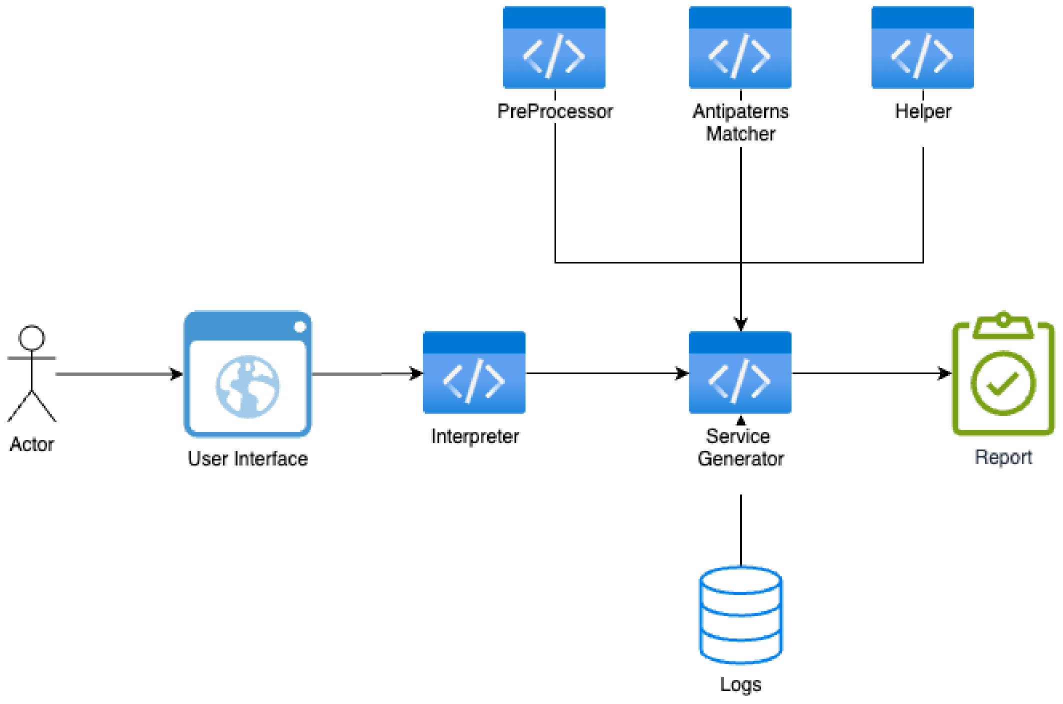


Figure 11: Implementation Structure

Fig.11 shows the implementation structure of the developed antipattern detection method. To make it convenient for the user to use this tool, a User Interface was implemented in which the user can select predefined rule cards with antipatterns. In more detail, what other components the system consists of is described below:

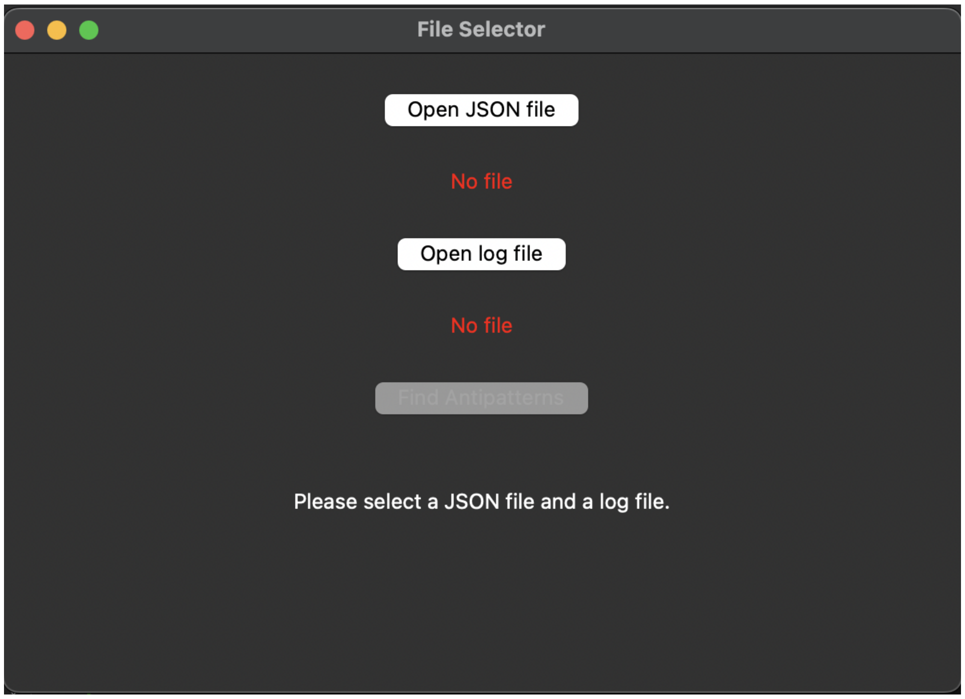
- **Actor:** This is a user initiating the process by uploading a rule card and event logs of an information system to trigger events that require analysis.
- **User Interface (UI):** This component serves as an interface through which the performer interacts with the system. This is where the user uploads the JSON file with the rules card and the IS logs. The selected files are then sent to the interpreter for further analysis.

- **Interpreter:** This module processes input data from the user interface, interpreting commands and data. It acts as an intermediary between the user interface and subsequent processing steps, ensuring proper formatting and forwarding of input data. Its main purpose is to *validate* the rules card. She checks that the card was written without errors, that all the specifications specified in the card exist. At the exit of the interpreter, we get a validated card that will be used in the main module of the *Service Generator*.
- **Helper:** This is one of the three auxiliary modules of the system. The main purpose of this module is to identify the antipattern according to the specifications verified by the interpreter.
- **Antipattern Matcher:** This module is responsible for *generating* an SQL query to search for an antipattern in the event database. The SQL query is formed from the results obtained from the Helper module. *Antipattern Matcher* does not interact directly with the *Helper* module, so as not to overload the module itself and making it easily changeable and integrated into other areas of the system.
- **Preprocessor:** The preprocessor module performs the initial processing of data, preparing it for more detailed analysis. This includes cleaning up the data, structuring it, or extracting the necessary elements.
- **Service Generator:** This module is the heart of the system being developed, as it implements the main method of exposing antipatters. This module imports 3 auxiliary modules: *Helper, AntipatternsMatcher, PrePocessor*. This module, after receiving a *validated* rule card using Helper and AntipatternsMatcher, has a ready-made SQL query database of events of a certain IS. Then, using this SQL, a query will be executed in the database. The results will be preprocessed using the PreProcessor module. The final results will be recorded in a report that will be available to the user in the format *.xlsx*.
- **Logs:** This database collects logs from various stages of the process of an information system. It stores data that will be used for further analysis, search and debugging.

- **Report:** The end result of the process is a report created based on data received from the Service Generator module. This report contains a summary table with detailed information about the detected antipatterns, broken down by the entered specifications. More information about this report will be described in the section 3.5.

3.4.2 UI

As described earlier, the UI module is the connecting link between the user and the system being developed. And it was important when designing this interface to make it as simple and understandable as possible for the user. The Fig.12 shows the main screen of the system. There are 3 buttons on it: A button for selecting a JSON file with a rule card, a button for selecting logs/databases, and a button for launching an anti-pattern detection method based on these input parameters. This button becomes available only when 2 input files are selected. There is also a file load indicator on the screen. It indicates whether the desired file has been selected. Fig.13 shows how the screen should look when the user has done everything right to start the system.

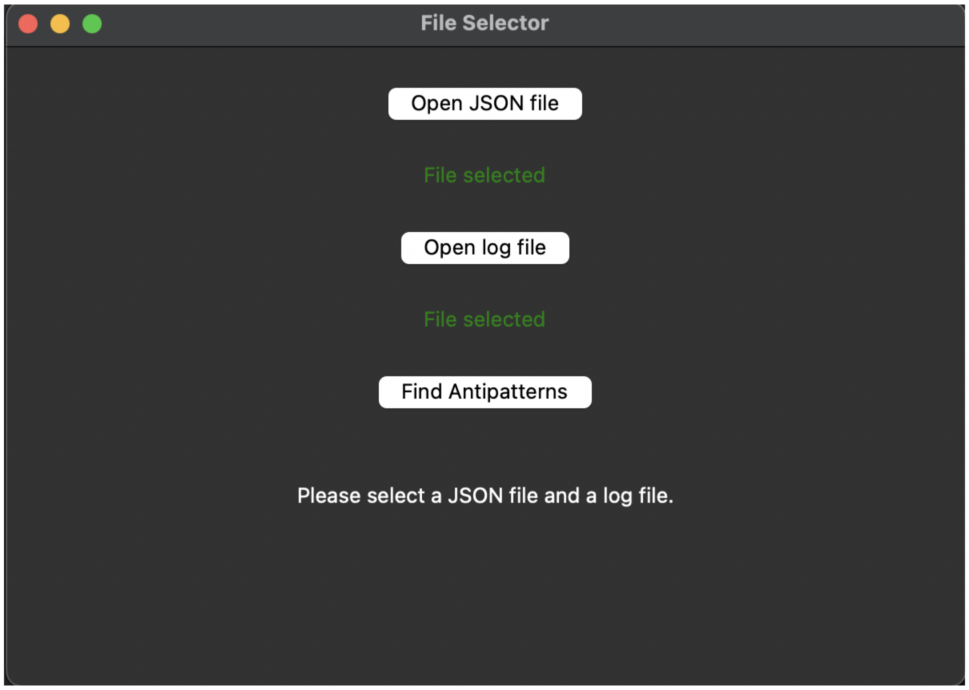


Please select a JSON file and a log file.

Figure 12: Initial state of UI

Fig.14 shows the notification that should appear when the system completes successfully.

After this notification appears, generated system report will automatically opened to user. Example of this report is shown on Fig.15. The generated report will be saved in the root folder with the source code in *.xlsx* format. This type of file was chosen for convenient and detailed viewing of all names of operations, services, modules for which the original business rule was executed. That is, the items on the list should be reviewed by IS engineers to revise their system architecture. Each time the system starts, a new file will be generated with the name specified in the rules card. If you do not change the name of the card, but only its content, then the old generated report will be lost. Subsection 3.5 describes in more detail how to analyze the results obtained in the report.



Please select a JSON file and a log file.

Figure 13: The state of the UI when everything is loaded correctly

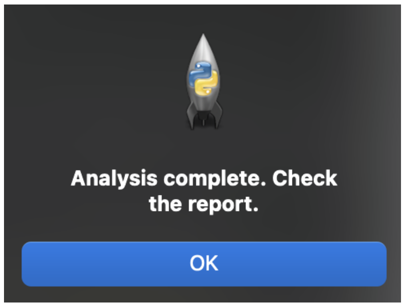
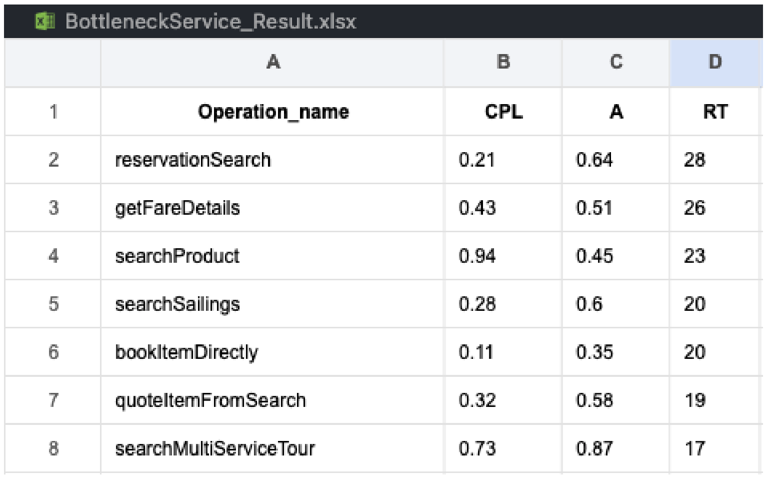

Analysis complete. Check the report.

Figure 14: Receiving notification when system finishes successfully


BottleneckService_Result.xlsx

	A	B	C	D
1	Operation_name	CPL	A	RT
2	reservationSearch	0.21	0.64	28
3	getFareDetails	0.43	0.51	26
4	searchProduct	0.94	0.45	23
5	searchSailings	0.28	0.6	20
6	bookItemDirectly	0.11	0.35	20
7	quoteItemFromSearch	0.32	0.58	19
8	searchMultiServiceTour	0.73	0.87	17

Figure 15: Generated report

3.4.3 Service Generator

This module is the main one, so let's look at it in more detail. To configure this parameter, we will need: a name, an operator, an array of rules and logs/database. After initialization, we run the **main()** method, which has the structure presented in the Listing 4. Listing 5 and 6 present methods for obtaining an antipattern based on rules and their specifications and a method for executing a query in the database.

Listing 4: Main method in Service Generator

```
def main(self):
  antipattern = self.detect_antipattern_based_on_rules()
  query_result = self.execute_query(antipattern)
  self.result_process(query_result, antipattern)
```

Listing 5: Method for detecting antipattern based on rules

```
def detect_antipattern_based_on_rules(self):
  antipattern = ''
  metrics_id = list(self.rules.keys())
  for item in Antipattern:
    if set(metrics_id) == set(item.value):
        antipattern = item.name
        break
        return antipattern
```

Listing 6: Method for executing antipattern based SQL query

```
    def execute_query(self, antipattern, logs):
        conn = sqlite3.connect(logs)
        cursor = conn.cursor()
        query = AntipatternsMatcher.sql_queries[antipattern]
        cursor.execute(query)
        results = cursor.fetchall()
        conn.close()
        return results
```

The next and final result will be devoted to the results obtained. We will look at the format in which the system issues the report, look at its contents, analyze a couple of examples and draw conclusions based on the results obtained.

3.5 Results Analysis

In this subsection, we analyze the results of the rule-based antipattern detection method applied to various services. We focus on the detection of Bottleneck Service, Chatty Service, and Duplicated Service antipatterns in our predefined logs of CRS IS. The analysis leverages key metrics such as coupling (CPL), availability (A), response time (RT), payload size (PLD), and number of method invocations (NMI).

3.5.1 Bottleneck Service Analysis

The Bottleneck Service antipattern is characterized by high coupling, low availability, and high response time. Table 1 presents the results for operations identified as potential bottlenecks.

The operation *searchProduct* exhibits the highest coupling (0.94) and lowest availability (0.45), indicating it as a critical bottleneck service. This high coupling suggests that *searchProduct* is heavily relied upon by other services, creating a potential single point of failure. The low availability indicates that this service may frequently be unavailable, leading to delays and potential system outages. Additionally, the high response time (RT) of 23 milliseconds for *searchProduct* further exacerbates the issue, as it indicates slow performance, likely due to the high demand and low availability. Improving the performance and availability of this service could significantly enhance the overall system efficiency.

Operation Name	**CPL**	**A**	**RT**
reservationSearch	0.21	0.64	28
getFareDetails	0.43	0.51	26
searchProduct	0.94	0.45	23
searchSailings	0.28	0.60	20
bookItemDirectly	0.11	0.35	20

Table 1: Bottleneck Service Detection Results

3.5.2 Chatty Service Analysis

The Chatty Service antipattern involves services with excessive communication characterized by high payload sizes and response times. Table 2 shows the results for detected chatty services.

Operation Name	**PLD**	**RT**
maintainInvoiceTrigger	184	3929
getBasicCancelInformation	626	1684
searchGeoZoneWithDetails	259	1502
manageAdviceTriggers	184	1486
getItemsCancelInformation	608	1235

Table 2: Chatty Service Detection Results

The operation *maintainInvoiceTrigger* has a significantly high response time (3929 ms) despite a relatively low payload size (184). This indicates that the service is involved in numerous interactions, causing delays and potential performance bottlenecks. Excessive communication between services, as seen with *getBasicCancelInformation* and *searchGeoZoneWithDetails*, also results in high response times (1684 ms and 1502 ms, respectively). These chatty interactions can lead to network congestion and increased latency, negatively impacting the overall system performance. Optimizing these services to reduce unnecessary communication and streamline interactions can improve response times and reduce latency.

3.5.3 Duplicated Service Analysis

The Duplicated Service antipattern is identified by services with redundant methods. Table 3 presents the results for operations with high average number of identical methods (ANIM).

Operation Name	**ANIM**
findReservationEntries	158
getAgeClass	64
getGeoZoneListByGeoZoneCodes	1224
getLocationHierarchyListByGeoZoneCodes	1193
getRailwayStationForRailAndFlyAirport	112

Table 3: Duplicated Service Detection Results

The operation *getGeoZoneListByGeoZoneCodes* has the highest ANIM (1224), indicating extensive duplication. This redundancy can lead to maintenance challenges, as updates and changes need to be made in multiple locations, increasing the risk of inconsistencies and errors. Furthermore, duplicated services consume additional resources, both in terms of memory and processing power, which can degrade overall system performance. Reducing redundancy by refactoring these services to eliminate duplicated methods can simplify maintenance, reduce resource consumption, and improve system reliability.

3.5.4 Evaluation Metrics: Precision and Recall

To evaluate the effectiveness of our anti-pattern detection method, metrics such as *precision* and *recall* are commonly used. Precision is defined as the ratio of correctly identified positive instances (true positives) to the total number of instances identified as positive (both true positives and false positives). It measures the accuracy of the detection method in identifying only relevant instances. Mathematically, precision is given by the formula 1:

$$\text{precision} = \frac{|\{\text{existing antipatterns}\} \cap \{\text{detected antipatterns}\}|}{|\{\text{detected antipatterns}\}|} \tag{1}$$

High precision indicates that the method produces a low number of false positives, meaning most of the detected instances are indeed correct.

Recall, also known as sensitivity, is the ratio of correctly identified positive instances (true positives) to the total number of actual positive instances (true positives and false negatives). It measures the completeness of the detection method in identifying all relevant instances. Mathematically, recall is given by the formula 2:

$$\text{recall} = \frac{|\{\text{existing antipatterns}\} \cap \{\text{detected antipatterns}\}|}{|\{\text{existing antipatterns}\}|} \tag{2}$$

High recall indicates that the method produces a low number of false negatives, meaning it successfully identifies most of the existing antipatterns.

For the event logs of the CRS information system analyzed in this work, we are unable to calculate precision and recall accurately because we do not have the exact number of existing anti-patterns. This limitation highlights the challenge of evaluating detection methods in real-world scenarios where the ground truth is not readily available. Despite this, the identification of potential anti-patterns using our rule-based approach provides valuable insights and a foundation for further investigation and refinement.

The analysis reveals several key anti-patterns within the event logs. Specifically, operations such as *searchProduct*, *maintainInvoiceTrigger*, and *getGeoZoneListByGeoZoneCodes* exhibit characteristics of bottleneck services, chatty services, and duplicated services respectively. These findings suggest areas where the system could be optimized to improve performance and reduce inefficiencies.

By analyzing precision and recall, we can further refine our detection rules and algorithms to enhance the accuracy and completeness of our anti-pattern detection method. This iterative process will help in achieving a balance between precision and recall, ensuring that our system is both accurate and comprehensive in identifying architectural issues.

Future work may involve refining the detection rules and expanding the analysis to include additional anti-patterns for a more comprehensive system optimization. By proactively identifying and addressing these anti-patterns, the system can enhance its operational efficiency and reliability, ultimately leading to better user experiences and reduced operational costs.

RESULTS AND CONCLUSION

Results

The rule-based anti-pattern detection method was applied to the event logs of the European Travel Computer Reservation System (CRS). The method's efficacy was evaluated based on the detection of several anti-patterns, including Bottleneck Service, Chatty Service, and Duplicated Service. Key metrics such as coupling (CPL), availability (A), response time (RT), payload size (PLD), and the number of method invocations (NMI) were utilized to analyze the results.

Bottleneck Service Analysis: This anti-pattern was identified by high coupling, low availability, and high response time. The searchProduct operation exhibited the highest coupling (0.94) and lowest availability (0.45), marking it as a critical bottleneck service. Its high response time (23 ms) further confirmed its status as a significant performance hindrance.

Chatty Service Analysis: Characterized by excessive communication, the maintainInvoiceTrigger operation showed a significantly high response time (3929 ms) despite a low payload size (184). This indicated frequent interactions causing delays, identifying it as a Chatty Service.

Duplicated Service Analysis: Identified by redundant methods, the getGeoZoneListByGeoZoneCodes operation had the highest average number of identical methods (1224), indicating extensive duplication. This redundancy can lead to maintenance challenges and increased resource consumption.

Evaluation Metrics: *Precision* and *Recall*: Due to the lack of exact numbers of existing anti-patterns, precision and recall were not calculated. However, the identification of potential anti-patterns using the rule-based approach provided valuable insights for further investigation and refinement.

The detection of these anti-patterns highlights critical areas for system optimization, aiming to improve performance and reduce inefficiencies. Future work will focus on refining detection rules and expanding analysis to include additional anti-patterns, enhancing overall system reliability.

Conclusion

The development of a rule-based method for detecting anti-patterns in event logs represents a significant advancement in process mining and system optimization. By leveraging predefined rules and domain-specific knowledge, this method offers a systematic approach to identifying deviations from desired process behaviors. The integration of techniques from process mining, machine learning, and domain-specific language design has demonstrated the method's effectiveness in detecting various anti-patterns across different domains.

This work's motivation, driven by challenges in the CRS information system, underscores the importance of proactive monitoring and detection mechanisms in ensuring system efficiency and reliability. Automating the detection of anti-patterns using predefined rules enables organizations to anticipate and mitigate potential issues, preventing system crashes and inefficiencies.

Future research will focus on improving the reliability, scalability and adaptability of the method in accordance with changing system requirements. The integration of advanced methods of analysis, anomaly detection and real-time monitoring will further expand the capabilities of the method in dynamic information systems. In order to improve the quality of recognition, it is proposed in the future to create a machine learning model capable of recognizing antipatterns. Then the two methods will be compared using the recall and precision metrics described above. It is quite possible that the proposed method in this paper will be much more effective than the method developed using machine learning. In addition, plans for the future include the development of an automated process for converting event logs from XML format to structured databases, which will further simplify data processing and analysis.

In general, this work contributes to the improvement of methods for eliminating patterns in intelligent process analysis and is of practical importance for organizations seeking to optimize operational processes and improve system reliability in complex digital environments.

REFERENCES

[1] Onsiter. *What are anti-patterns in software development and how to avoid them [Online] Available:* URL: https://onsiter.medium.com/what-are-anti-patterns-in-software-development-and-how-to-avoid-them-3973bb53224d#. (8.01.2024).

[2] Sematext. *Event Log [Online] Available:* URL: https://sematext.com/glossary/event-log/. (8.01.2024).

[3] engati. Simply intellegent. *Rule-Based System [Online] Available:* URL: https://www.engati.com/glossary/rule-based-system. (8.01.2024).

[4] JetBrains. *Domain Specific Language [Online] Available:* URL: https://www.jetbrains.com/mps/concepts/domain-specific-languages/. (10.01.2024).

[5] Jordan Hollander. *Central Reservation Systems [Online] Available:* URL: https://hoteltechreport.com/news/what-is-a-crs. (13.01.2024).

[6] Amazon. *Что такое сервис-ориентированная архитектура (SOA)? [Online] Available:* URL: https://aws.amazon.com/ru/what-is/service-oriented-architecture/. (14.01.2024).

[7] Amazon Web Service. *Что такое структурированный язык запросов (SQL)? [Online] Available:* URL: https://aws.amazon.com/ru/what-is/sql/. (15.01.2024).

[8] Nick Barney. *Service-oriented architecture (SOA)? [Online] Available:* URL: https://www.techtarget.com/searchapparchitecture/definition/service-oriented-architecture-SOA. (14.01.2024).

[9] Francis Palma. Mathieu Nayrolles. Naouel Moha. Yann-Gaël Guéhéneuc. *SOA Antipatterns: an Approach for their Specification and Detection [Online] Available:* URL: https://www.researchgate.net/publication/273933610. (29.01.2024).

[10] Thomas Erl. “Service-Oriented Architecture: Concepts, Technology, and Design.” In: (2005). (30.01.2024).

[11] Nuclino. *A Guide to Functional Requirements [Online] Available:* URL: https://www.nuclino.com/articles/functional-requirements. (29.01.2024).

[12] Indeed Editorial Team. *What Does an Interpreter Do? Duties, Skills and Tips [Online] Available:* URL: https://www.indeed.com/career-advice/finding-a-job/what-does-interpreter-do. (4.02.2024).

[13] Geek for Geeks. *What is Systems Design – Learn System Design [Online] Available:* URL: https://www.geeksforgeeks.org/what-is-system-design-learn-system-design/. (17.02.2024).

[14] IBM. *What are business rules? [Online] Available:* URL: https://www.ibm.com/topics/business-rules. (19.02.2024).

[15] Lentreo. *Types of System Architectures [Online] Available:* URL: https://medium.com/@abhijitgunjal1648/types-of-system-architectures-eb37dd496971. (22.02.2024).

[16] Python Software Foundation. *Python is a programming language that lets you work quickly and integrate systems more effectively [Online] Available:* URL: https://www.python.org/. (22.02.2024).

[17] Molechka. *Что такое XML [Online] Available:* URL: https://habr.com/ru/articles/524288/. (21.02.2024).

[18] Shershakov Sergey A. *Multi-Perspective Process Mining with Embedding Configurations into DB-based Event Logs [Online] Available:* URL: https://publications.hse.ru/chapters/314834252. (14.03.2024).

[19] SQLite Consortium. *What Is SQLite? [Online] Available:* URL: https://www.sqlite.org/. (21.02.2024).

[20] Bill Dudney and Stephen Asbury. “J2EE AntiPatterns.” In: (2003). (12.03.2024).

[21] KentBeck MartinJ.Fowler and JohnBrant. “Refactoring: Improving the Design of Existing Code”. In: (1999). (12.03.2024).

[22] Mamdouh Ibrahim Luba Cherbakov and Jenny Ang. *SOA Antipatterns: The Ob- stacles to the Adoption and Successful Realization of Service-Oriented Architecture. [Online] Available:* URL: https://www.ibm.com/developerworks/webservices/library/ws-antipatterns/. (12.03.2024).

[23] Steve Jones. *SOA Anti-patterns [Online] Available:* URL: https://www.infoq.com/articles/SOA-anti-patterns. (12.03.2024).

[24] Eric Bruno Arnon Rotem-Gal-Oz and Udi Dahan. “SOA Patterns”. In: (2012). (12.03.2024).

[25] Mathieu Nayrolles et al. *Soda: A Tool Support for the Detection of SOA Antipatterns [Online] Available:* URL: `https://www.researchgate.net/publication/236174040`. (14.03.2024).