EXTRACTING MICROSERVICE DEPENDENCIES USING LOG ANALYSIS

by

Andres Rodriguez Ishida

Graduate Program in Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Masters of Computer Science

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

Acknowlegements

First and foremost, I would like to express my deepest gratitude to my supervisor Dr. Kostas Kontogiannis, Professor, Department of Computer Science, Western University, London Ontario, for your invaluable guidance, constant support and endless generosity throughout the completion of this thesis. Looking back to my fourth year of undergrad, when I spoke to you for the first time about potentially pursuing a masters, I would have never imagined that I would have gained such an incredible and inspiring mentor. Not just a mentor but most importantly a roll model in my career that knows that the human element is just as important as knowledge. Your kindness is a testament to who you are as a person and I look up to you and forward to our next journey.

I would also like to thank Christ Brealey of IBM Toronto Lab for their invaluable comments and technical discussions. I would also like to thank Prof. Miriam Capretz, Prof. Hanan Lutfiyya, and Prof. Mostafa MIlani for serving as examiners for this thesis.

To my family, my deepest gratitude to my grandma and my sister. Thank you for the immense love and support you always show me. To my parents, your sacrifice and dedication is the reason I am right here thank you for always putting my needs and goals over your own. I'll strive to always be the person you taught me to be and I hope I can repay your efforts by living the life you worked so hard to give me and that allowed me to dream.

Para mi familia. A mi abuela y hermana mi más profundo agradecimiento, tan profundo como el amor que siempre me demuestran y el apoyo incondicional que me han dado. A mis padres, todo lo que soy y sere en el futuro es gracias al sacrificio y dedicación que le entregaron a sus hijos. Gracias por siempre poner mi vida y mis metas por encima de las suyas y aunque se que su amor es desinteresado espero tener el honor de devolverles aunque sea una fracción de sus esfuerzos viviendo la vida que soñaron para mi.

Abstract

Microservice architecture is an architectural style that supports the design and implementation of very scalable systems by distributing complex functionality to highly granular components. These highly granular components are referred to as microservices and can be dynamically deployed on Docker containers. These microservice architecture systems are very extensible since new microservices can be added or replaced as the system evolves. In such highly granular architectures, a major challenge that arises is how to quickly identify whether any changes in the system's structure violates any policies or design constraints. Examples of policies and design constraints include whether a microservice can call or pass data to another microservice, and whether data handled by one microservice can be stored in a specific database. In order to perform such type of analysis a model that denotes call and data dependencies between microservices must be constructed. In this thesis we present a technique that is based on log analysis and probabilistic reasoning to harvest, model, and associate logged events for compiling a labeled, typed, directed multigraph that represents call and data exchanges between microservices in a given deployment. We refer to this graph as *Microservices Dependency Graph* or MDG. The nodes of the graph denote microservices, service busses, publish-subscribe frameworks, and databases, while the edges denote data exchanges as well as send and receive requests. The graph contains a different edge for each type of interaction (i.e. data transfer, invocation, or response). We approach the problem of compiling a Microservice Dependency Graph in five major steps. The first step focuses on creating a metamodel for representing logged events and designing a metamodel for representing the MDG schema. The second step focuses on identifying associations and similarities between logged events. These associations create groups of events which may relate in the context of a transaction. The third step focuses on defining domain-specific log analysis logic based on a set of facts and weighted rules which encode complex relationships between events. These facts and rules constitute a knowledge base. The fourth step focuses on the application of a probabilistic reasoning engine to identify related events in the candidate groups of associated events, and impose an ordering relation between these events. Finally, the fifth step focuses on the compilation of the Microservice Dependency Graph. The prototype system has been applied on an open source microservice architecture system that simulates the operations of a garage shop. The identification of dependencies between microservices is a pivotal first step towards the implementation of various future frameworks. First, the MDG can be used to develop compliance analysis frameworks for microservice architectures. Compliance analysis may take the form of auditing a deployed system, to ensure that data are flowing from one component to another as specified (e.g. compliance with privacy policies), or data are not reaching or used by microservices not authorized to process these specific data (e.g. compliance with access control policies). Second, the MDG can be used to develop *what-if* analysis utilities. More specifically, as new features are added to the MSA system, an MDG can be compiled from test cases and the software engineers can identify what the interactions between the MSA components would be, if these features were finally released. This is an important aspect for minimizing the risk of unwanted interactions when the system is in production. Third, the MDG can be used to identify failure risk. A possible avenue of research would be to train a model to identify interaction patterns that are known to lead to failures. In this respect, when a new feature is added, the MDG interactions can be fed to the trained model and identify the failure risk proneness if this feature were to be released. This is an important utility for achieving continuous integration and continuous deployment (CI/CD).

Keywords: Microservices, log analysis, component dependencies, reasoning, formal concept analysis

Summary for Lay Audience

Microservice architecture is an architectural style that supports the design and implementation of very scalable systems by distributing complex functionality to highly granular components. These microservice architecture systems are very extensible since new microservices can be added or replaced as the system evolves. In such highly granular architectures, a major challenge that arises is how to quickly identify whether any changes in the system's structure violates any policies or design constraints. In order to perform such type of analysis a model that denotes call and data dependencies between microservices must be constructed. In this thesis, we present a technique that is based on log analysis and probabilistic reasoning to harvest, model, and associate logged events for compiling a labeled, typed, directed multigraph that represents call and data exchanges between microservices in a given deployment. We refer to this graph as Microservices Dependency Graph or MDG. The nodes of the graph denote microservices, service busses, publish-subscribe frameworks, and databases, while the edges denote data exchanges as well as send and receive requests. The identification of dependencies between microservices is a pivotal first step towards the implementation of various future frameworks such as compliance analysis frameworks, what-if analysis utilities so that software engineers can identify what the interactions between the MSA components would be if a new feature is added to the system, and frameworks to train a machine learning model to identify microservice dependency patterns that are known to lead to failures or pose a failure risk.

Contents

Acknowlegements **ii**

Abstract **iii**

Summary for Lay Audience **v**

List of Figures **xi**

List of Tables **xiii**

List of Appendices **xiv**

1 Introduction **1**

1.1 Preamble 1

1.2 Conceptual Outline of the Approach 3

1.3 Thesis Contributions 4

1.4 Thesis Organization 5

2 Background and Related Work **6**

2.1 Microservices 6

2.1.1 Introduction 6

2.1.2 Advantages 6

Replaceability and Strong Modularization 6

Continuous Delivery 7

Scalability 8

2.1.3 Challenges 8

2.2 Shift-Left 9

2.2.1 Introduction 9

2.2.2 Technical Debt 9

2.2.3 Benefits 10

2.2.4 Application 10

Test-Driven Development 10

Behaviour-Driven Development 11

DevSecOp 11

2.3 Compliance 11

2.3.1 Compliance and Continuous Compliance 11

2.3.2 FedRAMP 12

2.3.3 GDPR 13

2.4 Formal Concept Analysis 14

2.4.1 Reverse Engineering 14

2.4.2 Re-Engineering 16

2.5 Markov Logic Networks 17

2.5.1 General Representation 17

2.5.2 Risk Management 18

2.6 Microservice Dependency Graph Analysis 19

2.6.1 Service/Invocation chain logs 19

2.6.2 Graph Algorithms on Dependency Graphs 21

2.6.3 Source Code Analysis 22

2.6.4 Dyanmic Service Graph Generation 23

2.6.5 Version-Based Microservice Analysis 23

2.7 Research Gap 24

3 Process Outline and Architecture 25

3.1 General Outline 25

3.2 Data Extraction 28

3.3 Log Schema Reconciliation 30

3.3.1 Schema Reconciliation 30

Filtering Data 30

Cross-Referencing SQL Database Events with Microservice Names . . 31

Cross-Referencing SQL Database Events with Microservice Events . . 33

3.3.2 Conceptual Event Association 34

Attribute Synonym Synchronization 34

Event Association - Conceptual Method 35

Automating the Attribute Synonym Process 36

3.4 System-Wide Event Matching 38

3.5 Microservice Dependency Graph Extraction 39

3.5.1 Event Chains 39

3.5.2 Path Extraction and Graph Formation 39

4 Event Association and Schema Reconciliation 41

4.1 Filtering Data 41

4.2 Cross-Referencing SQL Database Events with Microservice Names 43

4.3 Cross-Referencing SQL Database Events with Microservice Events 45

4.4 Event Association 49

4.4.1 Attribute Synonym Synchronization 49

4.4.2 Event Association - Conceptual Method 53

4.4.3 Automating the Attribute Synonym Identification Process 55

FCA Table Creation 57

FCA Lattice Rule Extraction 58

4.5 Summary 63

5 System-Wide Event Matching 65

5.1 Final Matching of System-wide Events 65

5.1.1 Fact Base 65

Fact Extraction 66

Fact Examples 69

5.1.2 Rule Base 71

Rule Extraction 73

5.1.3 Reasoning . . . 76

Markov Logic and Markov Logic Networks . . . 76

Training and Inference . . . 78

Inference Result Analysis . . . 79

5.2 Summary . . . 86

6 Microservice Dependency Graph Extraction 87

6.1 MDG Domain Model . . . 87

Class Description . . . 87

Relationship Description . . . 88

6.2 Event Chain Formation . . . 89

6.3 Path Extraction - Sequences of Events . . . 91

6.4 Micro Service Dependency Graph Creation . . . 96

6.4.1 Microservice Dependency Graph Applications . . . 97

MDG Traversal . . . 97

RMI Guard Implementation . . . 99

Development Aid . . . 100

6.5 Summary . . . 101

7 Experiments and Discussion 103

7.1 Infrastructure Set up . . . 103

7.1.1 The Systems Microservice . . . 103

Web App . . . 103

Customer Management API . . . 104

Vehicle Management API . . . 104

Workshop Management API . . . 104

Message Broker . . . 105

Auditlog Service . . . 105

Workshop Management Event Handler . . . 105

Notification . . . 105

Invoice . . . 105

Time 106

Mail 106

SQL 106

7.1.2 Systems Event Types 106

Customer Registered Event 106

Vehicle Registered Event 107

Workshop Planning Created Event 107

Maintenance Job Planned Event 107

Maintenance Job Finished Event 107

Day Has Passed Event 107

7.2 Sample Run and Output 108

7.2.1 Data Collection 108

7.2.2 Rule Training 108

7.2.3 Fact Association 109

7.2.4 Microservice Dependency Graph Output 110

7.3 Threats to Validity 110

8 Conclusion and Future Work **113**

8.1 Conclusion 113

8.2 Future Work 114

Bibliography **116**

Curriculum Vitae **123**

List of Figures

2.1 A comparison between traditional Waterfall Model and Shift-Left approach [4] 9

2.2 FedRAMP Authorization Process [2] 13

3.1 The block diagram of the approach 26

3.2 The sequence of process steps 26

3.3 A sample of the centralized logs 27

3.4 A breakdown of four 'noisy' message broker events 27

3.5 An example of a microservice event log containing database command information 28

3.6 A sample of a SQL database event log 29

3.7 An example event structure 30

3.8 An example set of associated events, highlighting matching values 31

3.9 An example FCA table [38] 32

3.10 An example FCA lattice [38] 33

3.11 A subset of the Pitstop systems FCA lattice 34

4.1 Database event (highlighted) within the partial event data for one HTTP Request partition 43

4.2 An example log of a message broker's containing SQL data 44

4.3 An example of two events with attributes highlighting the SQL parameter data pattern matching 45

4.4 An example of a raw SQL log file 47

4.5 An SQL event stored in a dictionary structure 48

4.6 An example illustrating inconsistent labeling formats between logs 50

4.7 A dictionary data structure containing all attribute synonyms 51

4.8 The extracted FCA lattice for the log data set 63

5.1 An example Markov Logic Network [48], [10] 77

5.2 An example Markov Logic Network [48], [10] 77

5.3 Log Breakdown for the event pair InvoiceEvent-5 (Top) and NotificationEvent-8 (Bottom) 81

5.4 Log Breakdown for the event pair InvoiceEvent-5 (Top) and RabbitEvent-8 (Bottom) 83

5.5 Log Breakdown for the event pair InvoiceEvent-5 (Top) and InvoiceEvent-6 (Bottom) 85

6.1 MDG Domain Model 88

6.2 Message Broker exchange initialization examples 95

6.3 Sequence Diagram illustrating an RMI Guard implementation 100

7.1 The solution architecture of the MSA system 'PitStop' 104

7.2 The MDG for the PitStop Application 111

List of Tables

4.1 Associated attributes on event logs 62

List of Appendices

Chapter 1

Introduction

1.1 Preamble

Over the past decade we have seen significant breakthroughs in the way very large enterprise systems are architected. The first major breakthrough, dealt with the introduction of virtualization. In this architectural paradigm, a virtual model of a computer referred to as *virtual machine* could host a number of applications. These virtual machines along with their hosted applications could be replicated and deployed dynamically and on a as needed basis in order to handle varying computational loads and user demand. The virtual machines could run on top of a specialized platform, the *hypervisor*, which in its turn can be hosted on an actual computer which could handle the load of the hypervisor with its two or more (usually more) virtual machines. However, the next breakthrough came when the concept of microservice architectural style was introduced. Microservice architecture is based on two foundational concepts. The first concept is that functionality can be delivered by granular components called microservices. These microservices can interact with each other using standardized inter-process communication protocols such as sockers, RMI, XML-RPC, or service oriented messaging such as SOAP (Web Services) or http (restful services). These interactions can be coordinated so that complex business logic can be enacted and delivered to the stakeholders. The second concept is that all these microservices can be deployed and hosted in specialized components called *containers*. In their turn, containers can be deployed and replicated on top of *hypervisors* in one or more physical servers, and interact with each other also using standard inter-process communication protocols. This microservice architecture pattern provides a major improvement over the *virtual machines* architecture as it does not require a whole virtual computer (along with its Operating System and utilities) to be deployed to host an application, but rather highly granular components which can be individually deployed on-demand on containers, which can also be deployed on-demand to meet performance and QoS requirements as load and usage patterns change.

Most large-scale systems share a number of common characteristics that pose unique challenges. First, they entail complex logic and interactions between many different and diverse components. Second, they are implemented as distributed components which engage in concurrent transactions. Third, they handle high volumes of data, traffic, and users. Fourth, they must constantly evolve so they are kept operational.

As it becomes apparent, large-scale systems can benefit the most by the microservice architecture style. First, components can be deployed in different containers residing in different physical servers and locations thus achieving invocation and location transparency. Second, components can communicate in many different ways to implement complex business logic. Third, components can be provisioned on-demand in order to meet performance requirements as the load and traffic changes. It can therefore be argued that microservice architecture provides a very efficient design patter for many large-scale systems.

However, all these benefits come at a cost. The dynamic provision of containers and microservices creates highly complex interactions, which if not designed or monitored properly may easily lead to violations of policies, constraints or other functional and non-functional requirements. Examples of these violations include the provision of containers and microservices in foreign jurisdictions violating thus federal laws (e.g. privacy laws), the transfer of sensitive data to databases in which other non-secure systems may also have access to, and the invocation of microservices in contexts which violate access control security requirements.

In order to identify and mitigate these problems, a detailed model of the *as-is* (not the *as-designed*) infrastructure must be built first. This *as-is* system model denotes how components interact and can be created by analyzing either the source code of the system of the logs emitted by each component. In this thesis, we propose a technique that extracts the *as-is* system model using only logged data. The proposed approach is based on four main steps. In the first step, the structure (i.e. schema) of each logger is analyzed so that synonym attributes (if any) can be identified. In the second step, schemas are *reconciled* by applying Formal Concept Analysis [27]. In the third step, a collection of rules is designed, so that based on domain knowledge, log entries can be *associated*. In this step, entries in different logs form chains (or sequences) of related to a transaction, events. For this step, we propose the use of Markov Logic to define the rules, and Markov Logic Networks [48] for deducing whether two events *match* that is, they are related in the context of a transaction. In the fourth step, the chains of associated events are analyzed and individual paths are extracted. From the extracted paths a typed, labelled, directed, multigraph is created. We refer to this graph as the *Microservice Dependency Graph* (MDG). The nodes of this graph are microservices, middleware components, or database servers, and the edges denote *data exchanges* from one entity (i.e. microservice, middleware or data base server) to another. The compilation of the Microservice Dependency Graph is the first towards developing systems for enforcing run-time compliance, or performing off-line auditing, or evaluating *what-if* change scenarios during development, supporting thus continuous integration and continuous deployment (CI/CD).

1.2 Conceptual Outline of the Approach

The microservice dependency extraction framework presented in this thesis has five major conceptual components.

The first conceptual component relates to modeling and focuses on the design and implementation of two meta models. The first meta-model denotes logs schemas and is intended to represent events at a higher level of abstraction than just text entries in log files. The second meta-model denotes two types of dependencies between components in microservice architectures, namely call dependencies and data exchange dependencies. As discussed above, in the context of this thesis we consider three type of components *service components* (i.e *microservices*), *middleware components* (e.g. service busses, pub/sub servers), and *database components*.

The second conceptual component relates to knowledge representation, and more specifically with *i*) sets of highly associated attributes between event schemas as these are identified by the use of Formal Concept Analysis; *ii*) a collection of first order logic predicates that denote properties of the events (e.g. attribute/value pairs) and relations between events (e.g. time proximity association between two events) and; *iii*) a collection of extraction algorithms that analyze the log models to generate a collection of ground facts that conform to the set of the aforementioned predicates. These predicates form a *fact base*.

The third conceptual component deals with domain-specific log analysis logic and is based on a set of weighted rules which encode complex relationships between events. These rules form the *rule base* of the system.

The fourth conceptual component deals with reasoning and the identification of highly associated *event traces*, that collectively denote call and data transfer dependencies between various components (i.e. microservices, middleware components, and databases).

The fifth conceptual component deals with the analysis of event traces and the compilation of a Microservice Dependency Graph (MDG), which is a labelled, typed, directed multigraph.

The analysis process is discussed in detail in Section 3.

1.3 Thesis Contributions

The thesis focuses on the areas of system modeling and domain-based log analysis. More specifically, the thesis makes the following contributions.

- A technique to extract attribute-level associations between events in different system logs.

- A technique for denoting event-level dependencies using domain specific logic represented as a collection of weighed rules.

- A framework which utilizes probabilistic reasoning to extract sequences of related events across logs and across components forming thus complete transaction traces.

- A novel meta-model for denoting call and data dependencies between components in microservice architectures.

- The design and implementation of a tractable algorithm for analyzing transaction traces in order to populate the aforementioned dependencies metamodel forming thus a complete Microservices Dependency Graph (MDG).

1.4 Thesis Organization

The thesis is organized as follows. In Chapter 2 we present related work found in the literature. In Chapter 3 we outline the process of the proposed system, while in Chapter 4 we discuss conceptual event association and log reconciliation using Formal Concept Analysis. In Chapter 5 we present the reasoning framework including the modeling of facts and rules. In Chapter 6 we present the process of assembling the Microservice Dependency Graph, while in Chapter 7 we present findings by using the proposed systems. Finally, in Chapter 8 we conclude the paper and provide pointers for future research.

Chapter 2

Background and Related Work

2.1 Microservices

2.1.1 Introduction

Microservices architecture (MSA) is based on a share-nothing philosophy that structures a system as a set of loosely-coupled small isolated autonomous units. [28].

The purpose of MSA is to modularize by dividing large complex software system into smaller parts that can be deployed independent of each other. The independently deployed microservices are abel to communicate through the network, for example with REST.

The MSA can be defined by three features [62]. First, each service (unit) should only be designed for one task and accomplish this task effectively. Second, services should be able to work together. Third, a universal interface should be used. The MSA philosophy focuses on setting the service boundaries based on the business boundaries, thus making it clear where code resides in the system for any given functionality[45].

2.1.2 Advantages

Replaceability and Strong Modularization

In MSA separate teams develop software in modules and are only responsible for understanding the subsection of modules that correspond to their partition. In contrast to the traditional monolithic architecture where developers are required to understand an entire software package.[62].

The strong modularization also helps with the maintenance of the software. As the system's life cycle progresses and inevitable changes are required, developers are no longer required to comprehend the entire system instead they need only to understand the small subset of modules that correspond to their changes [62]. In large complex systems, the larger a system is the more costly a replacement can be. The risk of failure when replacing important business processes can have immense negative effects. This is where MSA provides a solution, since each microservice are small units that are independently deployed from each other the process of replacing a single microservice is not as costly [62].

In the case where new microservices are added to MSA, they are able to freely use any technology without any constraints. Since each microservice is deployed independently of other microservices, there are no restrictions or constraints in regards to the technologies used to implement each service [62].

Additionally microservices reduce the risks associated with replacement in the scenario where a microservice temporary fails. In which case the remaining microservices maintain their operability and are not effected by the failure or replacement of the faulty microservice [62].

Handling legacy applications is also simplified with MSA. Since a legacy application would only require an interface in order to communicate with the MSA, thus the challenge of code level integration for legacy systems is avoided [62].

Continuous Delivery

MSA provides exceptional aid to the continuous delivery approach. Considering the fact that microservices are small independent units, their deployment into the continuous delivery pipeline is efficient. This is due to the fact that the small microservice can be quickly tested, which results in rapid feedback which leads to faster deployment into production. Additionally given the smaller size of microservices, the risk of any deployment issues decreases. Since, even if the microservice fails, the effect on already deployed microservices are minimal. MSA is also beneficial for techniques such as Blue/Green deployment [62].

Scalability

Scalability is the property of a system's ability to handle increasing growth through the use of additional resources [7]. There are two types of traditional scaling, vertical and horizontal scaling [6]. Horizontal scaling is the more popular method, which involves the replication of microservices into other machines but comes at the downside of additional overhead being produced. Vertical scaling focuses on providing microservices additional resources in order to maximize the utilization. However this method comes with the limitation the resources available to a machine and the expensiveness of upgrading these resources [32].

2.1.3 Challenges

In the survey paper from Ghofrani and Lübke [28] three main concerns were documented. The first concern is with the development and debugging of MSA systems. Some examples include debugging a microservice that relies on other services, too many repositories to maintain or networking between dockers. The second concern was with skill and knowledge. Some examples include, finding developers and engineers that are knowledgeable with MSA and transitioning from traditional monoliths. The third concern was with correct separation of domains and finding the appropriate service cuts.

In the case of traditional monolithic architecture, scaling has to be done as a whole. In contrast to MSA, where smaller services can be scaled as needed without the requirement that all other microservices also be scaled. This results in a more cost efficient distribution [45].

Considering that microservices interactions occur through APIs that are exposed to the network, this creates the additional threat of potential attacks. In contrast to the traditional monolithic, where the systems interactions are all internal [19].

The strong modularity of the MSA results in complex network activity. This can result in increased difficulty for monitoring, security, and auditing of the entire system[19].

Another challenge with MSA is with the approach to the systems modularization. Determining the right size microserivce and the right design boundary are a few challenges that if not addressed properly can lead to increased network communication [29]. Additionally the previously mentioned individual teams responsible for individual microservices can both be

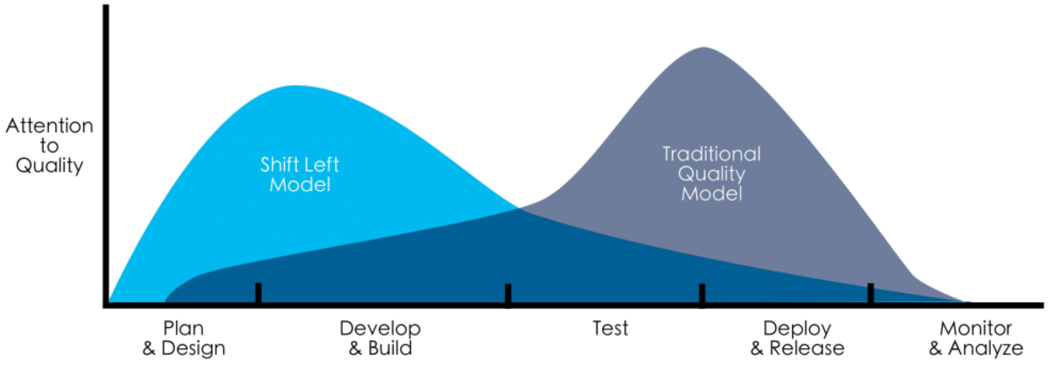


Figure 2.1: A comparison between traditional Waterfall Model and Shift-Left approach [4]

a benefit and a detriment. By focusing on the smaller picture teams may lose site of the big picture, for example are their local decisions coherent with the system's overall architecture and business goal [29].

2.2 Shift-Left

2.2.1 Introduction

The Shift-Left concept centers around the practice of performing more tests during the earlier stages of the software development life-cycle(see Figure 2.1). In contrast to the traditional waterfall model, in which testing is done towards the end of the development life cycle. In the Shift-Left approach, waiting until the system is produced in order to begin testing is no longer necessary. Instead all sorts of testing is executed earlier in the development life-cycle. Testing such as unit testing and integration testing are all completed at the beginning of the life-cycle and no longer towards the end.

2.2.2 Technical Debt

Technical debt is a concept introduced by Cunningham [13] in which he describes it as "shipping first time code is like going into debt. A low technical debt speeds development so long as it is paid back promptly with a rewrite". Brown et al. [8] defined technical debt as the "gap between the current state of a software system and some hypothesized 'ideal' state in which the system is optimally successful in a particular environment".

Attributes of technical debt includes monetary cost, bankruptcy, interest and principal, leverage, repayment and withdrawal [58]. Monetary cost are a result of technical debt, in which real financial consequences occur due to the inefficient utilization of developer's time. Bankruptcy refers to when the accumulated technical debt is overwhelming which results in termination of current progress and restarting is required. Interest and principal refers to the financial concept of interest payments, in which the time spent on faulty programming results in having to be paid back with interest via the time spent correcting those mistakes. Leverages refers to the trade-off between strategically sacrificing quality in exchange for shorter time-to-market. Repayment and withdrawal refers to the financial concept of credit cards. In which the credit rating of a team is determined by their ability to pay off technical debt.

2.2.3 Benefits

The main benefits of Shift-Left is the decrease in potential technical debt. The Shift-Left approach helps identify any potential issues early on in the life-cycle while changes to the design are not expensive. Additionally this allows for any potential issues to be resolved without the pressure of immediate deadlines.

2.2.4 Application

Test-Driven Development

Some of the benefits of TDD are the closing of the gap (shifting to the left) between design development and implementation feedback. Another focus of TDD is the culture of having developers write code that is automatically testable, which can result in the improvement of quality assurance. Additionally, in TDD there is continuous execution of automated test cases such that the identification of any error prone implementations can be captured efficiently. The focus of TDD is to shift from testing after implementation to testing before implementation [42].

Behaviour-Driven Development

Behaviour-Driven Development was developed by Dan North [14] with the focus on behaviour being defined with fine-grained specifications such that they can be automated [56]. Additionally the behaviour of the system is formatted in a ”Given-When-Then” structure using natural language sentences [14]. A benefit of BDD is the improvement in the communication between various project stakeholders. Another benefit is, since the software specifications are expressed in domain-specific terms, end users can easily understand them [5]. The Shift-Left philosophy is captured in BDD in the models Shift-Left of functional testing.

DevSecOp

DevSecOp is the unification of the development and operations team with the security team [51]. The focus of DevSecOp is to address security before the development stage, such that infrastructure security is addressed from the start. As well, DevSecOps focuses on automating security gates throughout the DevOps workflow while maintaining minial disruptions to operations. The benefit of DevSecOps is the added communication from the security teams, allows for sharing feedback and insight on known threats to developers. In this process the Shift-Left philosophy is interpreted as Shift-Left Security.

2.3 Compliance

2.3.1 Compliance and Continuous Compliance

Compliance in software development refers to how well a system obeys the established guidelines, policies, or specification. The industry standard for compliance certification is achieved through manual auditing of the system, however this method is error-prone, partial, and expensive [22]. Additionally security engineering techniques are normally conducted in a linear model, which is disadvantageous to the increasing development methods that are applying agile philosophies[43].

Continuous compliance is centered around the continuous verification of a systems regulatory compliance standards [22]. Aspects of continuous compliance can be seen in R-Scrum [23], in which compliance assurance is conducted at the end of each sprint. Filepp et al. [21] proposed a framework for continuous compliance in which they provide an automated solution for managing security compliance. Another approach to continuous compliance is by Moyon et al. [44], in which they focused on integrating the security standard requirements into the agile process model. Their proposed process model consists of three stages, Modeling, Validation, and Merging. Modeling focuses on representing security standards using the graphical modeling language known as Business Process Model and Notation (BPMN). Validation focuses on developers and engineers review the BPMN and Scaled Agile Framework models. Merging focuses on merging both models [44].

2.3.2 FedRAMP

The Federal Risk and Authorization Management Program (FedRAMP) is a program established by the United States federal government. FedRAMP provides a standardized approach for security assessment, authorization, and continuous monitoring for cloud technologies [2]. Some of the benefits include, reduced duplicative efforts, cost inefficiencies and inconsistencies. Additionally FedRAMP allows for the establishment of a public-private partnership that promotes innovation and advancement of secure information technologies [2]. FedRAMP provides two different processes for the authorization of a Cloud Service Offering, as shown in Figure 2.2.

The FedRAMP continuous monitoring program is based on NIST SP 800-137 [18]. The continuous monitoring process defined by NIST, includes the following practices: Define, Establish, Implement, Analyze and Report, Respond, Review and Update. Define, is the property of a continuous monitoring strategy that provides clear visibility into the awareness of vulnerabilities and assets with the usage of the latest threat information. Establish, refers to the establishing of metrics, status monitoring and control assessments that reflect security status,

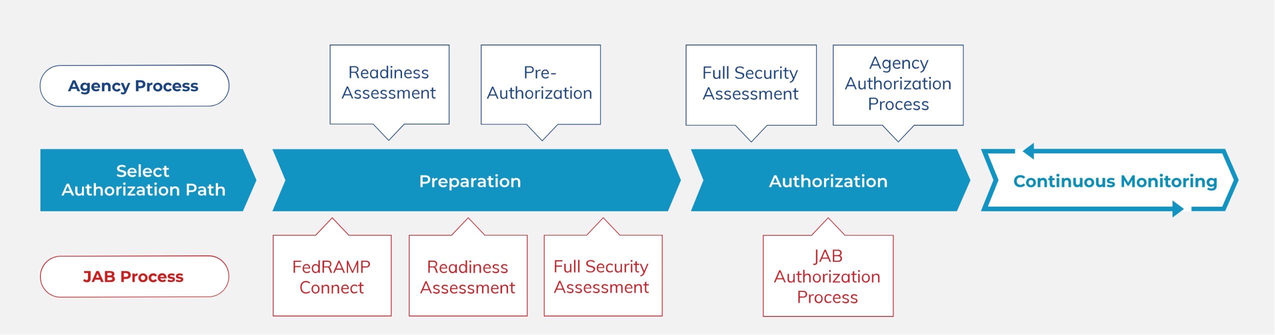


Figure 2.2: FedRAMP Authorization Process [2]

information infrastructure change detection, and operations environment. Implement, refers to the implementation of a continuous monitoring program that can be automated for collection, analysis and reporting. Analyze, refers to the reporting of the analyzed data paired with recommendations. Respond, refers to the decision making resulting from the report assessments. Review and Update, refers to the assessment and revision of the continuous monitoring strategy [2].

2.3.3 GDPR

The General Data Protection Regulation is a regulation that focuses on data protection and privacy in the European Union and European Economic Area or any organization that collects or processes information from EU citizens [34].

GDPR consists of seven main data processing principles [20].

- *”Lawfulness, fairness and transparency — Processing must be lawful, fair, and transparent to the data subject.”*
- *”Purpose limitation — You must process data for the legitimate purposes specified explicitly to the data subject when you collected it”*
- *”Data minimization — You should collect and process only as much data as absolutely necessary for the purposes specified.”*
- *”Accuracy — You must keep personal data accurate and up to date.”*

- *”Storage limitation — You may only store personally identifying data for as long as necessary for the specified purpose.”*
- *”Integrity and confidentiality — Processing must be done in such a way as to ensure appropriate security, integrity, and confidentiality (e.g. by using encryption).”*
- *”Accountability — The data controller is responsible for being able to demonstrate GDPR compliance with all of these principles.”*

2.4 Formal Concept Analysis

2.4.1 Reverse Engineering

Kumar and Kumar [31] applied Formal Concept Analysis to object oriented systems with the goal of identifying dependencies within the system. The proposed framework provides an efficient method for identifying the internal hierarchy structure between components during the reverse engineering process. The purpose of this framework is to reduce the cost of the maintenance process. The proposed framework consists of the following five steps, source code import, build FCA elements and properties, concept generation, generating high level views, and interpretation and analysis. In the first step, source code import, all the source code files and packages of the system are retrieved. In the second step, Build FCA Elements and Properties, the source code is parsed and converted into FCA objects and attributes. The type of dependency determines which objects and attributes are required from the incidence table. The third step, concept generation, is when the formal concepts are created. The algorithm used for generating the formal concepts is the Bottom up algorithm. The fourth step is generating high level views, in this step the formal concepts are visualized and the depicted connections between components aid the analyzer in comprehending the systems internal structure. In the final step, interpretation and analysis, the analyzer processes the internal structure from the previous step and determines if and where any refactoring is needed.

Cole et al.[12] proposed a framework called Conceptual Analysis of Software Structure (CASS) with the goal of aiding in the understanding of complex software systems. The generalisation and specialisation in the hierarchical structure of FCA allow for visualization that ranges from very general to very specific levels of abstraction. CASS takes a knowledge framework consisting of software artifacts, relationships between artifacts, rules for generating new relationships and applies formal concept analysis to gain insights about a software's structure. Exploration of the system's structure is achieved through graph based queries, that correspond to a specific portion of the code which are used to generate concept lattices. The resulting exploration techniques afforded by the concept lattices are call graphs, unfolding a package, package names, and combing aspects. The call graph is generated using the static call graph from source code and dynamic call graph from the actions during the systems runtime. This technique helps aid in the comprehension of the code's modularity and the dependencies between packages. The next technique, unfolding a package, depicts how a portion of the call graph can be further analyzed by unraveling a portion of the lattice. The third technique, package name, explores the systems structure based of the lattice created using package's names. The last technique, combining aspects, takes the static call graph and combines it with the package name lattice. This results in the outer layer depicting the package and class hierarchical structure, while the inner layer displays the sub-packages and how they are organized according to their path-names.

Tourwe and Mens [59] seek to advance the study of turning existing software systems into aspect-oriented systems. The proposed framework focuses on aspect mining through the analysis of a system using formal concept analysis to discover aspectual views. The contributions are two fold, the first is a specific configuration of the FCA algorithm for aspect mining. The second is the discovery of certain aspectual views derived from the specific configuration of the FCA algorithm. The aspect mining process consists of four steps. The first is generating elements and properties from the source code. The second is applying the FCA algorithm to the data obtained in the first step. The third step is filtering out unimportant concepts based on pre-defined heuristics. The fourth step is taking the resulting concepts and classifying them based on a given criteria. Lastly, the final concepts are visually presented to developers. The proposed framework is capable of retrieving a multitude of aspectual views, including programming idioms, design patterns and code duplication. The programming idioms that are able to be identified include the access methods and polymorphism. The design patterns that are identified are the visitor and abstract factory.

2.4.2 Re-Engineering

Kazato et al. [30] proposed a semi-automatic framework for extracting correspondence between features and program elements in a multi-layer system. This framework seeks to aid in the challenge of feature/concept location given the ability for a feature to composed throughout multiple layers. The FCA based feature location technique (FLT) consists of four steps, Preparing Scenarios, Extracting Execution Traces of Each Layer, Applying FCA, and Locating Feature Using Formal Concept. In the Preparing Scenarios step, analysts are tasked with creating two different scenarios for each feature. With one of the scenarios executing the given feature, while the other scenario does not. In the next step, Extracting Execution Traces of Each Layer, through dynamic analysis execution traces of each layer executing each scenario are examined. In the next step, Applying FCA, the formal contexts are obtained from the set of scenarios, the set of features, and their binary relation. Then the sets of formal concepts are retrieved from the FCA output, with each concept including a subset of scenario in its extent and subset of features in its intent. In the final step, Locating Feature using Formal Concept, analysts are now able to pinpoint a specific feature using the set of concepts. Analyst are then able to comprehend which program elements correspond to each feature. From there analyst can determine which surrounding concepts are relevant to their desired feature and investigate their dependencies prior to any changes made to the given feature.

Snelting and Tip [55] proposed a framework based on formal concept analysis for the detection and remediation of design problems in a class hierarchy. This framework intents to aid in the inability to predict a system's entropy from the stand point of how a class hierarchy will be used by an application or the resulting hierarchy extensions from maintenance. The framework first consists of a table depicting the relationships between types of variables and class members throughout the usage of the class hierarchy. Then the concept lattice is retrieved from the table by using Ganter's algorithm [26]. The resulting lattice aids developers in their understanding of the hierarchy of the entire system but also specific portions of the system. In terms of re-engineering several issues are recognized, the first are the data members that appear at the bottom of the lattice are not accessed elsewhere in the program. In contrast to variables which are not accessed by any members which are shown at the top of the lattice. As well any data members of a given base class which are not used by all of the subsequent classes of the base class are shown. Additional exploration of the lattice can result in quantification of cohesion and coupling through the usage of algebraic decomposition [35] [47].

2.5 Markov Logic Networks

2.5.1 General Representation

Domingos and Richard [52] proposed a technique that combines first-order logic and probabilistic graphical models. Their technique, Markov Logic Network (MLN), consists of a first-order knowledge base (KB) with weights attached to each formula (clauses). Unlike with first-order knowledge bases, in which it contains a set of hard constraints on the set of possible worlds, MLNs allow for the possibility of a world that violates one formula in the KB. If a world violates a formula in the KB, MLN simply considers this as less probable but doesn't have the hard constraint of making it impossible. In MLNs there is a corresponding weight to each formula which represents the strength or weakness of a constraint. These weighted formulas are the template for constructing the Markov networks. The weights are learnt using the Fortran implementation of L-BFGS [64] [9]. Inferences are done through Gibbs sampling using ten parallel Markov chains, with each initial state using MaxWalkSat. Clauses are learnt using the CLAUDIEN system [17]. MLNs have the ability to handle uncertainty, reduce brittleness, imperfect and contradictory knowledge, all while being able to incorporate a wide range of domain knowledge. The proposed technique is capable of collective classification, object identification, link-based clustering, social network modeling, and link prediction.

Niu et al. [46] seek to improve on the limitations of current MLN implementations, which result in MLN's inability to scale beyond small data sets. The proposed framework, TUFFY, leverages RDBMS to address the performance and scalability limitations of MLNs. The inference stage of MLN consist of two parts, the grounding phase and the search phase. In the current state-of-the-art MLN inference engine, Alchemy [1], the grounding phase is a top-down procedure. On the other hand, the proposed framework TUFFY implements a bottom-up grounding approach in order to take advantage of the RDBMS optimizer which results in faster execution time. To further improve on the inferencing function, TUFFY developed a novel hybrid architecture that focuses on the usage of local search procedures in main memory. The third contribution from TUFFY is a partitioning technique which allows the system to introduce parallelism.

2.5.2 Risk Management

Zawawy et al. [63] proposed a framework to aid in the diagnostic task of root cause analysis. The proposed framework is based on requirement goal models, which are used alongside Markov Logic Networks to develop a diagnostic knowledge repository. The motivation behind this framework is to develop a tool that aids human analysis in the overwhelming task of monitoring and evaluation of complex logging data for large complex systems. The proposed framework consists of three parts, building a knowledge base, observation generation and diagnosis. In the first part, the functional and non-functional system requirements are represented as a collection of goal models. In the second part, the goal models are used to generate the diagnostic rule knowledge base and the logging data is used to generate the ground atoms. In the third step, an MLN is constructed based on the knowledge base. The resulting goal model can be organized in a hierarchical structure which allows for greater comprehension and explainability of the diagnostic process.

Stülpnagel et al. [61] proposed a framework that centers around Markov Logic Networks to predict the expected availabilities of infrastructure services and components. The motivation behind the framework is the inefficient task of manual threat analysis, thus the proposed semi-automated approach. The proposed framework consist of two parts, a dependency graph and the Markov logic networks. The dependency graph contains all major infrastructure services and components. The data from the dependency graph is then used to generate the MLN, alongside the learnt weights for the measured availabilities. Afterwards through inference and the addition of new threats, the framework is able to predict the availabilities of the system's components that correspond to various threat conditions.

2.6 Microservice Dependency Graph Analysis

2.6.1 Service/Invocation chain logs

Y. Lan et al.[33] proposed a dependency model and dependency mining method based on call chain logs. Their research focused on extracting local dependencies and discontinuous dependencies. The proposed service dependency mining algorithm is composed of four steps. The first step is obtaining chain tracking log data from multiple data sources and undergo data classification and aggregation. The second step is counting all the local service calls, including the repeated ones. The third step is local service dependency mining using the count determined in step two. The final step is the generation of discontinuous service dependency candidate set. As a result, the proposed algorithm extracts the discontinuous service dependency extraction

The proposed dependency model by Y. Lan et al.[33] is based on service call chain logs unlike our proposed dependency model in which it is based on both individual microservice logs and SQL logs. The utility of Y. Lan et al,[33] framework is to provide assistance in the optimization and deployment of microservices, where as the value of our proposed framework is to provide data dependencies that allow for the identification of potential design and policy violations.

S. Mat et al [39] proposed GMAT (Graph-based Microservice Analysis and Testing), a tool capable of analyzing risky service invocation chains and trace the linkage between microservices for new versions of a system. Additionally the proposed GMAT is capable of automatically generating a *Service Dependency Graph (SDG)* which can be used to visualize and analyze the dependencies between microservices. Another feature of GMAT its ability to detect service invocation chains (SIC) and apply anomaly detection onto it. SIC are obtained by querying the SDG and filtering out redundant data, such has sub-paths. Once the SIC is obtained cyclic dependency is detected through the application of Tarjan's Strongly Connected Component algorithm [57].

GMAT obtains the service call information through the use of the Java Reflection mechanism or alternatively using the extended Swagger(OpenAPI). As a result GMAT is only applicable to systems that make use of either Java Reflection or Swagger to obtain the service invocation call data. Comparatively, our proposed framework does not require the system to include specific documentations or mechanism. Our proposed dependency model is created using the logs generated by the system. As a result our framework is more flexible and given the richer data obtain through the logs, the dependency model includes richer data dependencies between microservices.

Gaidels and Kirikova[24] focus on assessing the quality of the underlying architecture in microservice based system. Through the use of graph algorithms, such as centrality and community detection algorithms, they were able to aid in the detection of architectural anti-patterns, critical components and cycling dependency within the service dependency graph. The two classes of algorithms used for static graph analysis were centrality algorithms and community detection algorithms. The centrality algorithms are capable of identifying the most critical node(s), and provide additional information regarding credibility, accessibility, and bridges between groups. Some examples of centrality algorithms used include, Degree centrality, Harmonic centrality and Betweenness centrality. Community identification are used to help understand complex networks by identifying coherent substructure[11]

Similarily, I. U. P. Gamage and I. Perera[25] also focused on identifying anti-patterns in a microservice based system using graph algorithms on a service dependency graph. The proposed approach is divided into four steps. The first two steps are, retrieve architectural data of the system and create the models containing the service dependencies. The third step is to apply the graph algorithms to the system model. The fourth step is to visualize the system model and any relevant metrics. The proposed framework focused on identifying anti-patterns using specific graph algorithms, they are as follows: The Knot using Clustering coefficient, Nano Service using Degree centrality, Service Chain using a custom algorithm, Bottleneck service using Degree Centrality, and Cyclic Dependency using Strongly connected components.

The service dependency graph used by both Gaidels and Kirikova[24] and I. U. P. Gamage and I. Perera[25] was generated using established tracing data tools. The dependency graph used by previous research focuses on service invocation calls, in contrast to our dependency graph which contains dependencies regarding the specific flow of data throughout the system. In terms of assessing the quality of the architecture, our framework focuses on design and policy violations whereas they focus on identifying anti-patterns, critical components and cycling dependency.

JCallGraph developed by Liu et al.[37] is a tracing and analytic tool used to represent the invocation relationship between microservices in a microservice based system. The three main capabilities of the proposed framework are, invocation modeling within milliseconds, minimal over-head without impacting applications performance, zero-intrusion and application-agnostic. Tracing is achieved through the use of JSF, a microservice management platform, and the addition of minimal critical tracing points in the middlewares to record the request-response relationship between microservices. In order to maintain the low impact on the system, the framework only samples successful invocation but records all failed invocation occurrences since it can be used for root-cause analysis.

The invocation graph produced by JCallGraph[37] is created using the microservice management platform JSF to trace invocation relationships through tracing points. The proposed framework's use of JSF results in zero code intrusions to the system, similarly to our proposed framework since it is logged based approach. JCallGraph also produces additional information relating to invocation dependencies, it also contains information regarding their frequency from both the perspective of a callee and a caller in an invocation chain. In to our proposed framework, where frequency is not depicted but rather the data flow between microservices.

2.6.2 Graph Algorithms on Dependency Graphs

GSMART (Graph-based and Scenario-driven Microservice Analysis, Reuse, and Testing) developed by Ma et al.[40] is a tool to aid in development and operation of a microservice based system. The four main functions of GSMART are the following: Managing and visualizing dependency relationships between microservices, detecting cyclic dependency, selection of regression test cases, retrieval of existing microservices. The generation of the service dependency graphs is based on the collection of all service invocation links. Additionally this framework also implements a service invocation chain to further enrich the information in the service dependency graph. Detection of cyclic dependency is categorized into either weak or strong. Weak cyclic dependencies occur among multiple services but not among multiple endpoints, while strong dependencies occur among multiple endpoints.

The service dependency graph created by GSMART[40], is created using the service invocation links obtain from the Reflection mechanism. In contrast to our proposed framework, there is no need for additional implementations in order to generate the dependency graph. In terms of graph analysis GSMART is capable of detecting faults related to cyclic dependency, while our proposed framework's graph analysis focuses on policy and propagation violations.

2.6.3 Source Code Analysis

Pigazzini et al [49] explore the field of research in architectural debt by providing a tool to explore the detection of architecture anti-patterns. Specifically, their research focuses on the identification of cyclic dependencies, hard-coded endpoints, and shared persistence. Cyclic Dependency detection is accomplished through the use of a microservice based system's call graph and Arcan. The tool created generates a call graph by analyzing the source code files and docker/Spring configuration files. Hard-Coded Endpoints are detected by scanning the source code and using pattern matching to identify IPv4 addresses and ports. Shared Persistence detection is achieved through the database information in the configuration files.

The framework created by Pigazzini et al.[49] uses a combination of configuration files and source code analysis in order to assess the underlying architecture, in contrast with our research where analysis of the system is based of only the log files. Given that the framework provided by Pigazzini et al.[49] requires the usage of Spring framework and docker, the anti-pattern detection cannot be applied to all systems. Comparatively to our proposed framework, given its log based approach, it does not have any system requirements.

2.6.4 Dyanmic Service Graph Generation

MicroHECL developed by Liu et al.[36] is a high-efficient root cause localization for availability issues in microservice based systems. The framework ranks root causes candidates for potential anomaly propagation chains by analyzing the dynamically generated service call graph. MicroHECL is capable of detecting three types of anomalies, performance anomaly, reliability anomaly, and traffic anomaly. MicroHECL consists of three parts, Service call graph construction, Anomaly propagation chain analysis, and Candidate root cause ranking. First the service call graph is constructed during the detection of an availability issue, using the service calls and metrics retrieved from the system. Then the anomaly propagation chain analysis function will traverse through the service call graph and determines a set of possible services as candidate root causes. Lastly the candidate root cause ranking function will take the set of root causes based on the Pearson correlation coefficient[3].

The service call graph in MicroHECL[36] is created at run-time by the run-time monitor, whenever an availability issue occurs. Compared to our proposed framework, in which it does not require the system to have any additional implementations to generate the dependency graph. However, both frameworks allow for an creation of updated snapshots of the current state of the microservice system. MicroHECL detects different types of anomalies using machine learning and statistical methods. Similarly, our proposed framework makes use of Markov Logic Networks to detect dependencies between microservices. Once the dependency graph has been generated both frameworks implement some sort of propagation analysis. MicroHECL implements an anomaly propagation chain analysis, in which it identifies potential root cause services for the anomaly. Our proposed framework implements a policy violation propagation analysis, in which it detects if there have been any propagation violations past the original policy violation source.

2.6.5 Version-Based Microservice Analysis

VMAMV by S. Ma et al.[41] is a Version-based Microservice Analysis, Monitoring, and Visualization tool developed for automatic design problem detection for microservice based systems with multiple versions in design time. Additionally the framework also detects service anomalies at runtime. The framework is composed of several parts and they are as follows, Microservice code analyzer, service registry register, monitoring object bridge, dependency graph module, and monitor module. The microservice code analyzer is a service side library that provides Java annotation functionality for the labeling of communication relationships between microservices. The service registry register contains the basic information for registering and unregistering of services. The monitoring object bridge is the bridge between services and monitored objects. The dependency graph module is responsible for the creation of the dependency graph and the detection of any errors or potential problems within the dependency graph. The monitor module is consists of the generation and analysis of metrics, alongside the detection of anomalies.

2.7 Research Gap

In section 2.6, we have explored various research publications in the field of Microservice Dependency Graph Analysis. As discussed above, their analysis of a microservice based system is centered around the service call dependency graph. Although this can provide critical information about the system in terms of certain anti-patterns or anomalies, it is still limited in capabilities due to the simplicity of the service call dependency graph. In contrast to the proposed framework described in this thesis, in which the focal point of the framework is a more data rich dependency graph. The proposed dependency graph contains specific data dependencies between microservices. The additional data provided in the dependency graph allows for more extensive evaluations of the microservice based system. The additional analysis capabilities of the dependency graph provide developers and engineers with more information regarding their system, which can result in the more efficient development and deployment of a microservice based system. Additionally, the research approaches discussed above for the most part either require additional implementations to the microservice based system or that the microservice based system be built on top of specific libraries in order to accommodate for their analysis tools. Unlike the technique proposed in this thesis, which does not have any requirements for the microservice based system since it is based on the systems logs.

Chapter 3

Process Outline and Architecture

3.1 General Outline

In this Chapter we present the overall microservice dependency extraction process, we discuss the rationale and importance of each step, we provide a corresponding short example for each process step, and we discuss how these steps can be automated so that a usable framework can be built. The process aims to yield a typed, labelled, directed multigraph, we refer to as *Microservice Dependency Graph* (MDG), and is composed of four steps.

The block diagram of the approach is depicted in Figure 3.1, while the sequence of the process steps is depicted in the activity diagram in Figure 3.2.

The first step (see Figure 3.1), deals with developing drivers that parse the event logs emitted by each component logger M_1, M_2, M_w, and extracting the individual event entries L_1, L_2, $\ldots$ L_w. The second step is to reconcile the schemas of each log file. The log schema reconciliation process aims to identify attributes in the schemas SL_1, SL_2, $\ldots$ SL_w of loggers M_1, M_2, M_w, which refer to the same concept (i.e. they relate). This can be a manual process by knowing the schema structure of each logger, or can be automated by using schema mapping techniques. Schema mapping is a technique that has been investigated in the context of databases [50], [53]. In this thesis, we use a technique known as Formal Concept Analysis (FCA) [27]. The use of FCA is discussed later in this Chapter and in detail in Chapter 4.

The second step yields a set of *associations* between *schemas* and is referred to as the *log schema reconciliation* step.

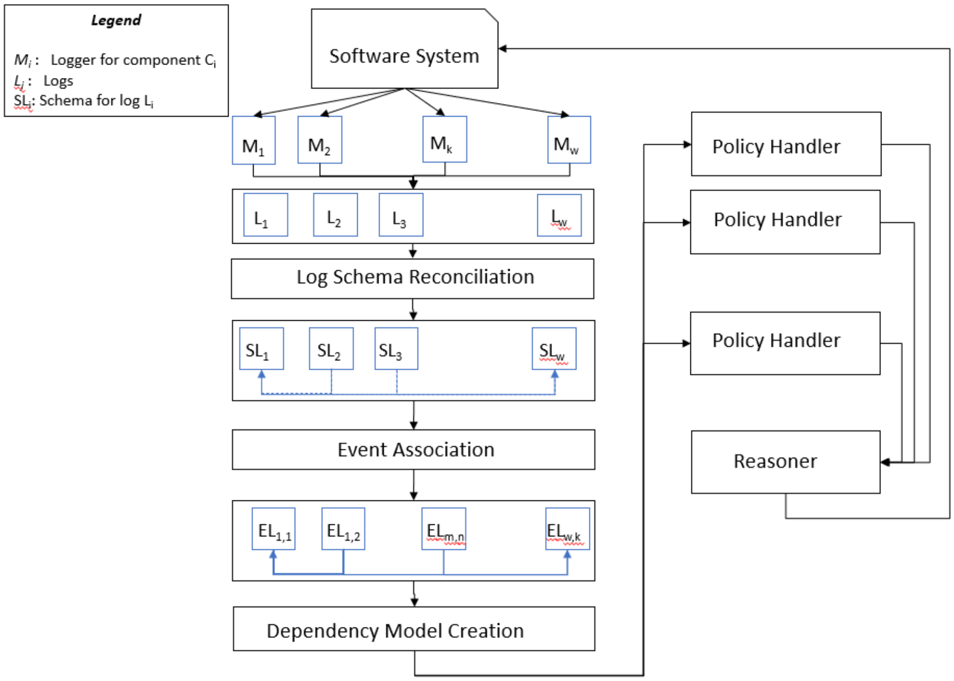


Figure 3.1: The block diagram of the approach

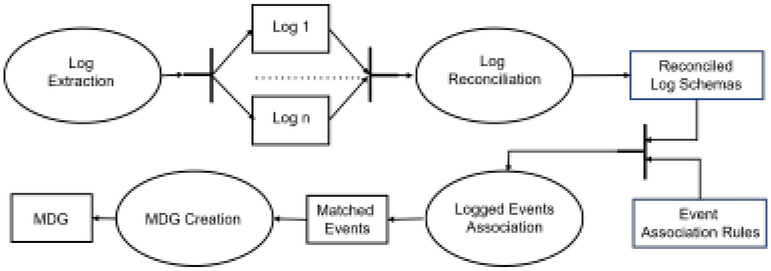
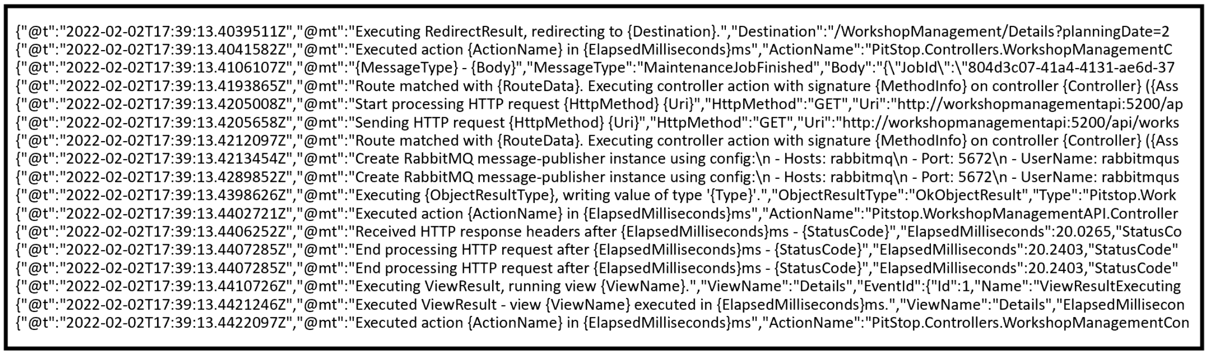


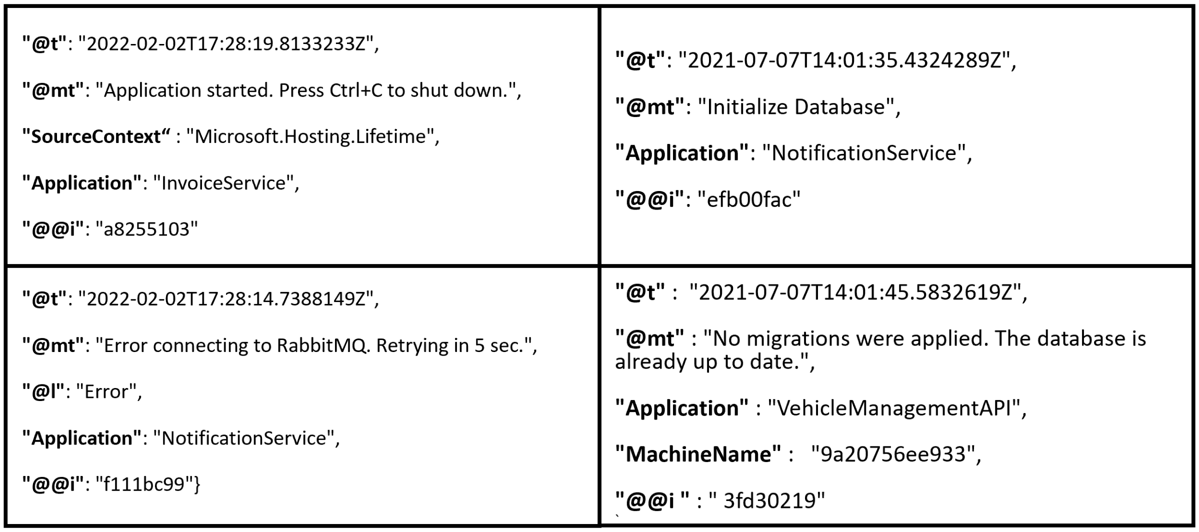
Figure 3.2: The sequence of process steps

The third step of the process is to use the logged events along with the *schema-level associations* in order to create *event-level associations*. As depicted in Figure 3.1, the event level associations are denoted as sets of sequences of events, where each sequence represents events emanating form different loggers, have associated attributes (see second step) and refer to the same transaction path. We refer to this step as the *System-Wide Event Matching* step. The event associations can be automated by applying event association domain logic encoded in the form of weighted rules as discussed later in this Chapter and in detail in Chapter 5. In the fourth step of the process the event sequences are analyzed, and a *Microservice Dependency Graph* (MDG) is compiled. The MDG is composed of nodes and edges. A node denotes a component



```
{"@t":"2022-02-02T17:39:13.4039511Z","@mt":"Executing RedirectResult, redirecting to {Destination}.","Destination":"/WorkshopManagement/Details?planningDate=2
{"@t":"2022-02-02T17:39:13.4041582Z","@mt":"Executed action {ActionName} in {ElapsedMilliseconds}ms","ActionName":"PitStop.Controllers.WorkshopManagementC
{"@t":"2022-02-02T17:39:13.4106107Z","@mt":"{MessageType} - {Body}","MessageType":"MaintenanceJobFinished","Body":"{\"JobId\":\"804d3c07-41a4-4131-ae6d-37
{"@t":"2022-02-02T17:39:13.4193865Z","@mt":"Route matched with {RouteData}. Executing controller action with signature {MethodInfo} on controller {Controller} ({Ass
{"@t":"2022-02-02T17:39:13.4205008Z","@mt":"Start processing HTTP request {HttpMethod} {Uri}","HttpMethod":"GET","Uri":"http://workshopmanagementapi:5200/ap
{"@t":"2022-02-02T17:39:13.4205658Z","@mt":"Sending HTTP request {HttpMethod} {Uri}","HttpMethod":"GET","Uri":"http://workshopmanagementapi:5200/api/works
{"@t":"2022-02-02T17:39:13.4212097Z","@mt":"Route matched with {RouteData}. Executing controller action with signature {MethodInfo} on controller {Controller} ({Ass
{"@t":"2022-02-02T17:39:13.4213454Z","@mt":"Create RabbitMQ message-publisher instance using config:\n - Hosts: rabbitmq\n - Port: 5672\n - UserName: rabbitmqus
{"@t":"2022-02-02T17:39:13.4289852Z","@mt":"Create RabbitMQ message-publisher instance using config:\n - Hosts: rabbitmq\n - Port: 5672\n - UserName: rabbitmqus
{"@t":"2022-02-02T17:39:13.4398626Z","@mt":"Executing {ObjectResultType}, writing value of type '{Type}'.","ObjectResultType":"OkObjectResult","Type":"Pitstop.Work
{"@t":"2022-02-02T17:39:13.4402721Z","@mt":"Executed action {ActionName} in {ElapsedMilliseconds}ms","ActionName":"Pitstop.WorkshopManagementAPI.Controller
{"@t":"2022-02-02T17:39:13.4406252Z","@mt":"Received HTTP response headers after {ElapsedMilliseconds}ms - {StatusCode}","ElapsedMilliseconds":20.0265,"StatusCo
{"@t":"2022-02-02T17:39:13.4407285Z","@mt":"End processing HTTP request after {ElapsedMilliseconds}ms - {StatusCode}","ElapsedMilliseconds":20.2403,"StatusCode"
{"@t":"2022-02-02T17:39:13.4407285Z","@mt":"End processing HTTP request after {ElapsedMilliseconds}ms - {StatusCode}","ElapsedMilliseconds":20.2403,"StatusCode"
{"@t":"2022-02-02T17:39:13.4410726Z","@mt":"Executing ViewResult, running view {ViewName}.","ViewName":"Details","EventId":{"Id":1,"Name":"ViewResultExecuting
{"@t":"2022-02-02T17:39:13.4421246Z","@mt":"Executed ViewResult - view {ViewName} executed in {ElapsedMilliseconds}ms.","ViewName":"Details","ElapsedMillisecon
{"@t":"2022-02-02T17:39:13.4422097Z","@mt":"Executed action {ActionName} in {ElapsedMilliseconds}ms","ActionName":"PitStop.Controllers.WorkshopManagementCon
```

Figure 3.3: A sample of the centralized logs

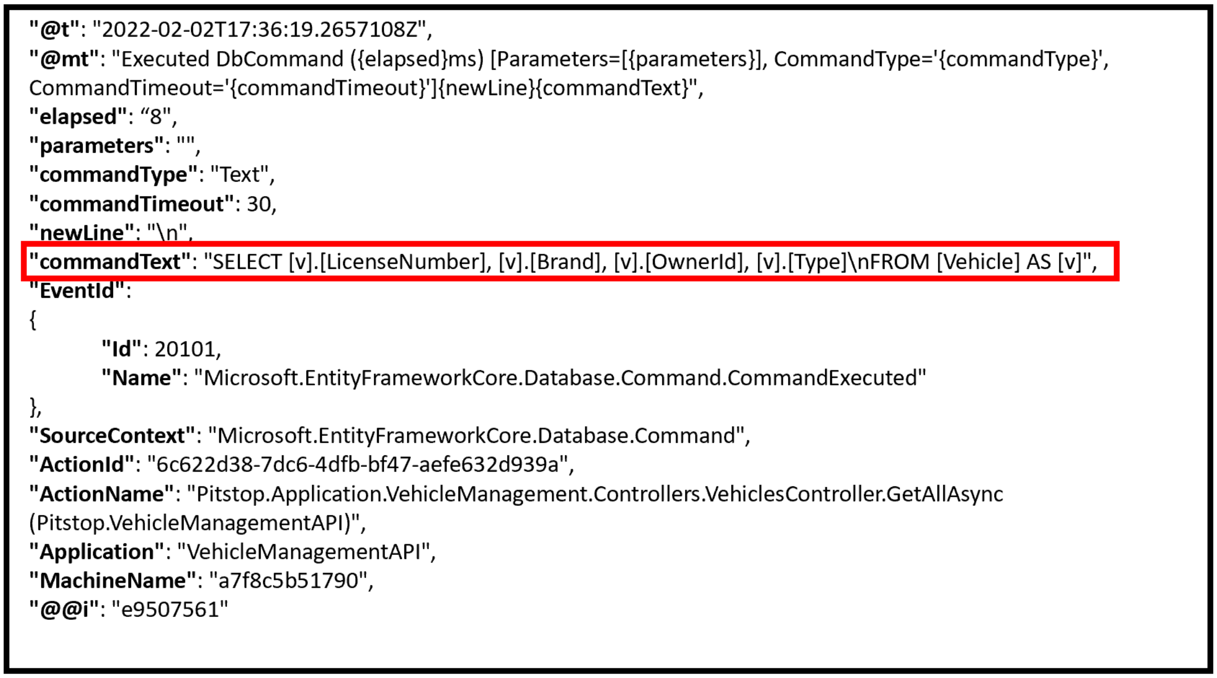


"@t": "2022-02-02T17:28:19.8133233Z", **"@mt"**: "Application started. Press Ctrl+C to shut down.", **"SourceContext“** : "Microsoft.Hosting.Lifetime", **"Application"**: "InvoiceService", **"@@i"**: "a8255103"	**"@t"**: "2021-07-07T14:01:35.4324289Z", **"@mt"**: "Initialize Database", **"Application"**: "NotificationService", **"@@i"**: "efb00fac"
"@t": "2022-02-02T17:28:14.7388149Z", **"@mt"**: "Error connecting to RabbitMQ. Retrying in 5 sec.", **"@l"**: "Error", **"Application"**: "NotificationService", **"@@i"**: "f111bc99"}	**"@t"** : "2021-07-07T14:01:45.5832619Z", **"@mt"** : "No migrations were applied. The database is already up to date.", **"Application"** : "VehicleManagementAPI", **"MachineName"** : "9a20756ee933", **"@@i "** : " 3fd30219"

Figure 3.4: A breakdown of four 'noisy' message broker events

in a microservice architecture (i.e. microservice, database, service bus, pub/sub framework), while an edge denotes a *data exchange* or a *call* (i.e. a request). Between two nodes there may exist more than one edge denoting different interactions. The domain model of the proposed *Model Dependency Graph* is described in Chapter 6 and is depicted in Figure 6.1. This fourth step concludes the *Microservice Dependency Graph* creation process.

As depicted in Figure 3.1, once a *Microservice Dependency Graph* is created, this can be fed to different *policy handlers*. A *policy handler* is associated with an edge of the MDG and denotes a specific policy requirement or constraint. Each policy handler evaluates a specific domain policy (e.g. a data transfer policy or constraint between two microservices) and produces a result which is fed to a *compliance reasoning* engine. Based on the results obtained



```
"@t": "2022-02-02T17:36:19.2657108Z",
"@mt": "Executed DbCommand ({elapsed}ms) [Parameters=[{parameters}], CommandType='{commandType}',
CommandTimeout='{commandTimeout}']{newLine}{commandText}",
"elapsed": “8",
"parameters": "",
"commandType": "Text",
"commandTimeout": 30,
"newLine": "\n",
"commandText": "SELECT [v].[LicenseNumber], [v].[Brand], [v].[OwnerId], [v].[Type]\nFROM [Vehicle] AS [v]",
"EventId":
{
        "Id": 20101,
        "Name": "Microsoft.EntityFrameworkCore.Database.Command.CommandExecuted"
},
"SourceContext": "Microsoft.EntityFrameworkCore.Database.Command",
"ActionId": "6c622d38-7dc6-4dfb-bf47-aefe632d939a",
"ActionName": "Pitstop.Application.VehicleManagement.Controllers.VehiclesController.GetAllAsync
(Pitstop.VehicleManagementAPI)",
"Application": "VehicleManagementAPI",
"MachineName": "a7f8c5b51790",
"@@i": "e9507561"
```

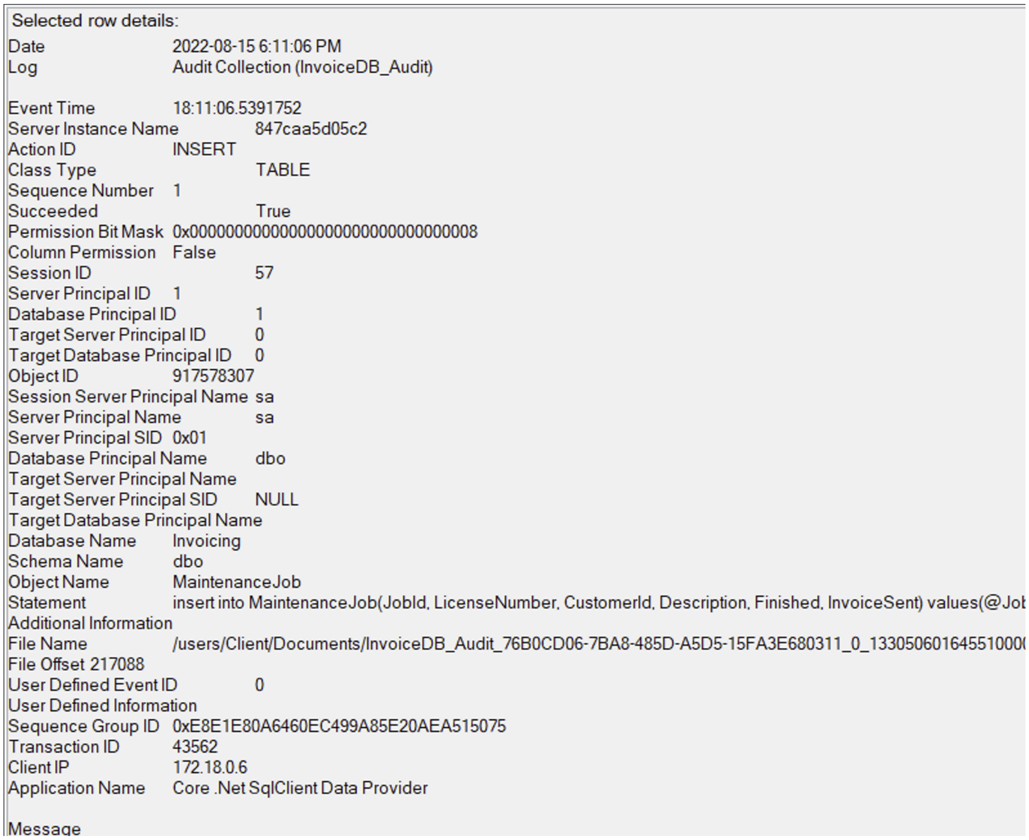
Figure 3.5: An example of a microservice event log containing database command information

by the individual policy handler, the compliance reasoning engine can then assess the overall compliance of the system. Even though the scope of the thesis is the extraction of the MDG, we discuss in Chapter 6 the potential uses of MDG and the overall architecture of the invocation of the different policy handlers.

In the following sections we outline each step, we discuss its rationale, and we present an illustrative example. The schema reconciliation, system-wide event matching, and MDG creation are discussed in detail in Chapters 4, 5, and 6.

3.2 Data Extraction

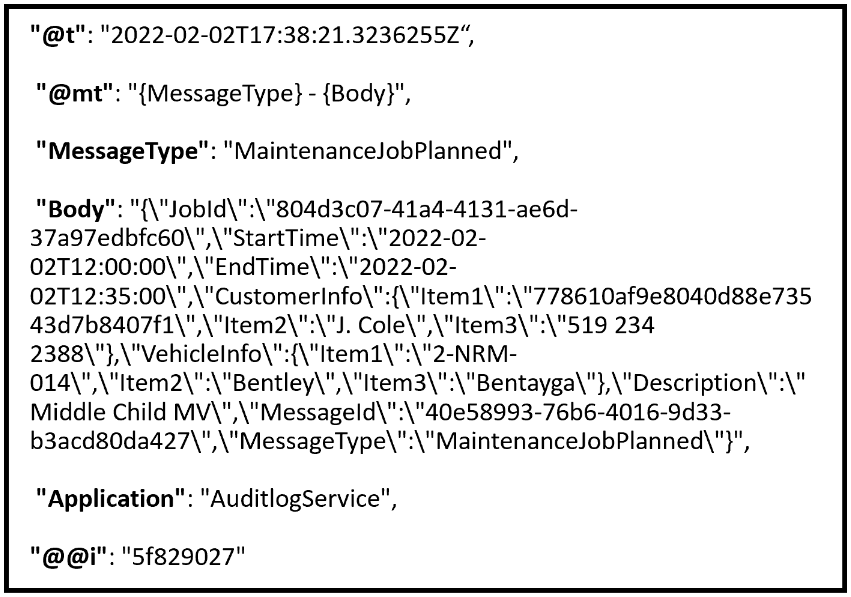
As depicted in the activity diagram in Figure 3.2, and discussed above, the first step is to harvest the logs from the different components (i.e. microservices, service busses, pub/sub infrastructure and data base servers). The log data collected for the open source microservice architecture system used to test the proposed framework, Pitstop [60], is collected from *Seq* [54]. *Seq* is a centralized logging server that the system implemented for the collection of all



```
Selected row details:
Date                    2022-08-15 6:11:06 PM
Log                     Audit Collection (InvoiceDB_Audit)

Event Time              18:11:06.5391752
Server Instance Name             847caa5d05c2
Action ID               INSERT
Class Type                       TABLE
Sequence Number         1
Succeeded                        True
Permission Bit Mask     0x00000000000000000000000000000008
Column Permission       False
Session ID                       57
Server Principal ID     1
Database Principal ID            1
Target Server Principal ID       0
Target Database Principal ID     0
Object ID               917578307
Session Server Principal Name sa
Server Principal Name            sa
Server Principal SID    0x01
Database Principal Name          dbo
Target Server Principal Name
Target Server Principal SID      NULL
Target Database Principal Name
Database Name           Invoicing
Schema Name             dbo
Object Name             MaintenanceJob
Statement               insert into MaintenanceJob(JobId, LicenseNumber, CustomerId, Description, Finished, InvoiceSent) values(@Job
Additional Information
File Name               /users/Client/Documents/InvoiceDB_Audit_76B0CD06-7BA8-485D-A5D5-15FA3E680311_0_133050601645510000
File Offset 217088
User Defined Event ID            0
User Defined Information
Sequence Group ID       0xE8E1E80A6460EC499A85E20AEA515075
Transaction ID          43562
Client IP               172.18.0.6
Application Name        Core .Net SqlClient Data Provider

Message
```

Figure 3.6: A sample of a SQL database event log

the logs produced throughout the *Pitstop* system. This centralized logging server supports the *Pitstop's* logging framework, *Serilog*. *Serilog* is utilized for its structured logging capabilities in complex, distributed and asynchronous systems [16]. Additionally the *Serilog* log formatting required the usage of *CLEF-Tool*(Compact Log Event Format Tool) [15] for converting the newline-delimited JSON data into standard JSON format. The SQL logs were obtained from the MS SQL Server. The process of retrieving the SQL logs required manual execution of the auditing process for each of the databases initialized in the SQL server. These were the steps taken for extracting the logs from the system, however the method in which the logs are obtained does not affect the concepts of the proposed technique. For example, lower level network traffic events can also be harvested without affecting the proposed technique.



```
"@t": "2022-02-02T17:38:21.3236255Z“,

 "@mt": "{MessageType} - {Body}",

 "MessageType": "MaintenanceJobPlanned",

 "Body": "{\"JobId\":\"804d3c07-41a4-4131-ae6d-
37a97edbfc60\",\"StartTime\":\"2022-02-
02T12:00:00\",\"EndTime\":\"2022-02-
02T12:35:00\",\"CustomerInfo\":{\"Item1\":\"778610af9e8040d88e735
43d7b8407f1\",\"Item2\":\"J. Cole\",\"Item3\":\"519 234
2388\"},\"VehicleInfo\":{\"Item1\":\"2-NRM-
014\",\"Item2\":\"Bentley\",\"Item3\":\"Bentayga\"},\"Description\":\"
Middle Child MV\",\"MessageId\":\"40e58993-76b6-4016-9d33-
b3acd80da427\",\"MessageType\":\"MaintenanceJobPlanned\"}",

 "Application": "AuditlogService",

"@@i": "5f829027"
```

Figure 3.7: An example event structure

3.3 Log Schema Reconciliation

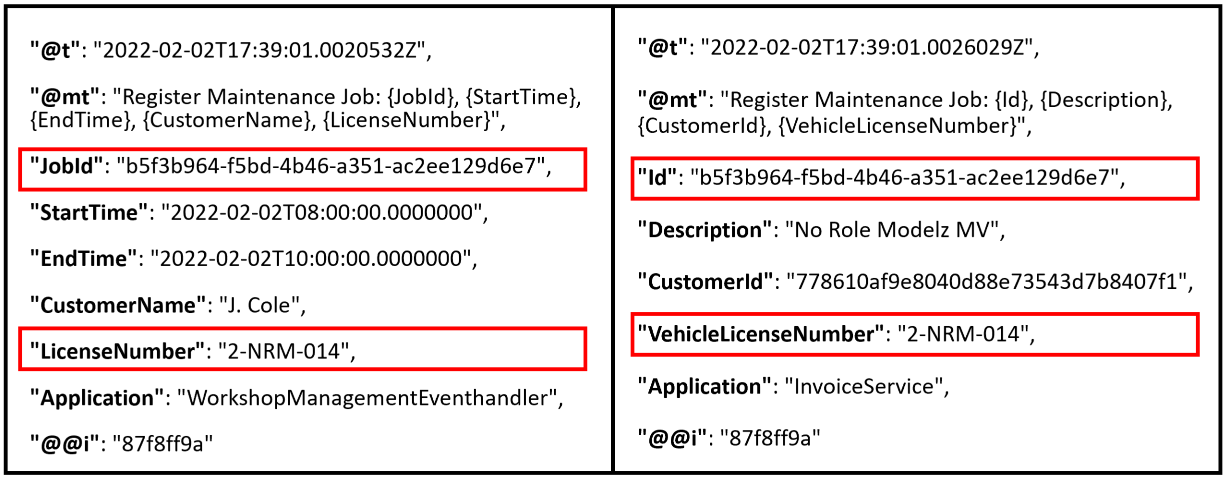
The second step of the process is divided into two parts, Log Schema Reconciliation and Conceptual Event Association.

Log Schema Reconciliation focuses on establishing relationships at the schema level, associations between Microservice events and SQL events. This part of the step is divided into three phases, *Filtering Data*, *Cross-Referencing SQL Database Events with Microservice Names*, and *Cross-Referencing SQL Database Events with Microservice Events*. In this section a brief overview is provided explaining each step, a thorough explanation of this process is provided in Chapter 4.

3.3.1 Schema Reconciliation

Filtering Data

The logs from the SQL database and the Microservice Architecture System (MSA) undergo a *filtering process*. As seen shown in Figure 3.3, the volume of data collected by a MSA system can be very high. This is a result of the logging frameworks implemented in the Microservice Architecture System collecting a large volume of information detailing the state of the system,



```
"@t": "2022-02-02T17:39:01.0020532Z",
"@mt": "Register Maintenance Job: {JobId}, {StartTime},
{EndTime}, {CustomerName}, {LicenseNumber}",
"JobId": "b5f3b964-f5bd-4b46-a351-ac2ee129d6e7",
"StartTime": "2022-02-02T08:00:00.0000000",
"EndTime": "2022-02-02T10:00:00.0000000",
"CustomerName": "J. Cole",
"LicenseNumber": "2-NRM-014",
"Application": "WorkshopManagementEventhandler",
"@@i": "87f8ff9a"
```

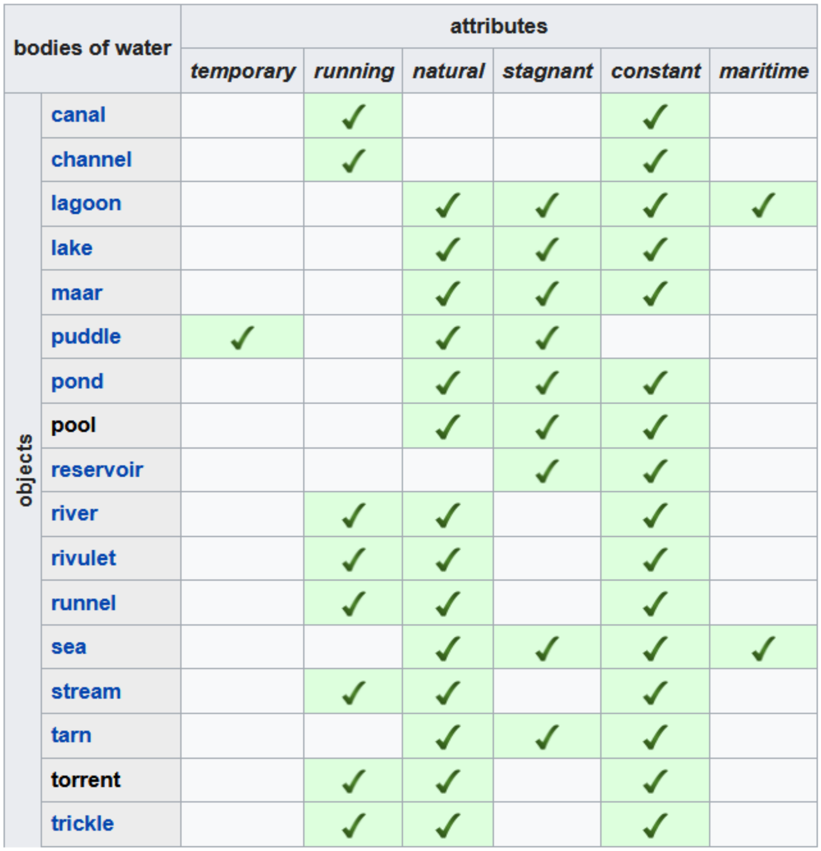
```
"@t": "2022-02-02T17:39:01.0026029Z",
"@mt": "Register Maintenance Job: {Id}, {Description},
{CustomerId}, {VehicleLicenseNumber}",
"Id": "b5f3b964-f5bd-4b46-a351-ac2ee129d6e7",
"Description": "No Role Modelz MV",
"CustomerId": "778610af9e8040d88e73543d7b8407f1",
"VehicleLicenseNumber": "2-NRM-014",
"Application": "InvoiceService",
"@@i": "87f8ff9a"
```

Figure 3.8: An example set of associated events, highlighting matching values

activities, communications between containers and much more, most of which may not be relevant in the search for data dependencies. Thus, a filtering process is required in order to reduce the amount of noise in the log data. An example of noisy data is shown in Figure 3.4, in which the sample events provide little to no information regarding dependencies in the system. Filtering is achieved through creating segments of events that correspond to HTTP requests. Through this filtering process we are able to identify events of interest, more specifically the *Executed DbCommand* as shown in Figure 3.5, which provides critical information regarding data manipulation in the SQL databases. In this step the database SQL logs also require a filtering processing. Similarly to the MSA system logs, the SQL logs contain a large amount of information that is not directly related to data dependencies. The SQL logs are filtered based on their *Action ID* property. More specifically we extract events that have their *Action ID* value set as either *INSERT, UPDATE* or *DELETE*. The reasoning for this criterion is these three types of events are directly correlated with the creation, modification and deletion of data, meaning any events of these types will provide essential information towards data dependencies throughout the system. A more in-depth explanation for this process is provided in Section 4.1.

Cross-Referencing SQL Database Events with Microservice Names

In the second phase, the filtered event logs undergo a cross referencing process using additional system logs. Cross referencing is required due to incompleteness and inconsistencies between



bodies of water		attributes					
		temporary	*running*	*natural*	*stagnant*	*constant*	*maritime*
objects	canal		✓			✓	
	channel		✓			✓	
	lagoon			✓	✓	✓	✓
	lake			✓	✓	✓	
	maar			✓	✓	✓	
	puddle	✓		✓	✓		
	pond			✓	✓	✓	
	pool			✓	✓	✓	
	reservoir				✓	✓	
	river		✓	✓		✓	
	rivulet		✓	✓		✓	
	runnel		✓	✓		✓	
	sea			✓	✓	✓	✓
	stream		✓	✓		✓	
	tarn			✓	✓	✓	
	torrent		✓	✓		✓	
	trickle		✓	✓		✓	

Figure 3.9: An example FCA table [38]

logging formats in respects to the labelling of data. The SQL logs provide useful information in terms of data creation, modification and deletion, however they provide no insight into the progressive flow of events that led to the SQL event occurring. These types of incompleteness is something that must be addressed in order to develop a complete dependency graph model. The first step towards data comprehension for the SQL events is to determine which microservice instantiated the SQL event. This is partially achieved through the SQL event property *Client IP*, as shown in Figure 3.6 which depicts a single *INSERT* SQL event. This property provides us information regarding to the source IP that instantiated the event, however this is where the inconsistencies between logging formats occur. In the previously analyzed MSA event logs, there are zero references to microservice IP's, as a result further information must be retrieved. This is accomplished through querying docker for the IP addresses of each microservice and with this additional data we are able to translate the SQL event's *Client IP* into its corresponding microservice name. Further explanation for this process is provided in

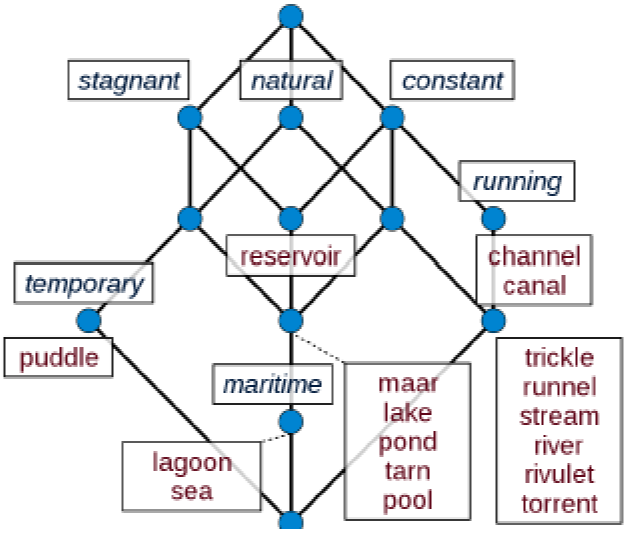


Figure 3.10: An example FCA lattice [38]

Section 4.2.

Cross-Referencing SQL Database Events with Microservice Events

In the third phase, initial event reconciliation occurs between the MSA event logs and SQL logs. From the previous phase we were able to establish the source microservice names for the SQL logs, in this phase we focus on associating which MSA events correspond to which SQL events. An example of an SQL log is depicted in Figure 3.6 and an example of a MSA event log is depicted in Figure 3.7. Separately each event log only contains a subset of the overall data that represents the complete state of the system at the instance of the events execution. This is due to the fact that the microservice event logs lack information indicating their involvement with the SQL databases and vice versa. This can be seen in Figure 3.7 where the data corresponding to the event creation and event contents are shown but provide no information regarding the SQL database manipulation. In contrast to the SQL events like the one shown in Figure 3.6, in which they contain information regarding the SQL data creation and the data contents but provide no information regarding the source event that initiated these data manipulations. The initial process for the association between MSA events and SQL events is achieved through an algorithm that establishes matches and potential matches based on specified criteria. The first criterion used to determine whether a MSA event and SQL event are

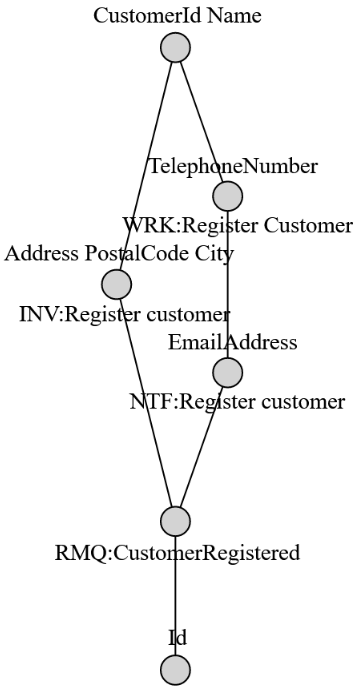


Figure 3.11: A subset of the Pitstop systems FCA lattice

a match is based on their timestamps, in order for the events to be associated they both must occur within a responsible time frame of each other. The second criterion used, is based on their data attributes. In order for the events to be associated, they must also contain the same or a subset of each others data attributes. A thorough explanation for this process is provided Section 4.3.

3.3.2 Conceptual Event Association

Attribute Synonym Synchronization

The first phase is a supplementation to overcome the short comings in the third phase of Section 3.3.1. In the third phase of Section 3.3.1 there is portion of the SQL events that were not able to be associated with MSA events. This corner case is not a product of nonexistent associations, rather it due to events being matches only based on their time stamps. This means that the two events did not contain any subset of matching attributes. This however does not mean the events are not associated with each other, instead this simply raises an issue with the event attributes. This corner case is a result of attributes that reference the same type of data having different naming conventions. An example of this can be seen in Figure 3.8, in which the two events should be associated because they contain instances of the same data. However, a closer inspection reveals that the attribute pairs *(JobId, LicenseNumber)* and *(Id, VehicleLicenseNumber)* reference the same data values but are labelled different. This inconsistency in the data results in the algorithm from the previous step not able to match these types of attributes, even though they should be matched. The solution for this problem is through the usage of the *Attribute Synonyms* list. The *Attribute Synonyms* is a collection of attributes and their corresponding synonyms. The collection is established through the attribute and value comparison throughout all other established MSA events, in which any occurrances like the one previously described are collected. The *Attribute Synonym* collects all the attributes across the established MSA events and extracts any instances of identical values with inconsistent attribute labels. Through this additional data implemented into another algorithm, we are able to establish the remaining MSA events with their corresponding SQL events. A detailed explanation of this process and its corresponding algorithms are further explained in Section 4.4.1.

Event Association - Conceptual Method

Previously we have sought to establish the association between MSA events and SQL events. In this part we focus on a conceptual method for establishing associations between MSA events. The methodology discussed here provides the initial ground work required for developing an automated process capable of establishing the event associations. In this part event associations are established as either *properMatches* or *partialMatches*. Event pairs are considered *properMatches* if they occur within a predefined time-frame and if either event's attribute values are a subset of the other event's attribute values. If an event pair only meet the time-frame criteria then they are established as a *partialMatch*. The first phase of this part begins with matching events based on their timestamps. In order to efficiently determine which event logs may be associated with each other, we first apply the criteria that two events have the potential of being associated with each other so long they occur within a responsible time frame. The responsible time frame established is a time frame of *+/- 500 milliseconds*.

In the second phase, the pairs of MSA events that meet the timestamp criteria are iterated through and examined for the second criteria. In this phase, events are considered to be a *properMatch* if the attribute value list of one event is a subset of the other event's attribute values. If the criterion is not met then the event pairs are established as a *partialMatch*. Once all the event pairings have been examined and all the initial *properMatches* have been established, then further analysis is conducted on the *partialMatches*.

The *partialMatches* are further analyzed in order to find any pairs that should be classified as *properMatches* but were not since they did not pass the second criterion. The second criterion was not applied as a strict restriction, but rather as a form of filtering the data by establishing the initial *properMatches*. The second criterion states that one of the event's attribute values must be a subset of the other event's attribute values. However, in a distributed system with various databases instances connected to various services, a receiving microservice may supplement the receiving data with other relevant data obtained elsewhere, in which case it would result in a response event containing more than just the receiving data. In order to capture this scenario, the *partialMatches* are analyzed using the criteria that states

"If one or both of the events in a partialMatch are not apart of an already established properMatch, then the partialMatch is considered a properMatch". This additional criteria captures the scenario previously described.

A detailed description for the process explained above is provided in Section 4.4.2.

Automating the Attribute Synonym Process

In the previous section, a conceptual framework to identify associations between events was established. In this section, the focus is on developing a methodology capable of achieving the same associations through an automated method.

We consider that each event is conforming with a schema and is represented as a JSON object with one or more attributes. An example event in JSON format is depicted in Figure 3.7. The schema reconciliation step has two phases. In the first phase, synonyms of attribute names are identified manually (e.g. *VehicleLicenseNumber* and *LicenseNumber* or *JobID* and *Id* are identified as synonyms - see Fig. 3.8). This a manual process but it can be automated using techniques proposed for schema matching such as the ones in [50] and [53]. In the second phase, *Formal Concept Analysis* (FCA) [27] is applied on the event schemas. In FCA theory, a *formal concept* is defined to be a pair (*A, B*), where *A* is a set of objects (called the *extent*) and *B* is a set of attributes (the *intent*) such that: a) the *extent A* consists of all objects that share the attributes in *B*, and dually; b) the *intent B* consists of all attributes shared by the objects in *A*. This creates a lattice where the nodes denote objects with attributes. The top element of the lattice (i.e. the most general concept) contains all objects and their common features (if any), while the bottom element of the lattice (i.e. the most specialized concept) contains the all the objects containing all the features. In the lattice an attribute *v* involves all objects at and above the node at which the attribute appears, while an object *A* is required for all attributes at and below the node at which the object appears. An example lattice is depicted in Figures 3.9 and 3.10 adapted from [38], while Table 4.1 and Figure 4.8 depict the actual objects and attributes denoted by analyzing the schemas of the sample microservice system we have experimented with [60].

An example of a subset of the Pitstop [60] lattice is shown in Figure 3.11. This lattice represents four events and eight attributes, as shown below:

```
EVENTS
('WRK:Register Customer')
('INV:Register Customer')
('NTF: Register Customer')
('RMQ:CustomerRegistered')

ATTRIBUTES
('CustomerId')
('Name')
('TelephoneNUmber')
('Address')
('PostalCode')
('City')
('EmailAddress')
('Id')
```

where, *WRK* represents *WorkshopManagementEventHandler*, *INV* represents *InvoiceService*, *NTF* represents *NotificationService*, and *RMQ* represents *Rabbitmq*.

From the structure of the lattice we can derive the relationships between events and attributes. For example, the attributes *'CustomerId'* and *'Name'* are at the top of the lattice which means all the events below contain these attributes. In contrast, the attribute *'Id'* is at the bottom of the lattice which means none of the events above contain this attribute. Similarly, the event *RMQ:CustomerRegistered* contains all the attributes in the events above itself, in this case the events are *'NTF:Register Customer', 'INV:Register Customer'* and *WorkshopManagementEventHandler*. The event *'NTF:Register Customer'* is below the event *WorkshopManagementEventHandler*, which means it contains all the attributes in *WorkshopManagementEventHandler*. In contrast to *RMQ:CustomerRegistered* which is below *NTF:Register Customer*, meaning the attributes in *NTF:Register Customer* are also in *RMQ:CustomerRegistered* except *RMQ:CustomerRegistered* contains attributes from the path that is in parallel to *NTF:Register Customer*. The parallel path is the one containing *INV:Register Customer*, meaning the attributes in *RMQ:CustomerRegistered* are the same as *NTF:Register Customer* except it also contains the attributes *'Address', 'PostalCode'* and *'City'* which come are found in *INV:Register Customer*. As described, the FCA lattice can provide a lot of insight into the relationship between events.

A detailed analysis and description of this process and the algorithms used are provided in Section 4.4.3.

3.4 System-Wide Event Matching

The automation of the event association step has two parts. In the first part, the logs are traversed and a collection of *facts* are emitted. A typical fact is of the form *feature(attribute, event)* indicating that *event* has *attribute*. A complete list of facts emitted in this phase of the process are listed in Section 5.1.1. A thorough description and analysis of this step is provided in Section 5.1.1.

In the second part, FCA results obtained previously in the second step, are used to compile rules as the ones depicted in Figure 5.1.2 to be fed to Alchemy [1], a Markov Logic Network inferencing engine. More specifically, the rules fed to Alchemy [1] are trained on the specific log data set. Training the rules means that a numerical weight is assigned to each rule. A higher weight indicates that the rule is more important. We have opted to use 50% of the log data set for training and the other 50% for testing. The result of the inferencing is a collection of facts indicating whether two events are matched, that is they belong to the same chain of events for a given transaction. A detailed description of the rule compilation process is provided in Section 5.1.2. While an in-depth description of the MLN implementation is provided in Section 5.1.3.

3.5 Microservice Dependency Graph Extraction

3.5.1 Event Chains

In this phase the previously associated event *pairs* are cross referenced with each other so that a chain of events is created. This is required since a list of event pairs only provides us with the information that two events are associated with each other, it does not provide any insight into the totality of events related to the specific event flow throughout the system. Thus, we must take the list of pairs of events and apply the transitive property. For example, if we have the event pairs *(EventA, EventB)* and *(EventB, EventC)* then using the transitive property we can establish that the three events, *(EventA, EventB, EventC)*, are all associated with each other in a chain. We apply this property to all the events and generate a list of chains, each chain containing a list of events that are associated with each other. The detailed process for the event chain extraction is further explained in Section 6.2.

3.5.2 Path Extraction and Graph Formation

In the previous phase we derived a collection of event chains (i.e. groups) that are associated with each other. However, this provides no information in regards to the logical flow of the events. More specifically, the groups do not provide information about the ordering of the events. For example, the event group *(EventA, EventB, EventC)* does contain information about which of the three events is the source event, which one is the second and which is the third in sequence. Furthermore, we are not able to make any deductions regarding the type of paths created by these events. For example, do all three events occur in a linear fashion, *(EventA→EventB→EventC)*, or are there two separate paths such as *(EventA→EventB)* and

(EventA→EventB→EventC). In order to determine the ordering we consult the associations with the SQL server, which may give us information about causal order (i.e. an event happens before its observable effects). More specifically, as it will be discussed later in this thesis, we consider a relationship *happenedBefore* in order to mode causality. This relationship can be derived to certain extend by the logs. For example, from the component logs we establish that *EventA* originates from the middleware, then the middleware distributes the event to all other components generating new events (causality between receiving a stimuli and reacting to it), such as the events *EventB* and *EventC* which can be considered response events to the event originating from the middleware. Using the *happenedBefore* relation we are able to establish to a certain extend the relative ordering, that is *EventB* and *EventC* occurs after *EventA*. Using this type of analysis we iterate through all the event groups and we derive corresponding paths. The entirety of the path extraction process is explained in detail in Section 6.3.

Once we have the list of all the derived paths we can then generate the MDG multigraph. This process is accomplished through an algorithm that iterates through each event in a path and adds a node for each microservice and an edge between each pair of consecutive events in the path. An in-depth explanation of this process is provided in Section 6.4.

The following Chapters discuss all of the steps described above in more detail.

Chapter 4

Event Association and Schema Reconciliation

4.1 Filtering Data

In order to reduce the volume of the logged events to be processed and make the system more tractable, we apply an event filtering technique so that we keep only events pertinent to compiling a microservice dependency graph.

The initial part of filtering exploits the HTTP based messaging between microservices. The analysis of HTTP requests/replies of message broker's log data can reveal a group of HTTP tuples containing the initial *Start processing HTTP request*, the final *End processing HTTP message* and all events that have occurred in-between. More specifically, each group corresponds to a different initial type of HTTP request (POST or GET), its corresponding reply or its corresponding *End processing HTTP message*, and all the events occurred in between. An example can be seen in Figure 4.1. In this example partial data for the events corresponding to one *HTTP request* partition is shown. In the provided example only partial data is shown, the @*t* and @*mt* components are shown, in order to avoid cluttering. The partition begins with an event containing the *Start processing HTTP request* and ends with the event containing the *End processing HTTP request*.

After the logs have been filtered and organized into lists of various HTTP message types, further analysis is conducted on the events captured during the HTTP message's timeline in order to obtain additional insights into the data interactions between microservices. The analysis is based on extracting information related to data manipulation requested (e.g. the request for a CREATE, READ, UPDATE or DELETE operation to a corresponding data store). For example, the occurrence of an embedded SQL INSERT statement along with its arguments reveals data exchanges between a microservice and a database. An example is shown in the highlighted HTTP event in Figure 4.1 and the complete logged event is shown in Figure 4.2.

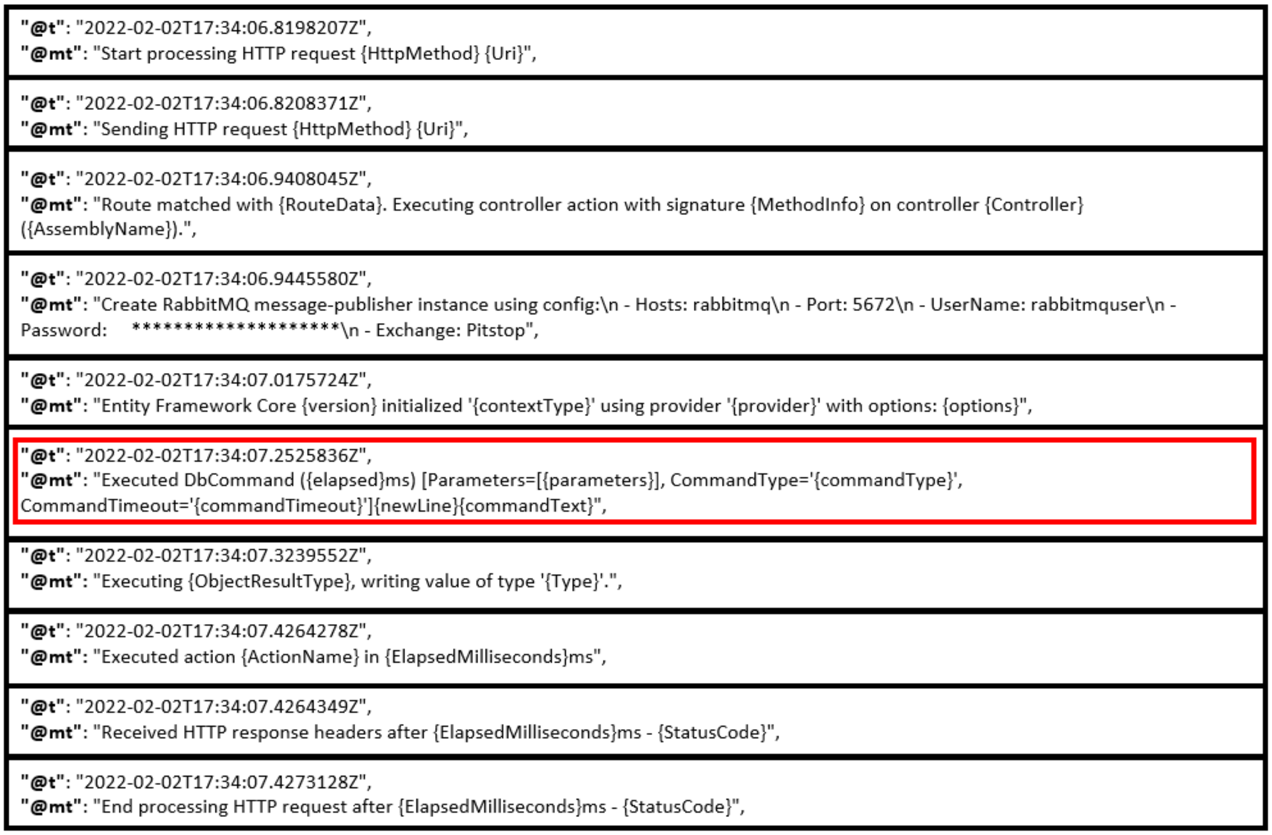
The identification of an SQL command within the logs provides initial evidence of data dependencies. Using the event log's message template, which can be seen in Figure 4.2 as the *"@mt"* attribute, the full event logs can then be queried for any other occurrences of a similar instance of a SQL command.

Consequently, the SQL statement parameters are extracted from each event that references the corresponding enclosed SQL statement. The parameters of the SQL statement are characterized by their enclosure using brackets. This can be seen once again in the highlighted portion of Figure 4.2 with some of the example parameters including *(CustomerId, Address, City, EmailAddress, Name, Postal Code, TelephoneNumber)*. The collection of the extracted SQL parameters can then be cross-referenced against the full log corpus to select any *Non-Executed DbCommand* event that contain references to the same parameters.

The resulting output returns a list of microservice events containing elements from the SQL statement parameters. An example of non-message-broker events that were found using this pattern matching method can be seen in Figure 4.3, with the pattern matched SQL statement parameters being highlighted. The highlighted attributes, *"Id", "Name", "Address", "PostcalCode", "City", "TelephoneNumber"* and *"Email"* all correspond to *Executed DbCommand* events that contained those attributes as SQL Statement parameters.

The final part of filtering data is the filtering of the SQL data. Prior to filtering out any non-relevant SQL events, the raw log file must first be processed and formatted into a quarriable data structure. The reasoning for the restructuring of the SQL logs can be seen in Figure 4.4, in which the raw log data is shown to not be properly organized. Thereforethe SQL logs are required to be formatted into something more efficient. Figure 4.5 depicts a dictionary data structure containing all the data of a single SQL event.

After the SQL events have been properly formatted, the SQL events must undergo a filtering process. Similarly to the MSA event logs, the SQL logs are filtered in order to reduce the volume of data needed to be processed. The SQL filtering is based on the events corresponding to the *INSERT, UPDATE, DELETE* commands. The reasoning for these three specific commands, is due to the fact that these type of commands are directly related to the creation, modification and deletion of data.

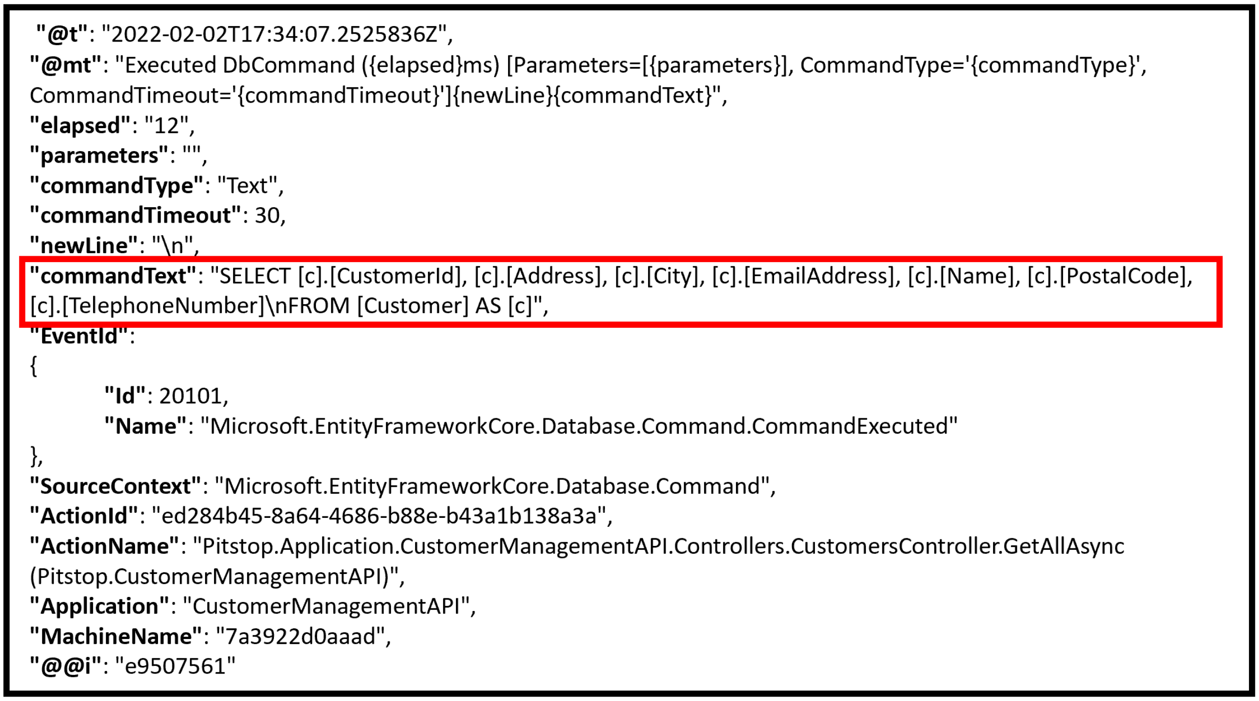


```
"@t": "2022-02-02T17:34:06.8198207Z",
"@mt": "Start processing HTTP request {HttpMethod} {Uri}",

"@t": "2022-02-02T17:34:06.8208371Z",
"@mt": "Sending HTTP request {HttpMethod} {Uri}",

"@t": "2022-02-02T17:34:06.9408045Z",
"@mt": "Route matched with {RouteData}. Executing controller action with signature {MethodInfo} on controller {Controller}
({AssemblyName}).",

"@t": "2022-02-02T17:34:06.9445580Z",
"@mt": "Create RabbitMQ message-publisher instance using config:\n - Hosts: rabbitmq\n - Port: 5672\n - UserName: rabbitmquser\n -
Password:    ********************\n - Exchange: Pitstop",

"@t": "2022-02-02T17:34:07.0175724Z",
"@mt": "Entity Framework Core {version} initialized '{contextType}' using provider '{provider}' with options: {options}",

"@t": "2022-02-02T17:34:07.2525836Z",
"@mt": "Executed DbCommand ({elapsed}ms) [Parameters=[{parameters}], CommandType='{commandType}',
CommandTimeout='{commandTimeout}']{newLine}{commandText}",

"@t": "2022-02-02T17:34:07.3239552Z",
"@mt": "Executing {ObjectResultType}, writing value of type '{Type}'.",

"@t": "2022-02-02T17:34:07.4264278Z",
"@mt": "Executed action {ActionName} in {ElapsedMilliseconds}ms",

"@t": "2022-02-02T17:34:07.4264349Z",
"@mt": "Received HTTP response headers after {ElapsedMilliseconds}ms - {StatusCode}",

"@t": "2022-02-02T17:34:07.4273128Z",
"@mt": "End processing HTTP request after {ElapsedMilliseconds}ms - {StatusCode}",
```

Figure 4.1: Database event (highlighted) within the partial event data for one HTTP Request partition

4.2 Cross-Referencing SQL Database Events with Microservice Names

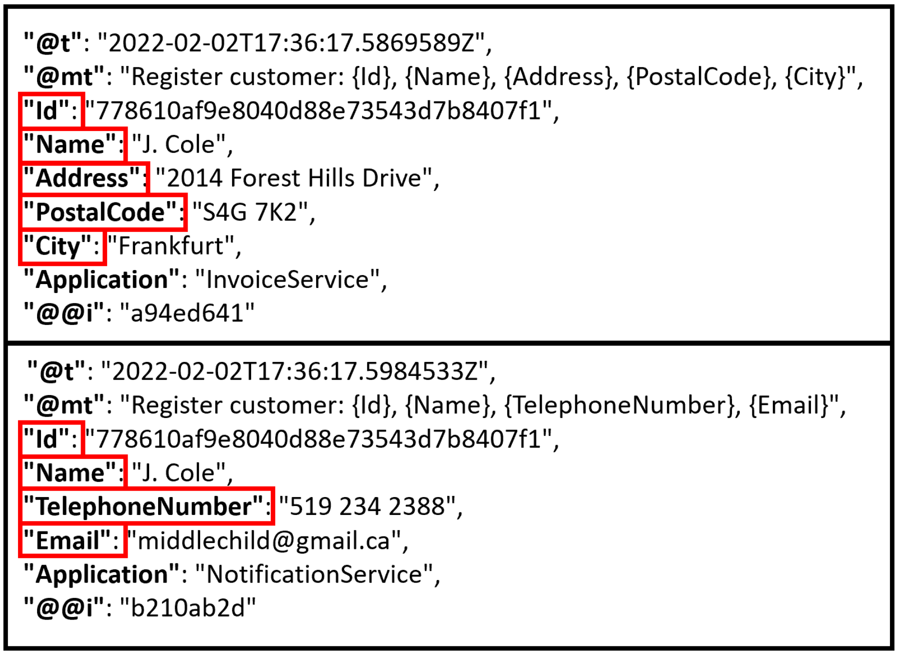
The database event logs undergo a cross referencing process with additional system logs. In the previous phase database events and the events emitted by all other components were selected based on their attribute values. However, individually each of these log events does



```
 "@t": "2022-02-02T17:34:07.2525836Z",
"@mt": "Executed DbCommand ({elapsed}ms) [Parameters=[{parameters}], CommandType='{commandType}',
CommandTimeout='{commandTimeout}']{newLine}{commandText}",
"elapsed": "12",
"parameters": "",
"commandType": "Text",
"commandTimeout": 30,
"newLine": "\n",
"commandText": "SELECT [c].[CustomerId], [c].[Address], [c].[City], [c].[EmailAddress], [c].[Name], [c].[PostalCode],
[c].[TelephoneNumber]\nFROM [Customer] AS [c]",
"EventId":
{
        "Id": 20101,
        "Name": "Microsoft.EntityFrameworkCore.Database.Command.CommandExecuted"
},
"SourceContext": "Microsoft.EntityFrameworkCore.Database.Command",
"ActionId": "ed284b45-8a64-4686-b88e-b43a1b138a3a",
"ActionName": "Pitstop.Application.CustomerManagementAPI.Controllers.CustomersController.GetAllAsync
(Pitstop.CustomerManagementAPI)",
"Application": "CustomerManagementAPI",
"MachineName": "7a3922d0aaad",
"@@i": "e9507561"
```

Figure 4.2: An example log of a message broker's containing SQL data

not contain sufficient information to establish proper associations with other events. Therefore, more information has to be considered. As depicted in Figure 4.4, in a database logged event, the only data representative to the microservice which originated the database event is the database event attribute *Client IP*. However, aside from the SQL event logs, there does not exist microservice logs that contain information related the IP address of the microservice. In order to obtain this information, the docker containers listing the microservice must be dynamically queried, this is due to the fact that the IP address for each docker container varies upon system initialization. Once the docker container IP information has been retrieved we can then cross reference all the database logged events (i.e .the SQL log events) and match the *Client IP* with the corresponding container name, which in this case is the microservice's name. The end result is the ability to associate each database event with the microservice name that caused it. Still, we need to go one step deeper, that is associate the specific microservice *event* that caused the specific database event. This is discussed in detail in the following section below.



```
"@t": "2022-02-02T17:36:17.5869589Z",
"@mt": "Register customer: {Id}, {Name}, {Address}, {PostalCode}, {City}",
"Id": "778610af9e8040d88e73543d7b8407f1",
"Name": "J. Cole",
"Address": "2014 Forest Hills Drive",
"PostalCode": "S4G 7K2",
"City": "Frankfurt",
"Application": "InvoiceService",
"@@i": "a94ed641"
```

```
"@t": "2022-02-02T17:36:17.5984533Z",
"@mt": "Register customer: {Id}, {Name}, {TelephoneNumber}, {Email}",
"Id": "778610af9e8040d88e73543d7b8407f1",
"Name": "J. Cole",
"TelephoneNumber": "519 234 2388",
"Email": "middlechild@gmail.ca",
"Application": "NotificationService",
"@@i": "b210ab2d"
```

Figure 4.3: An example of two events with attributes highlighting the SQL parameter data pattern matching

4.3 Cross-Referencing SQL Database Events with Microservice Events

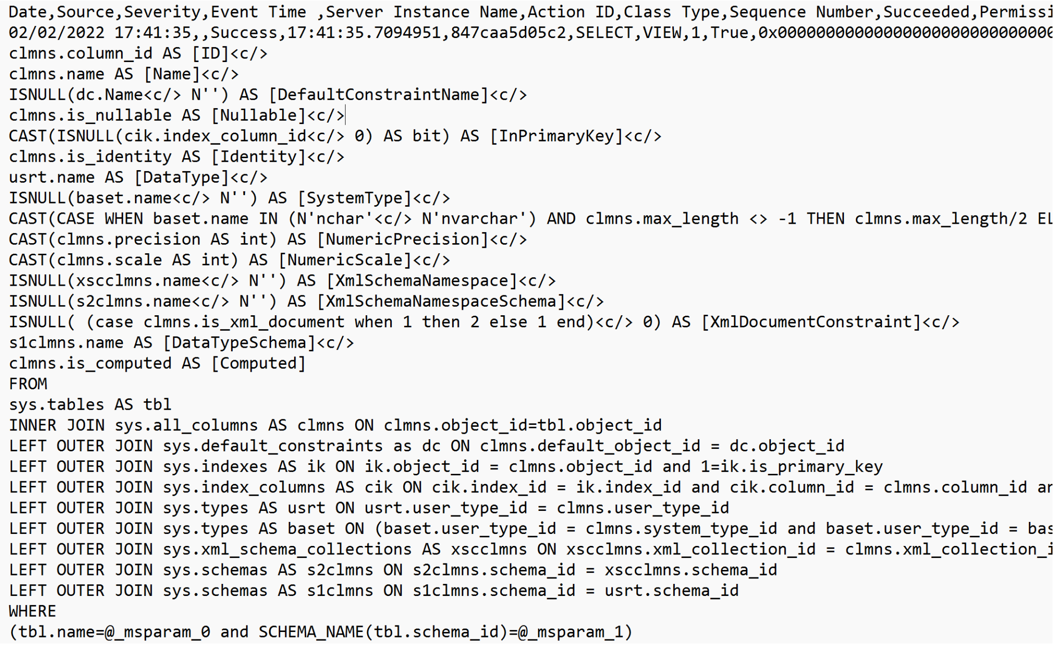
In the previous step, a specific database SQL event was associated with a microservice *name*. Here, we proceed the analysis further, for associating a specific database SQL event with a specific causing microservice *event*. The algorithm used to match a database event logs with a microservice event (from the microservice identified in the previous step) is depicted in Algorithm 4.1.

The algorithm consists of two parts. In the first part of the algorithm determines potential matches based on the events timestamps and the intersecting set between the event's attribute sets. After all potential matches have been found, the algorithm then goes through all potential matches and determines which potential match is most likely to be the correct match.

Algorithm 4.1 takes two parameters *MicroserviceEvents* and *SqlEvents*. The first parameter *MicroserviceEvents* is a list containing all the microservice events that have been previously established to contain relevant dependency data. Similarly, the second parameter is a list con-

Algorithm 4.1 MSA and SQL Event Reconciliation

```
– Let $MicroserviceEvents$ be the set of microservice events derived from previous phases
– Let $SqlEvents$ be the set of SQL events derived from previous phases
- Let $MatchedEvents$ be the set of pairs $\langle sql, microservice \rangle$ of matched events
- Let $potentialEvents$ be a list of potential matching events
- Let $timeEvents$ be a list of time matching events
- Let $IncompleteEvents$ be the set of pairs $\langle microservice, timeEvents \rangle$

procedure reconcileEvents($MicroserviceEvents$, $SqlEvents$)
    for each $event_m$ in $MicroserviceEvents$ do
        $microserviceAttributes$ = set($event_m$.getAttributes())
        $microserviceTimestamp$ = $event_m$.getTimeStamp()
        $potentialEvents$ = []
        $timeEvents$ = []
        for each $event_s$ in $SqlEvents$ do
            $sqlAttributes$ = set($event_s$.getAttributes())
            $sqlTimestamp$ = $event_s$.getTimeStamp()
            if $sqlTimestamp$ is within the timeframe of $microserviceTimestamp$ then
                $intersection$ = $sqlParameters \cap microserviceParameters$
                if $intersection$ not empty then
                    $potentialEvents$.append($event_s$)
                else
                    $timeEvents$.append([$event_s$])
                end if
            end if
        end for
        $matchFound$ = False
        for $event_p$ in $potentialEvents$ do
            if $event_p$.getMicroservice() == $event_m$.getMicroservice() then
                $MatchedEvents$.append([$event_m$,$event_p$])
                $matchFound$ = True
                break
            end if
        end for
        if matchFound == False then
            $IncompleteEvents$.append([$event_m$, $timeEvents$])
        end if
    end for
    return $MatchedEvents$, $IncompleteEvents$
end procedure
```



```
Date,Source,Severity,Event Time ,Server Instance Name,Action ID,Class Type,Sequence Number,Succeeded,Permissi
02/02/2022 17:41:35,,Success,17:41:35.7094951,847caa5d05c2,SELECT,VIEW,1,True,0x00000000000000000000000000000
clmns.column_id AS [ID]<c/>
clmns.name AS [Name]<c/>
ISNULL(dc.Name<c/> N'') AS [DefaultConstraintName]<c/>
clmns.is_nullable AS [Nullable]<c/>
CAST(ISNULL(cik.index_column_id<c/> 0) AS bit) AS [InPrimaryKey]<c/>
clmns.is_identity AS [Identity]<c/>
usrt.name AS [DataType]<c/>
ISNULL(baset.name<c/> N'') AS [SystemType]<c/>
CAST(CASE WHEN baset.name IN (N'nchar'<c/> N'nvarchar') AND clmns.max_length <> -1 THEN clmns.max_length/2 El
CAST(clmns.precision AS int) AS [NumericPrecision]<c/>
CAST(clmns.scale AS int) AS [NumericScale]<c/>
ISNULL(xscclmns.name<c/> N'') AS [XmlSchemaNamespace]<c/>
ISNULL(s2clmns.name<c/> N'') AS [XmlSchemaNamespaceSchema]<c/>
ISNULL( (case clmns.is_xml_document when 1 then 2 else 1 end)<c/> 0) AS [XmlDocumentConstraint]<c/>
s1clmns.name AS [DataTypeSchema]<c/>
clmns.is_computed AS [Computed]
FROM
sys.tables AS tbl
INNER JOIN sys.all_columns AS clmns ON clmns.object_id=tbl.object_id
LEFT OUTER JOIN sys.default_constraints as dc ON clmns.default_object_id = dc.object_id
LEFT OUTER JOIN sys.indexes AS ik ON ik.object_id = clmns.object_id and 1=ik.is_primary_key
LEFT OUTER JOIN sys.index_columns AS cik ON cik.index_id = ik.index_id and cik.column_id = clmns.column_id ar
LEFT OUTER JOIN sys.types AS usrt ON usrt.user_type_id = clmns.user_type_id
LEFT OUTER JOIN sys.types AS baset ON (baset.user_type_id = clmns.system_type_id and baset.user_type_id = bas
LEFT OUTER JOIN sys.xml_schema_collections AS xscclmns ON xscclmns.xml_collection_id = clmns.xml_collection_i
LEFT OUTER JOIN sys.schemas AS s2clmns ON s2clmns.schema_id = xscclmns.schema_id
LEFT OUTER JOIN sys.schemas AS s1clmns ON s1clmns.schema_id = usrt.schema_id
WHERE
(tbl.name=@_msparam_0 and SCHEMA_NAME(tbl.schema_id)=@_msparam_1)
```

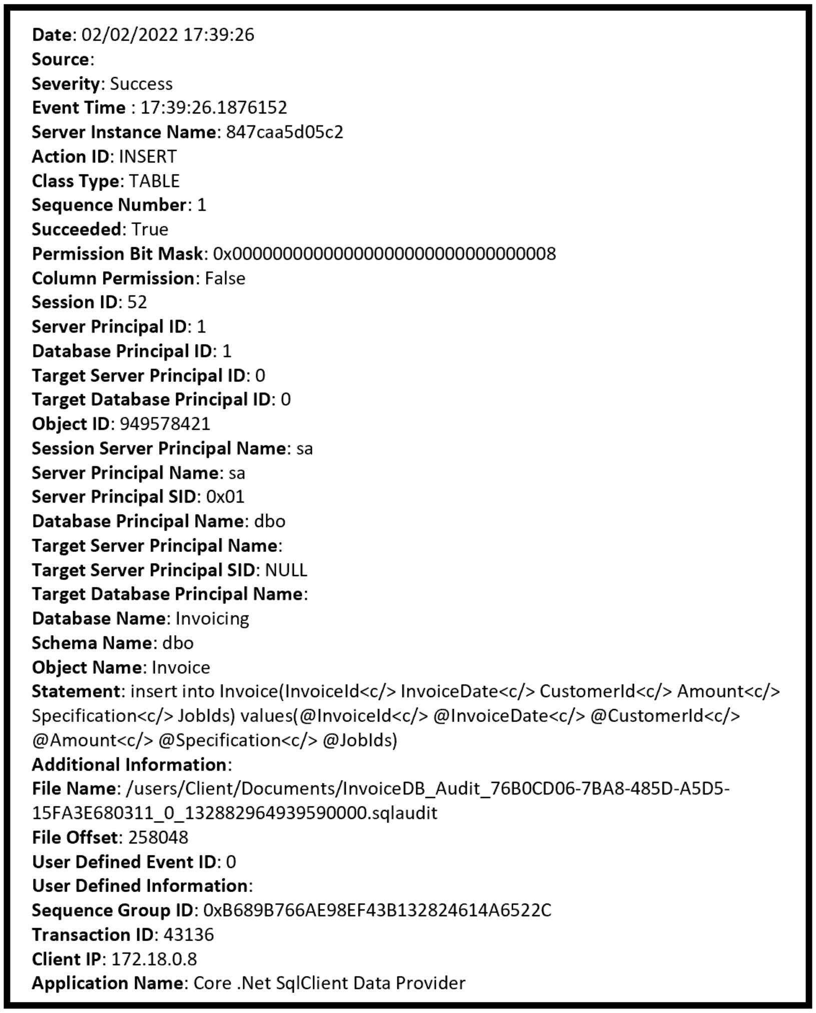
Figure 4.4: An example of a raw SQL log file

taining all the database SQL events containing relevant dependency data. Both of these event lists were obtained from the process discussed in Sections 4.1 and 4.2.

In the beginning of the algorithm, *lines 8-11*, the algorithm loops through all microservice events in the *MicroserviceEvents* list and extracts the event's attributes and timestamp information. The event attributes are stored in a set. Next, in *lines 12-13* two lists are instantiated. The first list, *potentialEvents*, will contain all potential events. Potential events are defined as events that occur within a predefined time-frame of each other and share at least one attribute in common. The second list, *timeEvents*, will contain all the time events. Time events are defined as events that occur within a predefined time-frame of each other, but do not contain any attributes in common.

Next, the algorithm iterates through all the SQL events in the *SqlEvents* and extracts the attributes and timestamp information from each SQL event, as shown in *lines 14-16*. In *lines 17* the algorithm checks whether the current SQL event occurred within an established time-frame(+/- 0.5seconds) of the current microservice event. This is the initial determining attribute that is checked to determine whether or not a microservice event and SQL can be considered match.

In order for the event pairs to be identified as potential event matches, the intersection



Date: 02/02/2022 17:39:26
Source:
Severity: Success
Event Time : 17:39:26.1876152
Server Instance Name: 847caa5d05c2
Action ID: INSERT
Class Type: TABLE
Sequence Number: 1
Succeeded: True
Permission Bit Mask: 0x00000000000000000000000000000008
Column Permission: False
Session ID: 52
Server Principal ID: 1
Database Principal ID: 1
Target Server Principal ID: 0
Target Database Principal ID: 0
Object ID: 949578421
Session Server Principal Name: sa
Server Principal Name: sa
Server Principal SID: 0x01
Database Principal Name: dbo
Target Server Principal Name:
Target Server Principal SID: NULL
Target Database Principal Name:
Database Name: Invoicing
Schema Name: dbo
Object Name: Invoice
Statement: insert into Invoice(InvoiceId<c/> InvoiceDate<c/> CustomerId<c/> Amount<c/> Specification<c/> JobIds) values(@InvoiceId<c/> @InvoiceDate<c/> @CustomerId<c/> @Amount<c/> @Specification<c/> @JobIds)
Additional Information:
File Name: /users/Client/Documents/InvoiceDB_Audit_76B0CD06-7BA8-485D-A5D5-15FA3E680311_0_132882964939590000.sqlaudit
File Offset: 258048
User Defined Event ID: 0
User Defined Information:
Sequence Group ID: 0xB689B766AE98EF43B132824614A6522C
Transaction ID: 43136
Client IP: 172.18.0.8
Application Name: Core .Net SqlClient Data Provider

Figure 4.5: An SQL event stored in a dictionary structure

between the two attribute sets (*sqlAttributes and microserviceAttributes*) must not be empty. Therefore, if the event pairs share a minimum of one attribute in common, then the algorithm will append the current database event into a list of potential matches for the current microservice event. If the two events do not share any attributes in common, then the SQL event is appended onto the *timeEvents* list. This criteria is defined in *lines 18-22*

The third criteria for establishing the SQL event and MSA event association, is through the source microservice of the SQL Event that was established in Section 4.2. In *lines 26* the algorithm initiates a Boolean variable *matchFound*, in order to keep track of whether or not a SQL match has been found. In *lines 27-31*, the algorithm iterates through all potential database event pair matches (i.e. the ones that have similar timestamp and attribute values with the microservice event) and selects the database event with a source microservice that matches paired microservice event. If the pair of events meet all three criteria then they are established as *MatchedEvents* and the pair is appended to the*MatchedEvents* list. Additionally the Boolean variable *matchFound* is set to *True* and the for loop is exited. Once the iteration is either completed or exited, the algorithm will check the value of the Boolean variable *matchFound*. If the value is true, the algorithm continues onto the next $event_m$ in the *MicroserviceEvents* list. If the value is false, the algorithm will pair all the *timeEvents* with the current $event_m$ and append this tuple to the *IncompleteEvents* list. This list of incomplete events will be used for further analysis for compiling the MDG. Lastly, in *lines 38*, the algorithm will return two lists. The first is *MatchedEvents*, a list containing matched pairs of SQL and Microservice events. The second is *IncompleteEvents*, a list containing Microservice events paired with a list of *timeEvents*.

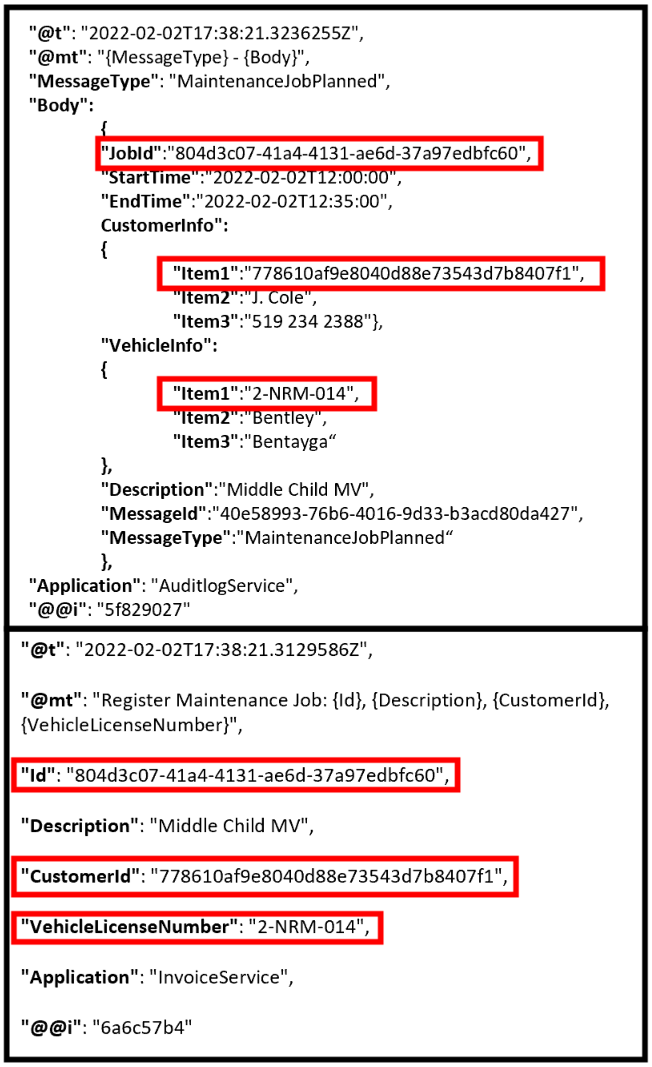
4.4 Event Association

In the previous sections, Section 4.2 and Section 4.3, most of the microservice event logs were matched with their corresponding database logs. The main purpose of this section is to present that process that establishes pairs of events that are associated with each other. These associations will then be used in Chapter 6 for generating the *Microservice Dependency Graph*. The process for establishing these event pairs is illustrated in Algorithm 4.3

4.4.1 Attribute Synonym Synchronization

Previously, in Section 4.3 two lists were obtained from Algorithm 4.1. Notably the list *IncompleteEvents*, containing unmatched microservice events alongside a list of time matched SQL events. The reasoning for this incompleteness is due to the inconsistency in labelling formats between loggings systems. As depicted in Figure 3.1, we assume that the microservice system being analyzed has numerous loggers $M1$, $M2_2$, … M_w for its different components (i.e. microservices, service busses, pub/sub frameworks, databases) and each such logger incorporates its own schema. This introduces the problem that two attributes, in two events, logged by two different loggers, and which refer to the same information are considered as not associated because they do not match by name. The database research community has investigated the problem of *schema matching* for many years and over the years a number of very efficient automated approaches have been proposed [50], [53].

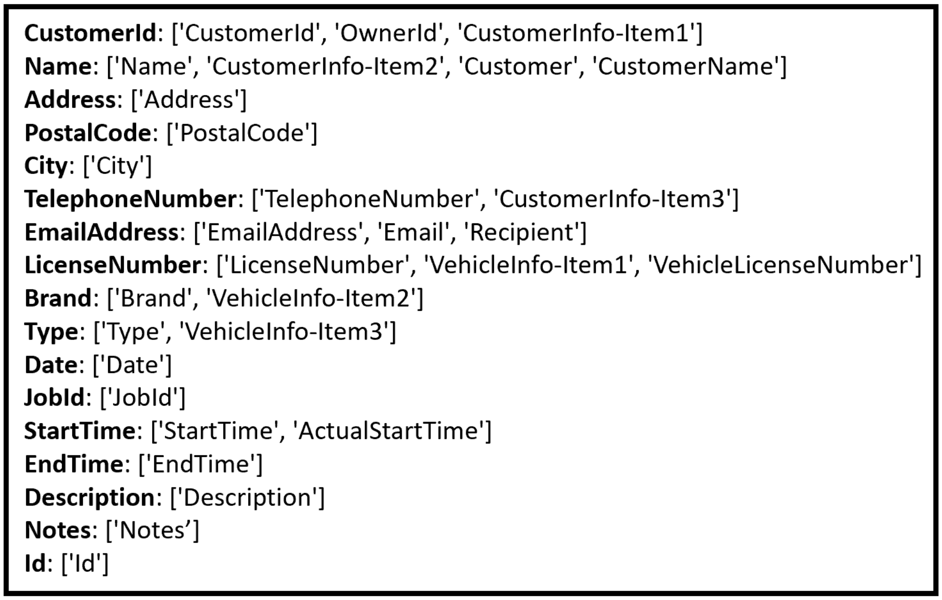
An example of this situation is shown in Figure 4.6, in which the data value for highlighted attribute value pair of the first event match the data value for the highlighted attribute value pair of the second event. In the example the attributes *('JobId', 'Item1', 'Item1')* of the first event, contain the same values as the attributes *('Id', 'CustomerId', 'VehicleLicenseNumber')* in the second event. However as it is shown, the attribute names are inconsistent. Hence the existence of the *IncompleteEvents* list.



```
"@t": "2022-02-02T17:38:21.3236255Z",
"@mt": "{MessageType} - {Body}",
"MessageType": "MaintenanceJobPlanned",
"Body":
          {
          "JobId":"804d3c07-41a4-4131-ae6d-37a97edbfc60",
          "StartTime":"2022-02-02T12:00:00",
          "EndTime":"2022-02-02T12:35:00",
          CustomerInfo":
          {
                    "Item1":"778610af9e8040d88e73543d7b8407f1",
                    "Item2":"J. Cole",
                    "Item3":"519 234 2388"},
          "VehicleInfo":
          {
                    "Item1":"2-NRM-014",
                    "Item2":"Bentley",
                    "Item3":"Bentayga“
          },
          "Description":"Middle Child MV",
          "MessageId":"40e58993-76b6-4016-9d33-b3acd80da427",
          "MessageType":"MaintenanceJobPlanned“
          },
"Application": "AuditlogService",
"@@i": "5f829027"
```

```
"@t": "2022-02-02T17:38:21.3129586Z",

"@mt": "Register Maintenance Job: {Id}, {Description}, {CustomerId},
{VehicleLicenseNumber}",

"Id": "804d3c07-41a4-4131-ae6d-37a97edbfc60",

"Description": "Middle Child MV",

"CustomerId": "778610af9e8040d88e73543d7b8407f1",

"VehicleLicenseNumber": "2-NRM-014",

"Application": "InvoiceService",

"@@i": "6a6c57b4"
```

Figure 4.6: An example illustrating inconsistent labeling formats between logs



CustomerId: ['CustomerId', 'OwnerId', 'CustomerInfo-Item1']
Name: ['Name', 'CustomerInfo-Item2', 'Customer', 'CustomerName']
Address: ['Address']
PostalCode: ['PostalCode']
City: ['City']
TelephoneNumber: ['TelephoneNumber', 'CustomerInfo-Item3']
EmailAddress: ['EmailAddress', 'Email', 'Recipient']
LicenseNumber: ['LicenseNumber', 'VehicleInfo-Item1', 'VehicleLicenseNumber']
Brand: ['Brand', 'VehicleInfo-Item2']
Type: ['Type', 'VehicleInfo-Item3']
Date: ['Date']
JobId: ['JobId']
StartTime: ['StartTime', 'ActualStartTime']
EndTime: ['EndTime']
Description: ['Description']
Notes: ['Notes']
Id: ['Id']

Figure 4.7: A dictionary data structure containing all attribute synonyms

The problem is to be able to identify the pairs of event types that share the maximal set of attributes and corresponding attribute values. For our work, and since the number of microservices was low (i.e. 12 microservices) we initially perfomed performed the synonym analysis manually. The manual process is illustrated in Algorithm 4.2. The manual process is simple. By iterating through all microservice events, the attribute key-value pairs were recorded. Then the key-values pairs were cross referenced with each other and any instance in which the two values matched but their corresponding keys did not, were recorded and the keys are established as synonyms. Our approach utilizes an *Attribute Synonym* dictionary data structure to provide this attribute-level schema mapping. The attribute synonym dictionary is shown in Figure 4.7. The purpose of this attribute synonym dictionary is to aid in the short comings discussed in Section 4.3. The reason why some microservice events were unable to be matched with their corresponding database events were due to the inconsistencies in labeling conventions between the two types of event logs. The pseudo-code for this algorithm is shown in Algorithm 4.2

This manual analysis provided insights on how to automate the process by using Formal Concept Anaysis. The automated process is discussed in Section 4.4.3.

Algorithm 4.2 is very similar to the Algorithm 4.1, in which the a majority of the computation is through the iteration of two event lists.

Algorithm 4.2 Event Reconciliation with Attribute Synonym Supplementation

```
- Let $IncompleteEvents$ be the set of time matched pairs $\langle microserviceEvent, timeEvents \rangle$
- Let $AttributeSynonyms$ be a dictionary containing all the attributes and their corresponding synonyms
- Let $CompletedEvents$ be the set of matching event pairs $\langle microserviceEvent, sqlEvent \rangle$
procedure AttributeSynonymSupplementation($IncompleteEvents$, $AttributeSynonyms$)
    for each $incompleteEvent$ in $IncompleteEvents$ do
        $microserviceEvent = incompleteEvent$.getMicroserviceEvent()
        $microserviceParameters$ = set($microserviceEvent$.getParameters())
        for each $timeEvent$ in $incompleteEvent$ do
            $timeEventParameters$ = set($timeEvent$.getParameters())
            $intersection = timeEventParameters \cap microserviceParameters$
            $differences = timeEventParameters \triangle microserviceParameters$
            for each $difference$ in $differences$ do
                $synonyms = AttributeSynonyms$.getSynonyms($difference$)
                for each $synonym$ in $synonyms$ do
                    if $synonym$ is in $differences$ & $synonym \neq difference$ then
                        $CompletedEvents$.append([$microserviceEvent$,$timeEvent$])
                    end if
                end for
            end for
        end for
    end for
    return $CompletedEvents$
end procedure
```

Algorithm 4.2 takes two parameters *IncompleteEvents* and *AttributeSynonyms*. The first parameter *IncompleteEvents* is the result of the incomplete event matching that occurred in 4.3. The second parameter is the *AttributeSynonyms* which contains all the attribute synonyms for data labelling in the system.

In the beginning, the algorithm, (*line 5*) iterates through all instances events in the unmatched event list. In *lines 6-7*, the algorithm extracts the parameters from the unmatched microservice event. In *lines 8-9* the algorithm iterates through the list of database events which have been considered to be a match with the microservice events based only on their timestamp attribute and also extracts their parameters. In *lines 10-11* the intersection and symmetric difference between the *microserviceParameters* and *timeEventParameters* sets is computed. This is the main difference between this algorithm and the Algorithm 4.1, where microservice event and data event matches were missed because of different naming conventions in the corresponding schemas. This issue is resolved in the remaining portions of the algorithm. In *lines 12-13* the *AttributeSynonym* is used to retrieve a list of synonyms for each symmetric difference between the two parameter sets. In *lines 14-16* the algorithm iterates through all the retrieved synonyms and checks whether there exists a synonym that also exists in the symmetric difference (*differences*) set but is not the same as the current *difference*. Thus, the algorithm will check if the symmetric difference list contains a matching synonym that corresponds to any except the current difference. If so, then current *microserviceEvent* and *timeEvent* are paired and appended to *CompletedEvents*. The pairing is established because the algorithm was able to find the existence of matching parameters and since the *timeEvents* already determined the events to be a match based on their time-frame then there is no need for repeating this same attribute verification.

4.4.2 Event Association - Conceptual Method

In the previous section, Section 4.4.1, the previously unmatched microservice events have now been matched with their corresponding SQL events. Now the process of establishing pairs of microservice events associated with each other can begin. This process is established in Algorithm 4.3.

There are two scenarios in which an event pair can be established as a *properMatch*. The first scenario is if they meet the following two criteria. The first criterion is based on the pair of events occurring within a predefined time-frame, meaning two events must occur within 500milliseconds otherwise they do not meet the first criterion. The second criterion is one of the event's attribute values is a complete subset of the other event's attribute values. If a pair of events meet both of these criteria, then they are established as *properMatches*. The second scenario is a continuation of the first scenario. In the scenario where an event pair only meets the first criterion, then they are established as a *partialMatch*. Once all the *properMatches* have been established, the *partialMatches* are cross referenced with the *properMatches* and any *partialPairs* containing *Non-ProperMatch* events are established as *properMatches*.

Algorithm 4.3 takes one parameter, *EventList*. The parameter *EventList* is the list produced in Section 4.4.1, which contains a list of microservice events associated with their corresponding database events. In *line 3* the list *properMatches* is initialized, this list will contain the pairs of events that have been established as a *properMatch*. In *line 4* the list *partialMatches* is initialized, this list will contain the pairs of events that have been established as *partialMatches*.

In *lines 5-8* the algorithm iterates through the *EventList* and stores the timestamp of its current event ($event_a$). During each iteration the algorithm goes through the *EventList* again and obtains the timestamp of its current event ($event_b$). This step is used to compare all combinations of event pairs resulting from *EventList*, while ignoring pairs comprised of the same event. The first criterion is reached in *lines 9-10*, in which the two events must have timestamps that occur within the predefined time-frame (500milliseconds) of each other. If they pass the first criteria then the data values for each of the two events are stored.

In *lines 11-17* the algorithm evaluates the final requirement for being established as a *properMatch*. If the either of the event's values are a subset of the other event's values, then they are considered a *properMatch*. Otherwise they are considered a *partialMatch*.

Once all initial *properMatches* have been found, the algorithm will determine which *partialMatches* should be considered *properMatches*. In *lines 22-23*, two Boolean variables are initiated. The variables, ($partial_aMatch$) and ($partial_bMatch$) will keep track of whether or not the events in the *partialPair* have already been established as part of a *properPair*. This condition is evaluated in *lines 24-30*, where the algorithm iterates through all the *partialPairs* in *partialMatches*. During each iteration the algorithm iterates through each *properPair* in *properMatches*. If the first event in the current *[partialPair]* has already been established in a *properPair* then the Boolean variable (*partial$_a$Match*) is set to *True*. The same procedure is excuted with the second event in the current *partialPair* and the Boolean variable (*partial$_b$Match*). Once all the *properPairs* have been iterated through the algorithm determines if the *partialPair* should be established as a *properMatch*, *lines 33-36*. If both events or one of the two events were not already established as part of a *properPair* then the *partialPair* is now established as a *properPair*.

The reasoning for this logic is because we cannot make the assumption that two events must contain a subset of each others data in order to be associated with each other. In a distributed system with various databases connected to various system microservices, a receiving microservice may supplement the receiving data with other relevant data obtained elsewhere, which would result in a response event containing more than just the receiving data. Additionally, there exists a corner case in which a published event does not contain any data, but rather it is used as a trigger to execute specific data events from other microservices. An example of this a *DayHasPassed* event, of which contains no data other than its label. However there may exist receiving microservices that contain the business logic that implement some sort of data event upon the reception of the *DayHasPassed* event. In this scenario there would no common attribute values between the two, but rather the only matching property would be the proximity of timestamps. Considering both of these scenarios, it would be inaccurate to only establish two events as a match if and only if their values are a subset of each other. Hence, we provided the additional logic for establishing *partialMatches* as *properMatches*.

4.4.3 Automating the Attribute Synonym Identification Process

In FCA theory, a *formal concept* is defined to be a pair (*A, B*), where *A* is a set of objects (called the *extent*) and *B* is a set of attributes (the *intent*) such that: a) the *extent A* consists of all objects that share the attributes in *B*, and dually; b) the *intent B* consists of all attributes shared by the objects in *A*. This creates a lattice where the nodes denote objects with attributes. The top element of the lattice (i.e. the most general concept) contains all objects and their

Algorithm 4.3 MSA Event Association

```
- Let $EventList$ be the set of matched MSA and SQL events from AttributeSynonymSupplementation()
procedure matchEventPairs($EventList$)
    $properMatches$ = []
    $partialMatches$ = []
    for each $event_A$ in $EventList$ do
        $event_ATimestamp$ = $event_A$.getTimestamp()
        for each $event_B$ in $EventList$ do
            if $event_A \neq event_B$ then
                $event_BTimestamp$ = $event_B$.getTimestamp()
                if $event_BTimestamp$ is within the timeframe of $event_ATimestamp$ then
                    $event_AValues$ = set(getValues($event_A$))
                    $event_BValues$ = set(getValues($event_B$))
                    if $event_aValues \subseteq event_bValues$ OR $event_bValues \subseteq event_aValues$ then
                        $properMatches$.append([$event_A$, $event_B$])
                    else
                        $partialMatches$.append([$event_A$, $event_B$])
                    end if
                end if
            end if
        end for
    end for
    $partial_aMatch$ = False
    $partial_bMatch$ = False
    for each $partialPair$ in $partialMatches$ do
        for each $properPair$ in $properMatches$ do
            if $partialPair$[0] in $properPair$ then
                $partial_aMatch$ = True
            end if
            if $partialPair$[1] in $properPair$ then
                $partial_bMatch$ = True
            end if
        end for
        if !($partial_aMatch$ and $partial_bMatch$) then
            $properMatches$.append($partialPair$)
        end if
    end for
    return $properMatches$
end procedure
```

common features (if any), while the bottom element of the lattice (i.e. the most specialized concept) contains the all the objects containing all the features. In the lattice, an attribute *v* involves all objects at and above the node at which the attribute appears, while an object *A* is required for all attributes at and below the node at which the object appears. In our approach, the FCA objects are the different *event types* while the FCA features are the *attributes* of the event types.

An example lattice is depicted in Figures 3.9 and 3.10 adapted from [38], while Table 4.1 and Figure 4.8 depict the actual objects and attributes denoted by analyzing the schemas of the sample microservice system we have experimented with [60]. Table 4.1 depicts the FCA formal context, while Fig. 4.8 illustrates the corresponding lattice that is obtained from the formal context.

The purpose of implementing FCA is to initiate the automation process for event association. This type of analysis is important for identifying the nature of the mappings and helps on the formation of the rules (i.e. how many pairs to attribute pairs consider for each rule).

FCA Table Creation

The first step towards automating the process is generating the FCA table. However, as established in Section 4.4.1, the inconsistency between labeling formats is an issue. Hence we present Algorithm 4.4, which addresses the attribute synonym issue as well as creates the FCA Table.

Algorithm 4.4 utilizes the *Attribute Synonym* to create a unified FCA table with property columns representative of the various labeling conventions. The resulting FCA can be seen in Figure 4.1. Algorithm 4.4 takes two parameters, the first is *AttributeSynonyms* that was obtained in Section 4.4.1. The *AttributeSystem* for the *PitStop* system is shown in Figure 4.7. The second parameter is *EventList*, which is a list of all the logged events containing relevant information to data dependencies established in Section 4.1 and Section 4.2. In *lines 4-6* the algorithm creates a csv file writer named *tableFCA*, which will be used to write all the object and property data for the logged events. The file will be written using a list of lists, where each inner list contains an entry to each column on the table.

Next, the algorithm creates *tableColumns* which represent the property headers of the FCA table, the property headers consist of the keys from the *AttributeSynonym* since each key represents a unified synonym for various inconsistent attribute labeling formats. The *tableRows* is a list used for storing all the event data corresponding to the table. In *lines 7-10* the algorithm iterates through all the list of events. Additionally, it creates *row* which is a list that stores the data for the current event, and it creates *eventID* and *eventAttributes* which contain the event's ID and attributes respectively. In *lines 11-13* the algorithm iterates through each of the columns in *tableColumns* (the event properties). During each iteration it gets the synonyms for the current *column*(property) and finds the intersection between the returned synonyms set and the current events attribute set.

In *lines 14-17*, if the intersection is not empty, then it means the current column is included in the current events attributes in which case an *'X'* is appended to the *row*. Otherwise an empty value is appended to *row*. An *'X'* value indicates the event contains the property and an empty value indicates that the event does not contain this property. In *lines 20* the *row* list which represents the event's data, is added to the list of other rows in *tableRows*.

In *lines 22-23* the table columns and list of rows are written onto the *tableFCA* csv. This table will then be used for creating the FCA lattice.

FCA Lattice Rule Extraction

Using the FCA table previously created, we are now able to construct a lattice and query the various intensions that were established. The extracted intensions from the lattice formulate the *Rules* for which the probabilistic reasoning engine in Chapter 5 can determine associations between microservice events. The process of extracting the *Rules* from the FCA Lattice is illustrated in Algorithm 4.5. The algorithm consists of two parts, in the first part of the algorithm will get all the intensions between events of the same type of microservices. In the second part, the algorithm will obtain all the intensions between events from different types of microservices.

The algorithm takes three parameters, the first parameter *Lattice* is the lattice formed from the formal contexts obtained in the previous step discussed in Section 4.4.3. The second parameter is *Pairs* which is a list of all pair combinations of microservices, including pairs containing the same microservice (i.e. *MicroserviceA, MicroserviceA*). The last parameter is

Algorithm 4.4 FCA Table Creation Using Attribute Synonym

```
- Let AttributeSynony be a dictionary in which the key is the attribute identifier and the value is a list of synonymous attribute labels
- Let EventList be a list of all the MSA events
procedure CreateTableFCA(EventList, AttributeSynonyms)
    tableFCA = csv.write(tableFCA.csv)
    tableColumns = AttributeSynonyms.getKeys()
    tableRows = []
    for event in EventList do
        row = []
        eventID = event.getEventType()
        eventAttributes = set(event.getAttributes())
        for column in tableColumns do
            synonyms = set(AttributeSynonyms[column])
            $intersection = eventAttributes \cap synonyms$
            if $intersection \neq \emptyset$ then
                row.append(’X’)
            else
                row.append(”)
            end if
        end for
        tableRows.append(row)
    end for
    tableFCA.writerow(tableColumns)
    tableFCA.writerow(tableColumns)
end procedure
```

MicroserviceEvents, which is a dictionary data structure in which the keys correspond to a unique microservice and the values correspond to a list of all the events that occured within said microservice.

The first half of the algorithm beings with *lines 6-7* where we iterate through each pair of microservices from the *Pairs* list, and the list *Rules* is initialized. The *Rules* list will contain all the pairs of events that the algorithm determines to be a match. In the next part of the algorithm (see *lines 8-10*), the algorithm checks to determine whether both microservices in the pair are the same. If they are the same then a counter variable is initialized and an event list is retrieved from *MicroserviceEvents* that corresponds to the current microservice. Next, the algorithm iterates through all combinations of events in *eventList* (see *lines 11-13*). Within each combination of events, the algorithm obtains a list of common properties between the two events through querying the *intensions* in the lattice (see *lines 14*). If the query returns one or more values then the two events are added onto the *Rules* list (see *lines 15-16*). The rest of the algorithm accomplishes the same thing as the first half, with the only difference being that the events being compared belong to different microservices (see lines*lines 21-25*).

A sample output of the FCA Lattice Rule Extraction Algorithm for schema reconciliation is depicted below.

```
Microservice Pair: NotificationService and
WorkshopmanagementEventHandler

Event Pair: NotificationService:RegisterCustomer
and WorkshopmanagementEventHandler:
RegisterCustomer
Intensions: ('CustomerId', 'Name', 'TelephoneNumber')

Event Pair: NotificationService:RegisterCustomer
and WorkshopmanagementEventHandler:
RegisterVehicle
Intensions: ('CustomerId')

Event Pair: NotificationService:RegisterMaintenanceJob
```

Algorithm 4.5 FCA Lattice Rule Extraction

```
- Let $Lattice$ be the lattice representation of the events
- Let $Pairs$ be a set of all microservice pairs $\langle microservice - i, microservice - j \rangle$
- Let $MicroserviceEvents$ be a dictionary with the key representing a microservice and the value representing a list of all events for that microservice
- Let $Rules$ be a set of all event intensions tuples $\langle event - i, event - j, intensions \rangle$
procedure LatticeRuleExtraction($Lattice$, $Pairs$, $MicroserviceEvents$)
    $Rules$ = []
    for each $pair$ in $Pairs$ do
        if $pair_i$ == $pair_j$ then
            counter = 0
            $eventList$ = $MicroserviceEvents[pair_i]$
            for each $event_a$ in $eventList$ do
                counter += 1
                for each $event_b$ in $eventList$[counter:] do
                    $commonProperties$ = $Lattice$.intensions([$event_a$, $event_b$])
                    if len($commonProperties$) $\geq 1$ then
                        $Rules$.append([$event_a$, $event_b$, $commonProperties$])
                    end if
                end for
            end for
        else
            for each $event_a$ in $MicroserviceEvents[0]$ do
                for each $event_b$ in $MicroserviceEvents[1]$ do
                    $commonProperties$ = $Lattice$.intensions([$event_a$, $event_b$])
                    if len($commonProperties$) $\geq 1$ then
                        $Rules$.append([$event_a$, $event_b$, $commonProperties$])
                    end if
                end for
            end for
        end if
    end for
end procedure
```

Event Types	Attributes																
	CustomerId	Name	Address	PostalCode	City	Telephone	Email	License	Brand	Type	Date	JobId	StartTime	EndTime	Description	Notes	Id
Rabbitmq:CustomerRegistered	✓	✓	✓	✓	✓	✓	✓										
Rabbitmq:VehicleRegistered	✓							✓	✓	✓							
Rabbitmq:WorkshopPlanningCreated											✓						
Rabbitmq:MaintenanceJobPlanned	✓	✓				✓		✓	✓	✓		✓	✓	✓	✓		
Rabbitmq:DayHasPassed												✓	✓	✓			
Rabbitmq:MaintenanceJobFinished												✓	✓	✓		✓	
InvoiceService:RegisterCustomer	✓	✓	✓	✓	✓												
InvoiceService:RegisterMaintenanceJob	✓							✓				✓			✓		
InvoiceService:FinishMaintenanceJob												✓	✓	✓			
InvoiceService:InvoiceSentToCustomer		✓															✓
NotificationService:RegisterCustomer	✓	✓				✓	✓										
NotificationService:RegisterMaintenanceJob	✓							✓				✓	✓		✓		
NotificationService:SentNotification		✓															
NotificationService:NotificationMailSent							✓										
NotificationService:RemoveFinishedMaintenanceJob												✓					
WorkshopManagementEventHandleer:RegisterCustomer	✓	✓				✓											
WorkshopManagementEventHandler:RegisterVehicle	✓							✓	✓	✓							
WorkshopManagementEventHandler:RegisterMaintenanceJob		✓						✓				✓	✓	✓			
WorkshopManagementEventHandler:FinishMaintenanceJob												✓	✓	✓			

Table 4.1: Associated attributes on event logs

```
and WorkshopmanagementEventHandler:RegisterVehicle
Intensions: ('CustomerId', 'LicenseNumber')

Event Pair: NotificationService:RegisterMaintenanceJob
and WorkshopmanagementEventHandler:
RegisterMaintenanceJob
Intensions: ('LicenseNumber', 'JobId', 'StartTime')

Event Pair: NotificationService:RegisterMaintenanceJob
and WorkshopmanagementEventHandler:FinishMaintenanceJob
Intensions: ('JobId', 'StartTime')
```

In the sample output above, we see the different event types and the shared attributes. The structured example indicates the microservice pairs that are being examined (in this case *NotificationService* and *WorkshopmanagementEventHandler*), and a list of five event pairs along with their *intensions* derived from the lattice using Algorithm 4.5. The first event pair shown in the example, is between the event *NotificationService:RegisterCustomer* and *WorkshopmanagementEventHandler*, with the derived intensions *'CustomerId', 'Name'* and *'TelephoneNumber'*. These intensions can be verified by looking at the attribute data in Table 4.1, in which the listened intensions are all properties for each of the two event types.

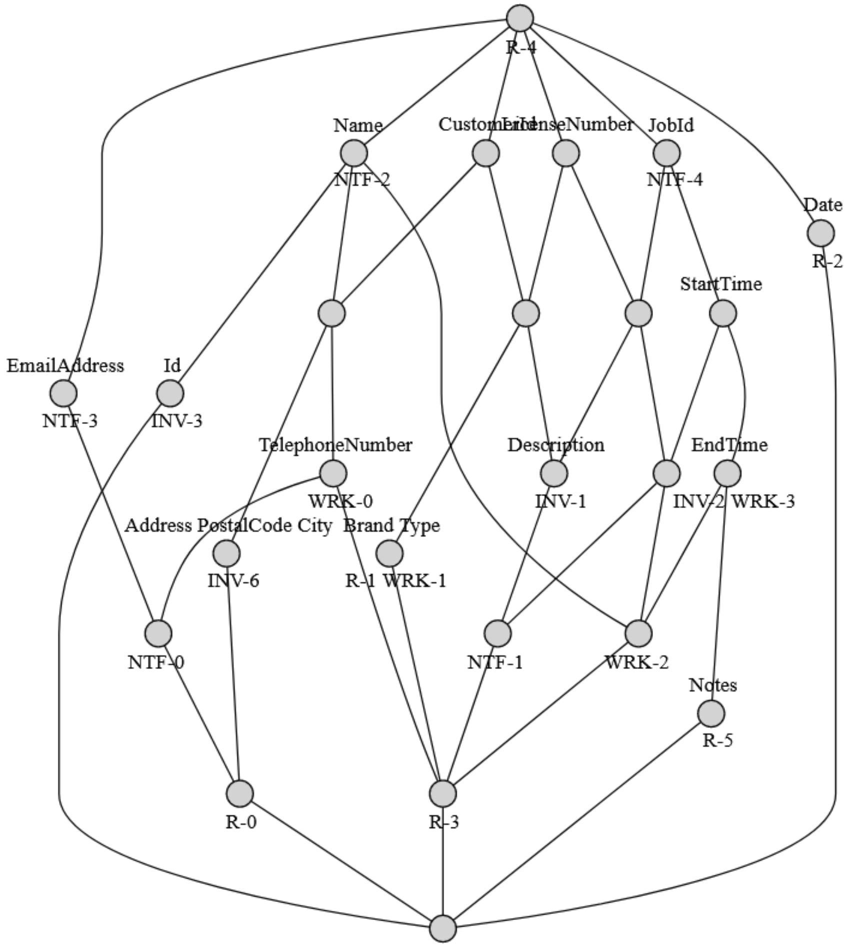


Figure 4.8: The extracted FCA lattice for the log data set

4.5 Summary

This chapter focused on the schema reconciliation and conceptual event association aspects of our proposed framework for generating a Microservice Dependency Graph. The initial part of the chapter focused on filtering the raw data logs in order to reduce the computational costs, as described in Section 4.1. The filtering process is required given the vastness of logging that occurs throughout a microservice based system, most of which is not relevant to the establishment of data dependencies. Next, cross-referencing was used to match SQL events with their corresponding source Microservice names as described in Section 4.2. This step was required due to the incompleteness of the logs which resulted in missing relational data. Specifically, SQL events contained creation, modification and deletion information however it does not contain any data regarding the microservice that invoked this action. Therefore, the SQL events need to be cross-referenced and associated with their corresponding source microservice. Afterwards cross-referencing was used once again except this time, SQL events were matched with their corresponding source Microservice events as described in Section 4.3. This step was required in order to progress towards data completion capable of representing the system. In order to begin to establish the flow of data dependencies throughout the system, the first step is establishing the dependencies between microservice events and SQL events.

Those three sections constituted the schema reconciliation portion of the chapter. The second portion of the chapter focused on initial development for event association automation. Section 4.4 was composed of three subsections. The first subsection was Section 4.4.1, in which synchronization between attribute synonyms was conducted in order to establish consistency throughout all the labeled attributes in the log data. This step is required due to labeling inconsistencies that resulted in incompleteness in the schema reconciliation, as well the attribute synonyms established will be used in the automation process. The next subsection, Section 4.4.2, discussed a conceptual methodology for establishing associations between microservice events. This subsection provided the initial ground work required for developing an automated process capable of establishing the event associations. The final subsection Section 4.4.3, discussed the implementtaon of the first step towards the automation of the event association process. This first step is based on FCA in order to represent the Microservice event log data in a concept hierarchy, which can then be used in a second step by a probabilistic reasoning engine as it wil lbe discussed in Chapter 5.

Chapter 5

System-Wide Event Matching

5.1 Final Matching of System-wide Events

Once the system-wide events (i.e. events between microsevices and between microservices and the database) have been *cross-referenced*, then the objective shifts on defining that two events *match* (i.e. associate). The event matching process is based on a probabilistic reasoning engine which aims to identify pairs of events that *match*.

We say that two events *match* if they contain common attributes, each of the attributes have the same value, and the events have occurred within a pre-specified time window. These events are said to be *matched* and are part of a transaction event chain. This reasoning process deducing whether two events match is based on three main elements. The first element is a set of *facts* that are extracted from the logs. The second element is a set of event matching rules. The third element is a probabilistic reasoning engine that utilizes Markov Logic Networks.

5.1.1 Fact Base

For the rule-based event association technique, we have identified five types of *fact* predicates as follows:

- *feature(attribute,event)*: Indicates that *event* has *attribute*
- *inOrder(event1,event2)*: Indicates that *event1* occurs (i.e. has timestamp) within the same timeframe (i.e. +/- *x* milliseconds) with *event2*

- *featureMatch(attribute1,attribute2,event1,event2)*: Indicates that *attribute1* in *event1* is a synonym with *attribute2* in *event2*
- *sameValue(attribute1,attribute2,event1,event2)*: Indicates that *attribute1* in *event1* has the same value as *attribute2* in *event2*
- *match(event1,event2)*: Indicates that *event1* (i.e. is related with *event2* in the same transaction chain.

Fact Extraction

The *feature()* predicate facts were extracted from the event logs of each event pair that were established in Chapter 4. The event logs were iterated through and a *feature()* predicate fact was generated for every individual attribute of the events that occurred in the logs. Therefore for an event with 'x' amount of attributes, it would result in 'x' amount of *feature()* predicate facts with each containing one of the 'x' attributes. The *feature()* fact consists of two parts, the name of the attribute and the second is an event identifier that represents the source microservice and the event type. The generated predicate facts are in the format presented below:

```
feature(attribute, event)
```

A similar process is required for the extraction of the *inOrder()* predicate facts. In this process all the event logs of each event pair from Chapter 4 are cross-referenced with each and compared based on their timestamp attribute. In the instance where two events occurring within a predetermined timeframe (+/-0.5 seconds) a predicate fact is generated. The *inOrder()* predicate fact consists of two event identifiers. The predicate format presented below:

```
inOrder(event1, event2)
```

The *featureMatch()* predicate facts are extracted from the Formal Concept Analysis that was discussed in Section 4.4.3. By applying Formal Concept Analysis, all possible pair combinations of the events are used to query the formal context to find the list of *intensions* between the two events. Using the list of event pair *intensions* a predicate fact is generated for each of the attributes in the list. The *featureMatch()* predicate fact generated consists of two event identifiers each representing a different event, and two attribute identifiers each representing each event's attribute. The format of the fact is presented below:

```
featureMatch(attribute1,attribute2,event1,event2)
```

The *sameValue()* and *match()* predicate facts are generated through a more detailed process. The pseudo-code explaining the algorithm can be found in Algorithm 5.1. The predicate fact *sameValue()* represents two events that contain the same attribute and their values are consistent with one another. The predicate fact *match()* represents two events that comply with a rule derived from the FCA lattice as discussed in Section 4.4.3. The format of the two facts is presented below:

```
sameValue(attribute1, attribute2, event1, event2)
match(event1, event2)
```

Algorithm 5.1, takes two parameters. The first parameter *eventList* is a set of all event pairs derived from the event list established in Chapter 4. The second parameter is *ruleBase*, which denotes a dictionary based on the *FCA Lattice Rule Extraction* (see Algorithms 4.5 and 4.4) where the keys are a pair of *microservices* and the values are a list of *intensions*. The data used for populating the *ruleBase* was obtained through the *FCA* implementation in Section 4.4.3.

The first part of Algorithm 5.1, (see *lines 6-7*), iterates through all event pairs in *eventList* and checks to see that the event pair does not consist of a single duplicate event. Next, the algorithm retrieves the list of intensions from *ruleBase* using the microserivce of each event as key, and then proceeds to extract the attributes from each of the events (see *lines 8-10*). The rules correspond to the FCA *intensions* between events of the two types of microservices, as described in Section 4.4.3.

Next, in *lines 11-13*, the algorithm iterates through each of the rules corresponding to the two current event microservices. In each iteration the algorithm also goes through each attribute within the current rule and initializes the Boolean variable *matchFound*. If the current attribute is not found inside either the list of attributes of *eventA* or *eventB*, then the Boolean variable previously established is set to *false*, as seen in *lines 14-16*. On the other hand, if the attribute is found in both *eventA* and *eventB*, then an external function is called to determine if the attribute value for each event are equal. If the attribute values are equal, then a fact of type *sameValue(attribute1, attribute2, event1, event2)* is created and stored. Otherwise, when the attribute values are not equal with each other, then the Boolean variable *matchFound* is set to *false* as seen in *lines 16-22*.

Algorithm 5.1 featureMatch and match fact extraction

```
– Let $eventList$ be the set of all events pairs from the event list derived in previous phases
– Let $ruleBase$ be a dictionary with keys being of type $\langle microservice_a, microservice_b \rangle$ and values being a list of $\langle intensions \rangle$
- Let $sameValueList$ be a list of sameValue facts
- Let $matchList$ be a list of match facts
procedure FACTEXTRACTION($eventList$, $ruleBase$)
    for each ($event_a$,$event_b$) in $eventList$ do
        if $event_a \neq event_b$ then
            $rules = ruleBase$ [($event_a$, $event_b$)]
            $event_aAttributes = event_a$.getAttributes()
            $event_bAttributes = event_b$.getAttributes()
            for each $rule$ in $rules$ do
                $matchFound$ = True
                for each $attribute$ in $rule$ do
                    if $attribute$ not in $event_aAttributes$ OR $event_bAttribute$ then
                        $matchFound$ = False
                    end if
                    if $attribute$ in $event_aAttributes$ AND $event_bAttribute$ then
                        $sameValue$ = sameValue($event_a$[$attribute$], $event_b$[$attribute$])
                        if $sameValue$ == True then
                            $sameValueList$.append(sameValue($attribute$, $attribute$, $event_a$, $event_b$))
                        else
                            $matchFound$ = False
                        end if
                    end if
                end for
                if $matchFound$ == True then
                    $matchList$.append(match($event_a$, $event_b$))
                end if
            end for
        end if
    end for
end procedure
```

At the end of the loop iterating through the attributes, if the Boolean variable *matchFound* is *true*, then this means the each attribute in the current FCA rule was found inside each of the two events and the attribute values were consistent (i.e. equal). In this case a fact of type *match(event, event)* is created and stored. Otherwise, if the Boolean variable, *matchFound* is *false*, then there was an instance in which a pair of attributes had different values, in which case no fact is generated. This is shown in *lines 26-27*.

Fact Examples

These are the types of facts populate a fact-base which will be used for training rules and deducing which event matches with which another event. An excerpt of the fact-base is presented below:

```
feature(CustomerId, InvoiceService:RegisterCustomer)
feature(Name, InvoiceService:RegisterCustomer)
feature(Address, InvoiceService:RegisterCustomer)
feature(PostalCode, InvoiceService:RegisterCustomer)
feature(City, InvoiceService:RegisterCustomer)

feature(CustomerId, NotificationService:RegisterCustomer)
feature(Name, NotificationService:RegisterCustomer)
feature(Telephone, NotificationService:RegisterCustomer)
feature(Email, NotificationService:RegisterCustomer)

featureMatch(CustomerId, CustomerId,
InvoiceService:RegisterCustomer,
NotificationService:RegisterCustomer)
featureMatch(Name, Name,
InvoiceService:RegisterCustomer,
NotificationService:RegisterCustomer)

inOrder(InvoiceService:RegisterCustomer,
NotificationService:RegisterCustomer)

sameValue(CustomerId, CustomerId,
InvoiceService:RegisterCustomer,
NotificationService:RegisterCustomer)
sameValue(Name, Name,
InvoiceService:RegisterCustomer,
NotificationService:RegisterCustomer)

match(InvoiceService:RegisterCustomer,
NotificationService:RegisterCustomer)
```

The example above is a result from the log traversal of the two microservice logs, *Invoice* and *Notification*, that correspond to the events *InvoiceService:RegisterCustomer* and *NotificationService:RegisterCustomer*. The example predicate facts depicted above can be categorized into the following six sets:

- First set contains the *feature(attribute, event)* predicate facts for *InvoiceService*
- Second set contains the *feature(attribute, event)* predicate facts for *NotificationService*
- Third set contains the *featureMatch(attribute1, attribute2, event1, event2)* predicate facts for *InvoiceService* and *NotificationService*
- Fourth set contains the *inOrder(event1, event2)* predicate facts for *InvoiceService* and *NotificationService*
- Fifth set contains the *sameValue(attribute1, attribute2, event1, event2)* predicate facts for *InvoiceService* and *NotificationService*
- Sixth set contains the *match(event1, event2)* predicate facts for *InvoiceService* and *NotificationService*.

The first set contains five predicate facts which were extracted from the *Invoice* event, in which a *feature(attribute, event)* predicate fact is generated for each of the attributes within the *Invoice* event. The second set contains four predicate facts that are similar to the previous set, except these predicate facts correspond to the *Notification* event. The third set contains the *featureMatch(attribute1, attribute2, event1, event2)* predicate facts for each attribute synonym match between the two events, the *Invoice* event and the*Notification* event. The fourth set contains the predicate facts of type *inOrder(event1, event2)* between the two *Invoice* and *Notification* events. The fifth set of predicate facts are the *sameValue(attribute1, attribute2, event1, event2)*, these predicate facts represent the existence of two common attributes between the events which when compared have consistent values. The final set of predicate facts is the *match(event1, event2)*. This predicate fact represents the existence of a two events that have been established to be associated with each other. Two events are established as being a match based on the existence of the previous predicate facts, *feature(attribute, event), featureMatch(attribute1, attribute2, event1, event2), inOrder(event1, event2), sameValue(attribute1, attribute2, event1, event2)*, according to an individual rule derived from the FCA Lattice in Chapter 4.

5.1.2 Rule Base

The second element in the event association process is a set of rules that aim to denote the domain logic whether two events relate to the same chain in a given transaction sequence. The rules conclude the predicate *match(event1,event2)*. In a nutshell, the rules encode the logic that:

> *"if two events have attributes that are reconciled in the FCA phase of the process, and if these attributes in the two events have the same values pairwise, and if the events have occurred in the same approximate time, then the two events belong to the same chain"*.

In this context, two questions arise. The first question relates to what types of attributes any two events should be considered on. The second question relates to how many attributes any two events should be considered for event matching purposes.

The answer to the first question lies on the *feature(attribute, event)* predicate that denotes that *event* has *attribute* and the *featureMatch(attribute1, attribute2, event1, event2)* predicate that denotes that *attribute1* and *attribute2* are synonyms in *event1* and *event2*. The answer to the second question lies on the results obtained from the FCA analysis. For example, as seen in Table 4.1 event types *Rabbitmq:MaintenanceJobPlanned*, *Rabbitmq:DayHasPassed* and, *Rab-*

bitmq:MaintenanceJobFinished share three attributes, namely *Job*, *StartTime* and *EndTime*, while event types *InvoiceService:Register Customer* and *NotificationService:RegisterCustomer* share two attributes namely *Customer* and *Name*. In this respect, we devise rules that consider one pair, two pairs, and three pairs of attributes to be matched in the rules.

A sample rule-set is provided below.

For one pair the rule is:

```
feature(a1,e1) ^ feature(a2,e2) ^
featureMatch(a1,a2,e1,e2) ^ sameValue(a1,a2,e1,e2) ^
inOrder(e1,e2) => match(e1,e2)
```

For two pairs the applicable rule is:

```
feature(a3,e3) ^ feature(a4,e4) ^
featureMatch(a3,a4,e3,e4) ^
sameValue(a3,a4,e3,e4) ^ feature(a5,e3) ^
feature(a6,e4) ^ featureMatch(a5,a6,e3,e4) ^
sameValue(a5,a6,e3,e4) ^ inOrder(e3,e4) =>
match(e3,e4)
```

For events with three pairs of reconciled attributes the applicable rule is:

```
feature(a7,e5) ^ feature(a8,e6) ^
featureMatch(a7,a8,e5,e6) ^
sameValue(a7,a8,e5,e6) ^ feature(a9,e5) ^
feature(a10,e6) ^ featureMatch(a9,a10,e5,e6) ^
sameValue(a9,a10,e5,e6) ^ feature(a11,e5) ^
feature(a12,e6) ^ featureMatch(a11,a12,e5,e6) ^
sameValue(a11,a12,e5,e6) ^ inOrder(e5,e6) =>
match(e5,e6)
```

These three rules denote the logic presented above, that is if the attributes are reconciled, and their values are the same, and the events have occurred within a predefined time window, then the two events are associated (i.e. they match).

Rule Extraction

The rules shown above are a general representation that encompass the rules derived from Algorithm 4.5.

The *One Pair Rule* is product of a general representation of FCA Lattice rules. An example of this is between the events *NotificationService:RemoveFinishedMaintenanceJob* and *WorkshopmanagementEventHandler:FinishMaintenanceJob*. From the FCA Lattice rule extraction in Algorithm 4.5 we discover the intension of *'JobId'* between the two event types. In combination with the predicate facts generated in Section 5.1.1 we are able to establish the *One Pair Rule*. An example outline for the FCA Lattice rule and its corresponding predicate facts are shown below:

```
Event Pair: NotificationService:RemoveFinishedMaintenanceJob
and WorkshopmanagementEventHandler:FinishMaintenanceJob
Intensions: ('JobId')

feature('JobId', NotificationService:RemoveFinishedMaintenanceJob)
feature('JobId', WorkshopmanagementEventHandler:FinishMaintenanceJob)

featureMatch('JobId', 'JobId',
NotificationService:RemoveFinishedMaintenanceJob,
WorkshopmanagementEventHandler:FinishMaintenanceJob)

inOrder(NotificationService:RemoveFinishedMaintenanceJob,
WorkshopmanagementEventHandler:FinishMaintenanceJob)

sameValue('JobId', 'JobId',
NotificationService:RemoveFinishedMaintenanceJob,
WorkshopmanagementEventHandler:FinishMaintenanceJob)

match(NotificationService:RemoveFinishedMaintenanceJob,
WorkshopmanagementEventHandler:FinishMaintenanceJob)
```

The construction of the *Two Pair Rule* is similar to that of the previous rule. An example of this occurrence can be seen between the events *InvoiceService:RegisterCustomer* and

NotificationService:RegisterCustomer. From the FCA Lattice rule extraction in Algorithm 4.5 we once again discover the existence of the intension containing *'CustomerId'* and *'Name'*. Supplemented with the facts generated in Section 5.1.1 we establish the *Two Pair Rule*. An example outlining the FCA Lattice rule and its corresponding facts are shown below:

```
Event Pair: InvoiceService:RegisterCustomer
and NotificationService:RegisterCustomer
Intensions:('CustomerId', 'Name')

feature('CustomerId', InvoiceService:RegisterCustomer)
feature('Name', InvoiceService:RegisterCustomer)
feature('CustomerId', NotificationService:RegisterCustomer)
feature('Name', NotificationService:RegisterCustomer)

featureMatch('CustomerId', 'CustomerId',
InvoiceService:RegisterCustomer,
NotificationService:RegisterCustomer)
featureMatch('Name', 'Name',
InvoiceService:RegisterCustomer,
NotificationService:RegisterCustomer)

inOrder(InvoiceService:RegisterCustomer,
NotificationService:RegisterCustomer)

sameValue('CustomerId', 'CustomerId',
InvoiceService:RegisterCustomer,
NotificationService:RegisterCustomer)
sameValue('Name', 'Name',
InvoiceService:RegisterCustomer,
NotificationService:RegisterCustomer)

match(InvoiceService:RegisterCustomer,
NotificationService:RegisterCustomer)
```

The construction of the final rule, the *Three Pair Rule*, is based on the same process as the previous two rules. For this rule the example is between the events *InvoiceService:FinishMaintanenceJob* and *WorkshopMangementEventHandler:FinishMaintenanceJob*. Based on the FCA Lattice rule extraction in Algorithm 4.5 we derive the intension containing *’JobId’, ’StartTime’* and *’EndTime’*. Accompanied with the predicate facts generated in Section 5.1.1 we establish the *Three Pair Rule*. An example of the FCA Lattice rule and the event Facts is shown below:

```
Event Pair: InvoiceService:FinishMaintanenceJob
and WorkshopMangementEventHandler:FinishMaintenanceJob
Intensions:(’JobId’, ’StartTime’, ’EndTime’)

feature(’JobId’, InvoiceService:FinishMaintanenceJob)
feature(’StartTime’, InvoiceService:FinishMaintanenceJob)
feature(’EndTime’, InvoiceService:FinishMaintanenceJob)
feature(’JobId’, WorkshopMangementEventHandler:FinishMaintenanceJob)
feature(’StartTime’, WorkshopMangementEventHandler:FinishMaintenanceJob)
feature(’EndTime’, WorkshopMangementEventHandler:FinishMaintenanceJob)

featureMatch(’JobId’, ’JobId’
InvoiceService:FinishMaintanenceJob,
WorkshopMangementEventHandler:FinishMaintenanceJob)
featureMatch(’StartTime’, ’StartTime’
InvoiceService:FinishMaintanenceJob,
WorkshopMangementEventHandler:FinishMaintenanceJob)
featureMatch(’EndTime’, ’EndTime’
InvoiceService:FinishMaintanenceJob,
WorkshopMangementEventHandler:FinishMaintenanceJob)

inOrder(InvoiceService:FinishMaintanenceJob,
WorkshopMangementEventHandler:FinishMaintenanceJob)

sameValue(’JobId’, ’JobId’,
InvoiceService:FinishMaintanenceJob,
WorkshopMangementEventHandler:FinishMaintenanceJob)
sameValue(’StartTime’, ’StartTime’,
InvoiceService:FinishMaintanenceJob,
WorkshopMangementEventHandler:FinishMaintenanceJob)
```

```
sameValue('EndTime', 'EndTime',
InvoiceService:FinishMaintanenceJob,
WorkshopMangementEventHandler:FinishMaintenanceJob)

match(InvoiceService:FinishMaintanenceJob,
WorkshopMangementEventHandler:FinishMaintenanceJob)
```

5.1.3 Reasoning

The third element of the event association process is the reasoning engine. We opted for a probabilistic reasoning framework for two reasons.

The first reason has to do with incomplete data. In First Order Logic if one or more of the premises of a Horn Clause is not satisfied the whole rule fails. In the case of software systems, some logs may be inaccessible, or not emitting the events requited by a specific rule. In that case we would like to still be able to reason and deduce with a reduced level of confidence that two events match.

The second reason has to do with completeness of the rule-set itself. Engineers may not be able to model with one rule-set all possible scenarios of two events matching. The more specific the rules become the fewer systems they will be applicable to, and the more general they become the less precision occurs in the results (i.e. more events are considered as being matched).

In this paper we utilize a probabilistic reasoning engine that is based on Markov Logic and Markov Logic Networks.

Markov Logic and Markov Logic Networks

Markov Logic combines statistical and logic-based approaches. Rules are denoted as sets of Horn Clauses mapped into a Conjunctive Normal Form. Rules are annotated with weights signifying the importance of each rule. Rules constitute a rule-base while ground predicates (predicates with ground atom values – i.e. facts) constitute the fact-base. The rule-base and the fact-base create what is referred to as the Markov Logic Network. In simplified terms, a Markov Logic Network is a graph where the nodes are ground predicates and edges link

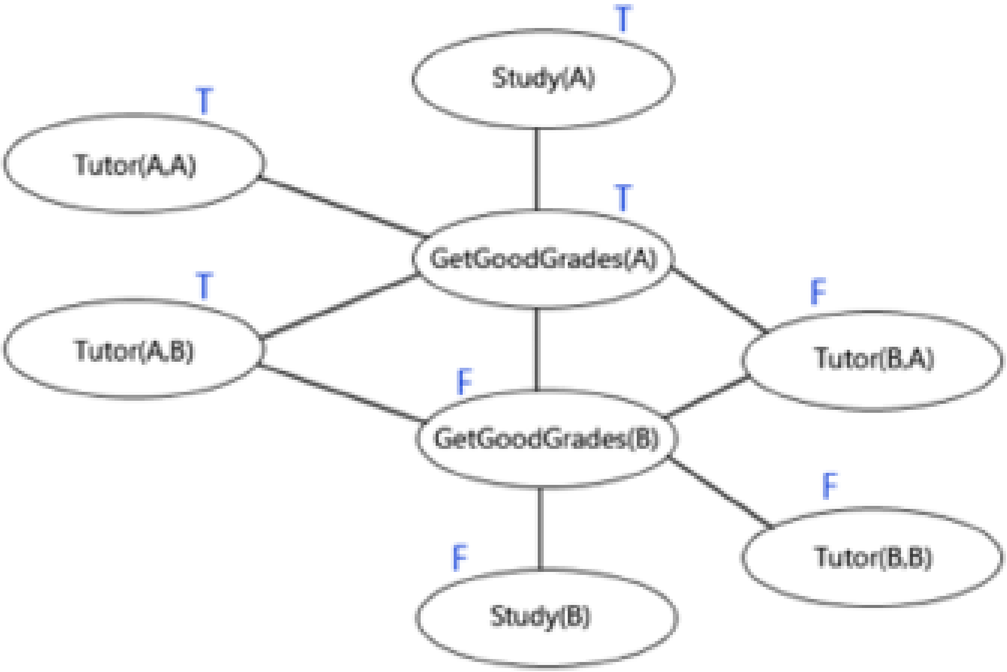
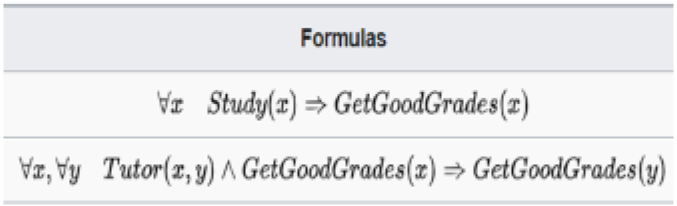


Figure 5.1: An example Markov Logic Network [48], [10]



Formulas
$\forall x \quad Study(x) \Rightarrow GetGoodGrades(x)$
$\forall x, \forall y \quad Tutor(x,y) \wedge GetGoodGrades(x) \Rightarrow GetGoodGrades(y)$

Figure 5.2: An example Markov Logic Network [48], [10]

ground predicates if these appear in a rule. The analysis of the Markov Logic Network assigns a probability value to a world (i.e. a collection of grounded facts and rules) by a formula of the form depicted in equation 5.1.3. An example of Markov Logic Network adapted from [10] is depicted in Figures 5.1 and 5.2, where the nodes are facts and the edges denote that two predicates appear in the same rule. The truth vales (T or F) associated with each fact denote the observation as to whether this fact is proven true of false during the fact acquisition phase.

The probability of a world *pw* is given by a formula of the form:

$$P(pw) \propto exp\left(\sum \text{weights of formulas it satisies}\right) \tag{5.1}$$

Training and Inference

For our work we have obtained the logs from the middleware service bus, the SQL data base server, and the microservices. From all these logs we have obtained the 50% of the logs for training the rules (i.e. assigning weights to the rules), and we kept the other 50% for testing. The testing focuses first on evaluating the accuracy of the obtained results, that is whether two events that have been identified as matches (i.e. the predicate *match(e1, e2)* has probability more than 90%) are indeed associated, and second whether there are any events that are *unmatched* (i.e. do not associate with any other event).

As discussed above, we have used Alchemy [1] for encoding the facts, the rules, and performing reasoning. As discussed above, Alchemy allows for assigning weights to rules using training, or assigning weights manually to rules, based on how important the analysts belied a rule is more important or less important than other rules in the rule-base.

We have experimented with both weight assignment techniques (automatic using training and manual) with comparable results.

It is worth noting the change in event labeling. Previously we had referenced events in the format *(MicroserviceName:EventType)*, this type of labeling was used to document quick information regarding the event. However in order to properly train the MLN, the various instances of the same event type occurring but with different data values must have unique events identifiers. For example two instances of the event *InvoiceService:RegisterCustomer* for two different users, cannot be labelled the same since they correspond to different transaction flow of data. Therefore a unique event identifier is required for all events. In order to provide the MLN with unique event IDs we formatted the events using the format *(MicroserviceName-EventCount)*. An example of the changed labeling format is shown below:

```
InvoiceService:RegisterCustomer -> InvoiceEvent-0
NotificationService:RegisterCustomer -> NotificationEvent-8
WorkshopMangementEventHandler:FinishMaintenanceJob -> WorkshopEvent-9
```

An excerpt of the results obtained by exercising the rules deducing the *match(e1, e2)* facts along with their probability scores is provided below.

```
match(InvoiceEvent-5,InvoiceEvent-5) 0.172033
match(InvoiceEvent-5,InvoiceEvent-6) 0.175032
match(InvoiceEvent-5,InvoiceEvent-7) 0.187031
match(InvoiceEvent-5,InvoiceEvent-8) 0.179032
match(InvoiceEvent-5,InvoiceEvent-9) 0.20303
match(InvoiceEvent-5,NotificationEvent-8) 0.99995
match(InvoiceEvent-5,NotificationEvent-9) 0.151035
match(InvoiceEvent-5,NotificationEvent-10) 0.155034
match(InvoiceEvent-5,NotificationEvent-11) 0.182032
match(InvoiceEvent-5,NotificationEvent-12) 0.179032
match(InvoiceEvent-5,NotificationEvent-13) 0.190031
match(InvoiceEvent-5,NotificationEvent-14) 0.187031
match(InvoiceEvent-5,NotificationEvent-15) 0.176032
match(InvoiceEvent-5,RabbitEvent-8) 0.99995
match(InvoiceEvent-5,RabbitEvent-9) 0.185031
match(InvoiceEvent-5,RabbitEvent-14) 0.193031
match(InvoiceEvent-5,WorkshopEvent-5) 0.99995
match(InvoiceEvent-5,WorkshopEvent-6) 0.19703
```

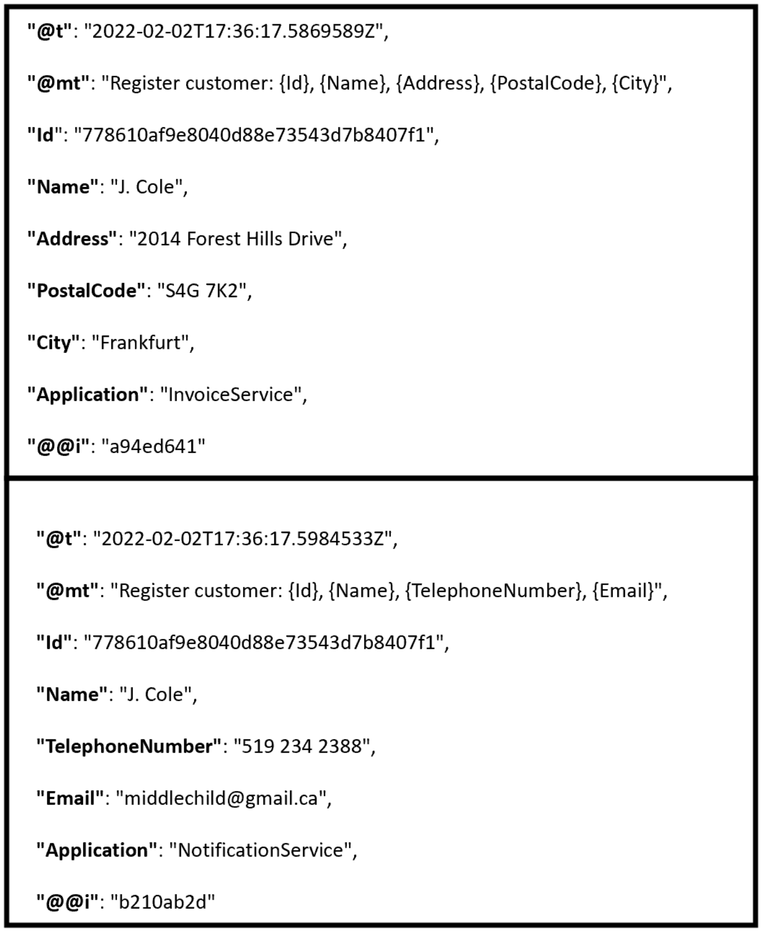
Inference Result Analysis

From the results shown above we can examine the two event pairs that achieved a probability of over 90%. The event logs for the pair of events, *InvoiceEvent-5* and *NotificationEvent-8*, is shown in Figure 5.3. A breakdown of the predicate facts generated by the first high probability event pair and the *Two Pair* rule from Section 5.1.2 is presented below:

```
Two Pair Rule :
feature(a3,e3) ^ feature(a4,e4) ^
featureMatch(a3,a4,e3,e4) ^
sameValue(a3,a4,e3,e4) ^ feature(a5,e3) ^
feature(a6,e4) ^ featureMatch(a5,a6,e3,e4) ^
sameValue(a5,a6,e3,e4) ^ inOrder(e3,e4) =>
match(e3,e4)

Event Pair:
(InvoiceEvent-5, NotificationService-8)
```

```
Facts:
feature(’Id’, InvoiceEvent-5)
feature(’Name’, InvoiceEvent-5)
feature(’Address’, InvoiceEvent-5)
feature(’PostalCode’, InvoiceEvent-5)
feature(’City’, InvoiceEvent-5)
feature(’Id’, NotificationService-8)
feature(’Name’, NotificationService-8)
feature(’TelephoneNumber’, NotificationService-8)
feature(’City’, NotificationService-8)

featureMatch(’Id’, ’Id’,
InvoiceEvent-5,NotificationService-8)
featureMatch(’Name’, ’Name’,
InvoiceEvent-5,NotificationService-8)
featureMatch(’City’, ’City’,
InvoiceEvent-5,NotificationService-8)

inOrder(InvoiceEvent-5, NotificationService-8)

sameValue(’Id’, ’Id’,
InvoiceEvent-5, NotificationService-8)
sameValue(’Name’, ’Name’,
InvoiceEvent-5, NotificationService-8)
```

As shown in the example above, the event pair produces nine *feature(attribute, event)* predicate facts each representing one of the event’s attributes. The event pair also produce three *featureMatch(attribute1, attribute2, event1, event2)* predicate facts each representing the attribute synonyms between the event pair. Additionally the event pair produced one *inOrder(event1, event2)* fact. The timestamps for each event can be seen in Figure 5.3 and based on the timestamps, the two events occurred just over ten milliseconds of each other. The final type of fact produced is the *sameValue(attribute1, attribute2, event1, event2)* fact for the attributes *Id* and *Name*. As shown in Figure 5.3, in which the *Id* attribute for both events are consist with a value



```
"@t": "2022-02-02T17:36:17.5869589Z",

"@mt": "Register customer: {Id}, {Name}, {Address}, {PostalCode}, {City}",

"Id": "778610af9e8040d88e73543d7b8407f1",

"Name": "J. Cole",

"Address": "2014 Forest Hills Drive",

"PostalCode": "S4G 7K2",

"City": "Frankfurt",

"Application": "InvoiceService",

"@@i": "a94ed641"
```

```
"@t": "2022-02-02T17:36:17.5984533Z",

"@mt": "Register customer: {Id}, {Name}, {TelephoneNumber}, {Email}",

"Id": "778610af9e8040d88e73543d7b8407f1",

"Name": "J. Cole",

"TelephoneNumber": "519 234 2388",

"Email": "middlechild@gmail.ca",

"Application": "NotificationService",

"@@i": "b210ab2d"
```

Figure 5.3: Log Breakdown for the event pair InvoiceEvent-5 (Top) and NotificationEvent-8 (Bottom)

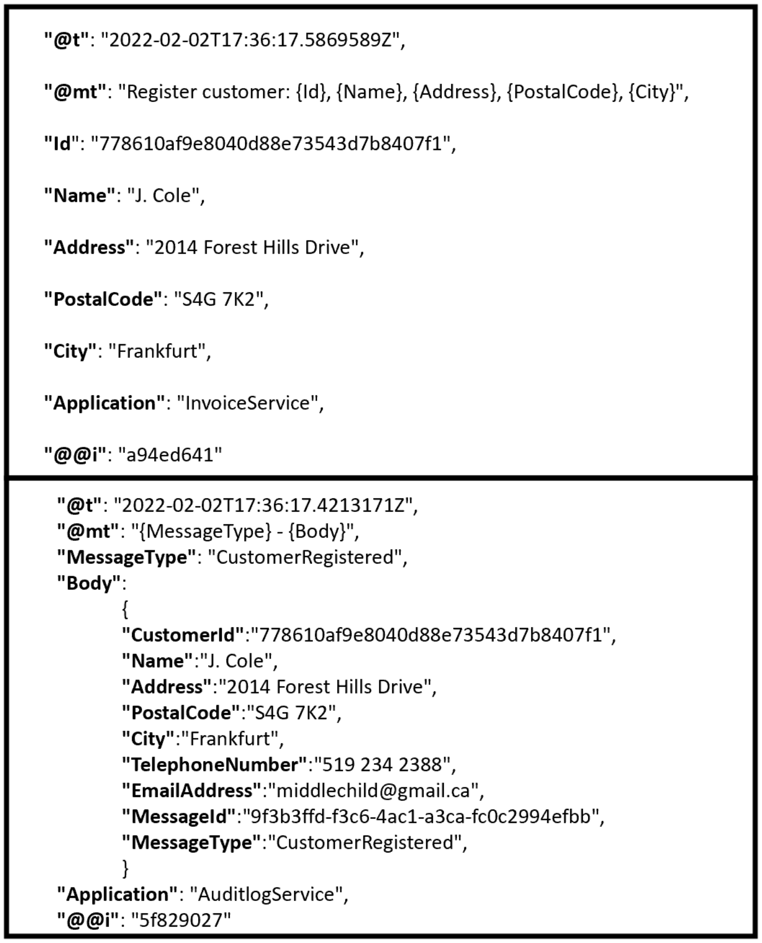
of *'778610af9e8040d88e73543d7b8407f1'* and the *Name* attribute for both events are also consist with a value of *'J. Cole'*. Therefore the fact *match(InvoiceEvent-5,NotificationEvent-8)* can is correctly inferred based on the *Two Pair* rule established in Section 5.1.2.

The second example from the MLN inferences, infer the highly probable association between events *InvoiceEvent-5* and event *RabbitEvent-8*. The event logs for both events are shown in Figure 5.4. A breakdown of the predicate facts generated by the two events alongside the *Three Pair* rule from Section 5.1.2 is presented below:

```
Three Pair Rule:
feature(a7,e5) ^ feature(a8,e6) ^
featureMatch(a7,a8,e5,e6) ^
sameValue(a7,a8,e5,e6) ^ feature(a9,e5) ^
```

```
feature(a10,e6) ^ featureMatch(a9,a10,e5,e6) ^
sameValue(a9,a10,e5,e6) ^ feature(a11,e5) ^
feature(a12,e6) ^ featureMatch(a11,a12,e5,e6) ^
sameValue(a11,a12,e5,e6) ^ inOrder(e5,e6) =>
match(e5,e6)

Event Pair:
(InvoiceEvent-5, RabbitEvent-8)

Facts:
feature(’Id’, InvoiceEvent-5)
feature(’Name’, InvoiceEvent-5)
feature(’Address’, InvoiceEvent-5)
feature(’PostalCode’, InvoiceEvent-5)
feature(’City’, InvoiceEvent-5)
feature(’CustomerId’, RabbitEvent-8)
feature(’Name’, RabbitEvent-8)
feature(’Address’, RabbitEvent-8)
feature(’PostalCode’, RabbitEvent-8)
feature(’City’, RabbitEvent-8)
feature(’TelephoneNumber’, RabbitEvent-8)
feature(’EmailAddress’, RabbitEvent-8)

featureMatch(’Id’, ’CustomerId’,
InvoiceEvent-5, RabbitEvent-8)
featureMatch(’Name’, ’Name’,
InvoiceEvent-5, RabbitEvent-8)
featureMatch(’Address’, ’Address’,
InvoiceEvent-5, RabbitEvent-8)
featureMatch(’PostalCode’, ’PostalCode’,
InvoiceEvent-5, RabbitEvent-8)
featureMatch(’City’, ’City’,
InvoiceEvent-5, RabbitEvent-8)

inOrder(InvoiceEvent-5, RabbitEvent-8)
```

```
sameValue('Id', 'CustomerId',
InvoiceEvent-5, RabbitEvent-8)
sameValue('Name', 'Name',
InvoiceEvent-5, RabbitEvent-8)
sameValue('Address', 'Address',
InvoiceEvent-5, RabbitEvent-8)
sameValue('PostalCode', 'PostalCode',
InvoiceEvent-5, RabbitEvent-8)
sameValue('City', 'City',
InvoiceEvent-5, RabbitEvent-8)
```



```
"@t": "2022-02-02T17:36:17.5869589Z",

"@mt": "Register customer: {Id}, {Name}, {Address}, {PostalCode}, {City}",

"Id": "778610af9e8040d88e73543d7b8407f1",

"Name": "J. Cole",

"Address": "2014 Forest Hills Drive",

"PostalCode": "S4G 7K2",

"City": "Frankfurt",

"Application": "InvoiceService",

"@@i": "a94ed641"
```

```
"@t": "2022-02-02T17:36:17.4213171Z",
"@mt": "{MessageType} - {Body}",
"MessageType": "CustomerRegistered",
"Body":
        {
        "CustomerId":"778610af9e8040d88e73543d7b8407f1",
        "Name":"J. Cole",
        "Address":"2014 Forest Hills Drive",
        "PostalCode":"S4G 7K2",
        "City":"Frankfurt",
        "TelephoneNumber":"519 234 2388",
        "EmailAddress":"middlechild@gmail.ca",
        "MessageId":"9f3b3ffd-f3c6-4ac1-a3ca-fc0c2994efbb",
        "MessageType":"CustomerRegistered",
        }
"Application": "AuditlogService",
"@@i": "5f829027"
```

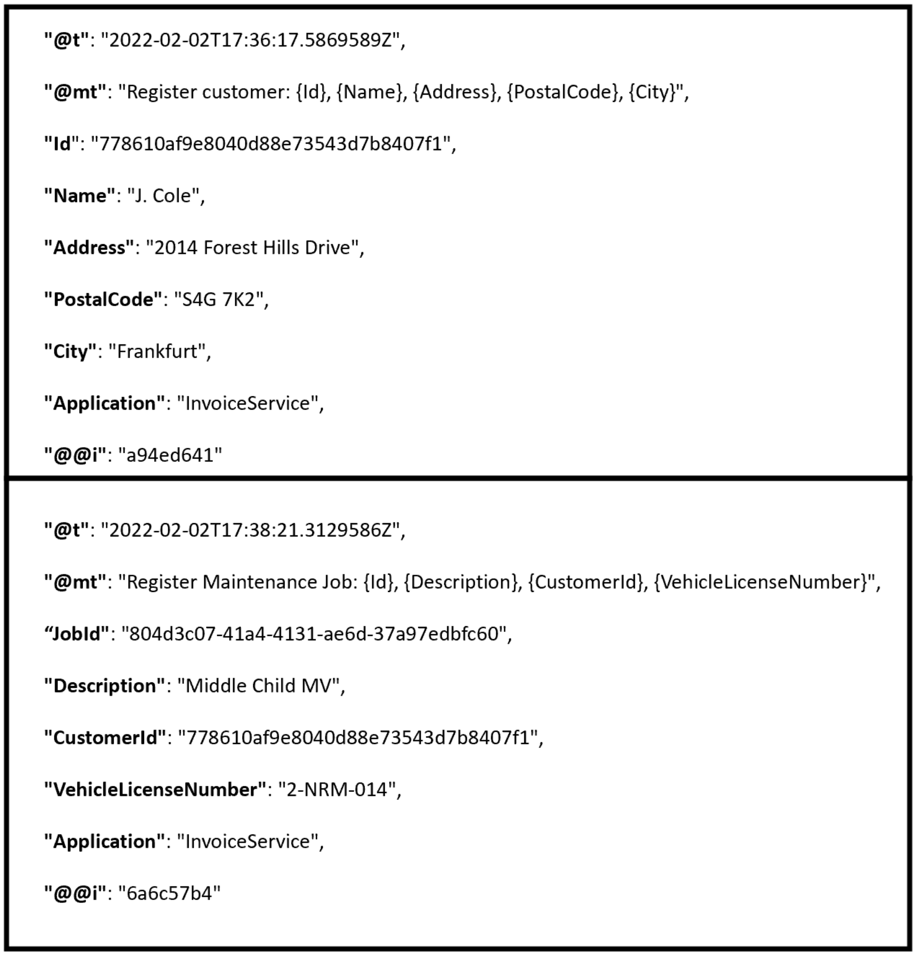
Figure 5.4: Log Breakdown for the event pair InvoiceEvent-5 (Top) and RabbitEvent-8 (Bottom)

As shown in the example above, the event pair produces twelve *feature(attribute, event)* predicate facts each representing one of the event's attributes. The event pair also produce five *featureMatch(attribute1, attribute2, event1, event2)* predicate facts each representing the attribute synonyms between the event pair. Additionally the event pair produced one *inOrder(event1, event2)* predicate fact. The timestamps for each event can be seen in Figure 5.4 and based on the timestamps, the two events occurred just over 160 milliseconds of each other. The final type of predicate fact produced is the *sameValue(attribute1, attribute2, event1, event2)* predicate fact for the attributes *Id* and *Name*. As shown in Figure 5.4, in which all the attributes of *InvoiceEvent-5* are also included in *RabbitEvent-8*. Comparing attribute values, all the attribute values for *(Id, Name, Address, PostalCode)* and *(City)* are consistent in both events. In this event pair there are five *sameValue(attribute1, attribute2, event1, event2)* predicate facts produced. Although the MLN is only trained using up to three pair rule, the existence of more than three pairs of attributes matching further support the association between the two events.

As a final example from the MLN inferences, we will breakdown a pair of events in which the MLN did not provide a high probability of them being a match. The event pair that will be examined is between the event *InvoiceEvent-5* and event *InvoiceEvent-6*. The event logs for both events are shown in Figure 5.5. A breakdown of the predicate facts generated by the two events that failed to meet any of the inference rules is presented below:

```
Event Pair:
(InvoiceEvent-5, InvoiceEvent-6)

Facts:
feature('Id', InvoiceEvent-5)
feature('Name', InvoiceEvent-5)
feature('Address', InvoiceEvent-5)
feature('PostalCode', InvoiceEvent-5)
feature('City', InvoiceEvent-5)
feature('JobId', InvoiceEvent-6)
feature('Description', InvoiceEvent-6)
feature('CustomerId', InvoiceEvent-6)
feature('VehicleLicenseNumber', InvoiceEvent-6)
```

```
featureMatch('Id', 'CustomerId',
InvoiceEvent-5, InvoiceEvent-6)

sameValue('Id', 'CustomerId',
InvoiceEvent-5, InvoiceEvent-6)
```



```
"@t": "2022-02-02T17:36:17.5869589Z",
"@mt": "Register customer: {Id}, {Name}, {Address}, {PostalCode}, {City}",
"Id": "778610af9e8040d88e73543d7b8407f1",
"Name": "J. Cole",
"Address": "2014 Forest Hills Drive",
"PostalCode": "S4G 7K2",
"City": "Frankfurt",
"Application": "InvoiceService",
"@@i": "a94ed641"
```

```
"@t": "2022-02-02T17:38:21.3129586Z",
"@mt": "Register Maintenance Job: {Id}, {Description}, {CustomerId}, {VehicleLicenseNumber}",
“JobId": "804d3c07-41a4-4131-ae6d-37a97edbfc60",
"Description": "Middle Child MV",
"CustomerId": "778610af9e8040d88e73543d7b8407f1",
"VehicleLicenseNumber": "2-NRM-014",
"Application": "InvoiceService",
"@@i": "6a6c57b4"
```

Figure 5.5: Log Breakdown for the event pair InvoiceEvent-5 (Top) and InvoiceEvent-6 (Bottom)

As shown in the example above, the event pair produced nine *feature(attribute, event)* predicate facts each representing one of the event's attributes. The event pair also produces one *featureMatch(attribute1, attribute2, event1, event2)* predicate fact that represents an attribute synonyms between the event pair. Notably the event pair produces zero *inOrder(event1, event2)* predicate facts, since the two events occurred did not occur within the pre-defined time frame. The timestamps for each event can be seen in Figure 5.5 and based on the timestamps, the two events occurred just over two minutes of each other. The final type of fact produced was the *sameValue(attribute1, attribute2, event1, event2)* fact for the attributes *Id* and *CustomerId*. As shown in Figure 5.5 the *Id* attribute value for *InvoiceEvent-5* events is consist with the *CustomerId* attribute value for *InvoiceEvent-6* with a value of *'778610af9e8040d88e73543d7b8407f1'*. However since the event pair only contains one matching attribute and did not occur within the pre-defined time frame then MLN infers that these two events are not associated with each other.

5.2 Summary

This chapter focused on the system-wide event matching aspect of our proposed framework for generating a Microservice Dependency Graph. The technique used for system-wide event matching centered around a probabilistic reasoning engine known as Markov Logic Networks. The implementation of the MLNs is based on the associations discovered in the conceptual event association in Section 4.4.2. In the first section of the chapter, Section 5.1.1, fact predicates for event pairs were developed. In that section, all possible microservice event pairings were analyzed and their resulting fact predicates were extracted based on the event's attribute values. These predicates were used later in the chapter for the development of the MLN. The next section, Section 5.1.2, rules were derived from the FCA concepts developed in Section 4.4.3. The derived rule-base consists of the intensions between all possible microservice event pairings in the FCA lattice. The extracted intensions represent the relationship between microservices in terms of their attributes. Through the usage of the fact predicates and the rule-base developed in the previous sections, Section 5.1.3 uses these inputs to develop, train and test a Markov Logic Network capable of inferring event pair associations. The output from the MLN consists of a list of microservice event pairs alongside a probability value representing the likelihood of the events being associated with each other. The resulting list of associated event pairs are used in the next chapter, Chapter 6 to develop the Microservice Dependency Graph.

Chapter 6

Microservice Dependency Graph Extraction

6.1 MDG Domain Model

In order to represent the Microservice Dependency Graph in a form which is processable by another software component which can automate the analysis process, we must first denote a schema (i.e. a domain model) for the proposed Microservice Dependency Graph (MDG). For this thesis, the MDG domain model is implemented as a collection of Meta-Object Facility Classes and is depicted in Figure 6.1. A description of each class and their corresponding attributes and associations are provided in the following sections.

Class Description

- *MDGNode*: Represents the node(s) in the microservice dependency graph
- *Microservice*: Represents the microservice(s) in a MSA based system
- *Infrastructure*: Represents the framework(s) that supports the systems organization
- *DataBase*: Represents the structured framework used for data storage
- *PubSub*: Represents the asynchronous communication framework known as ”Pub\Sub”

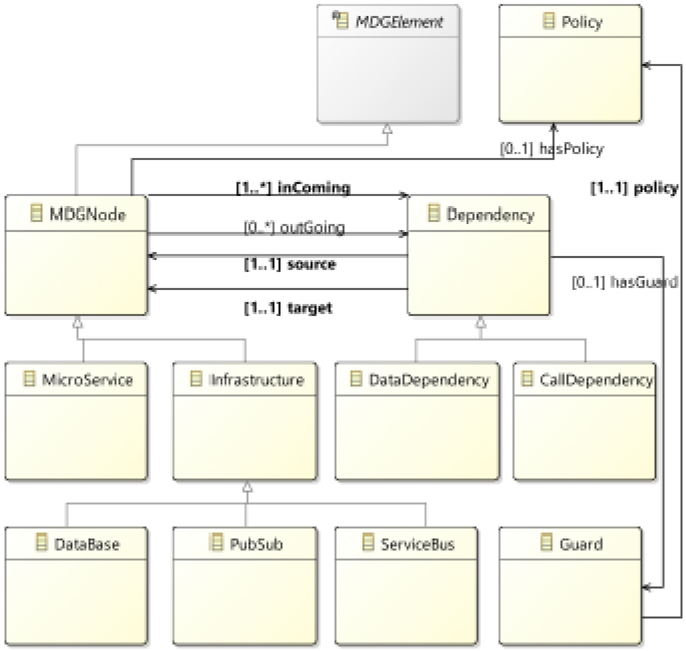


Figure 6.1: MDG Domain Model

- *ServiceBus*: Represents the communication framework utilized by the various microservices in the system

- *Dependency*: Represents the dependencies between microservices in the MSA based system

- *DataDependency*: Represents the dependencies associated with data between microservices in the MSA based system

- *CallDependency*: Represents the dependencies associated with invocation calls between microservices in the MSA based

- *Policy*: Represents the business logic requirements and restrictions for a system

- *Guard*: Represents the checkpoints used for the assessment of a predefined policy

Relationship Description

- *Association (MDGNode - Dependency)*: Each MDGNode is associated with one or more incoming Dependency as well as zero or more outgoing Depndency. Dependency type can be of either DataDependency or CallDependency

- *Association (Dependency - MDGNode)*: Each Dependency is associated with one source MDGNode and one target MDGNode
- *Association (MDGNode - Policy)* : Each MDGNode is associated with zero or one Policy
- *Association (Policy - Guard)* : Each Policy is associated with a Guard
- *Generalization (MDGNode - Microservice)*: MDGNode is a generalization of Microservice
- *Generalization (MDGNode - Infrastructure)*: MDGNode is a generalization of Infrastructure. Infrastructure type can be DataBase, PubSub, or ServiceBus
- *Generalization (Infrastructure - DataBase)*: Infrastructure is a generalization of DataBase
- *Generalization (Infrastructure - PubSub)*: Infrastructure is a generalization of PubSub
- *Generalization (Infrastructure - ServiceBus)*: Infrastructure is a generalization of ServiceBus
- *Generalization (Dependency - DataDependency)*: Dependency is a generalization of DataDependency
- *Generalization (Dependency - CallDependency)*: Dependency is a generalization of CallDependency

6.2 Event Chain Formation

The techniques and frameworks presented in Chapter 5 were used to deduce a list of event pairs, in which each event pair contains two events that are associated with each other. In the example from Section 5.1.3, this is manifested by the emission of facts of the form *match(InvoiceEvent-5, NotificationEvent-8)*, where *InvoiceEvent-5* is a specific event in the logs of *InvoiceEvent* service and, *NotificationEvent-8* is a specific event in the logs of the *NotificationEvent* service. Each such fact is associated with a probability score as deduced by the Markov Logic Network inferencing. For our work, we retain the facts for which their probability scores are equal or above 90%.

Once the reasoning process terminates, the selected $match(e_i, e_j)$ facts (i.e. the facts with a probability score >90%) are post-processed and we obtain a list of pairs of the form {⟨e_i, e_j⟩.... {⟨e_m, e_n⟩} indicating that event e_i matches with event e_j, and event e_m matches with event e_n.

The next step is to form *chains* from these pairs. Algorithm 6.2 analyzes these list of event pairs and creates a list of lists. Each inner list is a chain of associated (i.e. matched events). Here we present a simple version of the algorithm for illustration purposes.

The idea behind Algorithm 6.2 is to iterate through the list of pairs, initiate a new chain if the events do not belong already on the current chain, and for each such pair find all other pairs of events that match. For example, the pair ⟨e_i, e_j⟩ and the pair ⟨e_m, e_n⟩ will form a chain [e_i, e_j, e_m] if $match(e_i, e_j)$ and $match(e_i, e_m)$ or $match(e_j, e_m)$.

Algorithm 6.1 Event Chain Formation

```
– Let $Facts$ be a list of facts from the logs
– let $Rules$ be the list of Rules
- Let $matchedEvents$ be a list of pairs ⟨$e_i$, $e_j$⟩ of matched events
- Let $result$ = []
– Let $tempResult$ = []
– Let $matchedEvents = MLNReasoning(Facts, Rules)$

procedure calculateChains($matchedEvents$)
    for each ⟨ $e_i$, $e_j$⟩ in $matchedEvents$ do
        $tempResult$ = [$e_i$, $e_j$]
        for each ⟨ $e_k$, $e_m$⟩ in $matchedEvents$ where
            $e_k \neq e_i$ and $e_j \neq e_m$ do do
            if $e_k \in tempResult$ and
                $e_m \notin tempResult$ then
                $tempResult$ = add($tempResult$, $e_m$)
            end if
            if $e_m \in aggregateResult$ and
                $e_k \notin aggregateResult$ then
                $tempResult$ = add($tempResult$, $e_k$)
            end if
        end for
        $result$ = add($result$, $tempResult$)
    end for
    return $result$
end procedure
```

A excerpt of the chains obtained is depicted below.

```
CHAIN 1:
[’InvoiceService:RegisterCustomer’,
’NotificationService:RegisterCustomer’,
’Rabbitmq:CustomerRegistered’,
’WorkshopManagementEventHandler:RegisterCustomer’]

CHAIN 2:
[’InvoiceService:RegisterMaintenanceJob’,
’NotificationService:RegisterMaintenanceJob’,
’Rabbitmq:MaintenanceJobPlanned’,
’WorkshopManagementEventHandler:RegisterMaintenanceJob’]

CHAiN 3:
[’InvoiceService:FinishMaintenanceJob’,
’NotificationService:RemoveFinishedMaintenanceJob’,
’Rabbitmq:MaintenanceJobFinished’,
’WorkshopManagementEventHandler:FinishMaintenanceJob’]

CHAIN 4:
[’Rabbitmq:CustomerRegistered’,
’WorkshopManagementEventHandler:RegisterCustomer’,
’InvoiceService:RegisterCustomer’,
’NotificationService:RegisterCustomer’]
```

6.3 Path Extraction - Sequences of Events

The chains established in Section 6.2 contain information regarding which events are associated with each other, meaning it depicts events that correspond to the same instance of data creation and propagation. However the events in the chains do not contain any information regarding the *sequence* in which the events occurred (i.e. the directed paths) nor the sequence in which the microservices instantiated each event. From the example above, *CHAIN 1* contains four events, each containing the microservice that corresponds to that event. However, this does not paint the complete picture for the progressive flow of events for that instance of data. The event *Rabbitmq:CustomerRegistered*, is an event originating from the message broker. This however is an issue since the message broker is not a source of events rather an intermediary for the distribution of the event. Therefore the chain does not contain information regarding the source microservice for the message broker event. Additionally, the chain of events does not contain any information regarding SQL associations. In order to extract the complete paths for each chain, two additional pieces of information must be retrieved. The first is the source microservice for the message broker events, the second is the SQL event association for any of the chain events. Both of these pieces of information have already been established in Chapter 4, where MSA events were matched with their corresponding SQL events and message broker events were matched with their source microservice using SQL events.

Algorithm 6.2 illustrates the extraction of the individual paths from each chain list. The algorithm consists of two parts, the first part each event in the chain is identified as either a response event or a published event. In the second part, additional logic is implemented in order to establish the correct order sequence for the events. The logic used is presented below:

- Predicate - $Send(m_w, e_x)$: We define $Send(m_w, e_x)$ as the initiator(publisher) microservice m_w with its corresponding event e_x.
- Predicate - $Receive(m_y, e_x)$: We define $Receive(m_y, e_x)$ as the receiving microservice m_y receiving the event e_x.
- Predicate - $HappenedBefore(e_x, e_z)$: We define $HappenedBefore(e_x, e_z)$ as the event e_x occurring before the event e_z.
- Predicate - $Performs(m_y, e_z, m_w, e_x)$: We define $Performs(m_y, e_z, m_w, e_x)$ as the microservice m_y receiving the event e_x from microservice m_w and responding with event e_z such that e_x happens before e_z.

From the message broker event logs, we are able to establish the predicate $Send(m_w, e_x)$, since every data event logged in the message broker are intended to be distributed throughout the system. Additionally, from the event logs we can establish $Receive(m_y, e_z)$ through the existence of message broker exchange queue initialization, examples of these types of event logs are shown in Figure 6.2. Figure 6.2 contains three events, each event displaying the initialization of a channel in the message broker with the intended queue destination. By detecting these types of event logs occurring after an event is published into the message broker, we are able to establish to which microservice the published event will be directed. For example, in Figure 6.2 the queues established for each channel are *WorkshopManagement, Notification* and *Invoicing* are depicted. Additionally, from the event log the associated microservice is labeled under *Application* attribute. Therefore, each channel initialization provides the information regarding to which microservice the published event will be distributed. The predicate $HappenedBefore(e_x, e_z)$ is established by comparing the timestamps of the two events. The establishment of the predicates, $Send(m_w, e_x)$, $Receive(m_y, e_x)$ and $HappenedBefore(e_x, e_z)$ constitude the existence of the $Performs(m_y, e_z, m_w, e_x)$ predicate.

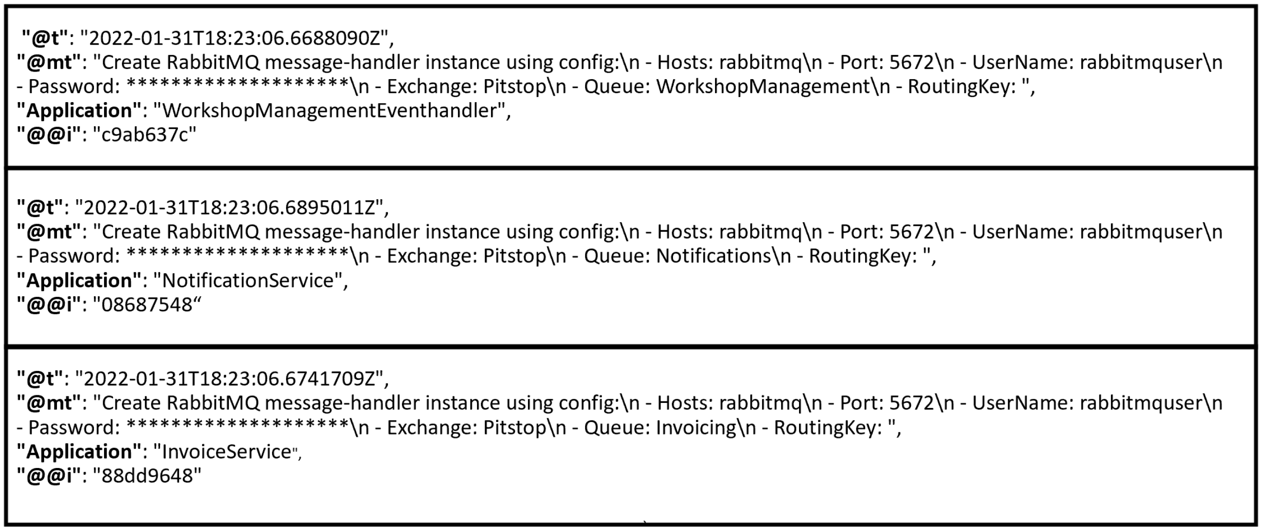
Algorithm 6.2 receives one parameter, *allChains*. This contains a list of chains of events that have been established to be associated with each other as discussed in Section 6.2. The algorithm begins by iterating through the list of chains and initializes a list and a dictionary. The *publisherEvent* list will contain the *MessageBroker* event and the *Publisher* microservice. The second is the *responseEvents* dictionary, in which the keys contain the *Response* events and the values represent their corresponding *SQL* event, (*lines 4-6*). Next the algorithm iterates through each event in the current chain and check to see if the event is from the message broker. If the event is from the message broker then the *Publisher* is extracted and stored, otherwise the event is considered a *Response* event in which case the *SQL* event is retrieved and stored, (*lines 7-13*). After all the events in the chain have been iterated through the algorithm iterates through each of the *responseEvents* and assemble the path. The path is constructed based on the message broker distribution logs previously explained. For each of the *Response* events the path will logically begin with the *Publisher* event followed by the *MessageBroker* event. This can be deduced as a *Response* event cannot occur prior to the instantiation from the *Publisher* event. Although there does not exist a logged event detailing the publisher microservice publishing the event to the message broker, for the purpose of completion, a 'pseudo-event' is created as a placeholder in the path list to demonstrate the source of the event. The label for this 'pseudo-event' will be in the format *Microservice:EventType*, where the *microservice* is the source microservice name and the *EventType* is the event type derived from the response events. Then, the path continues to the *Response* event, and finishes with its corresponding *SQL* event. The finalized path is then stored into the *allPaths* list, *LOC 18*, which will be used later on for the creation of the Microservice Dependency Graph.

Algorithm 6.2 Path Extraction from Event Chains

```
 1: Let allChains = calculateChains(matchedEvents)
 2: allPaths = []
 3: procedure PATHEXTRACTION(allChains)
 4:     for each chain in allChains do
 5:         publisherEvent = []
 6:         responseEvents = {}
 7:         for each event in chain do
 8:             if event.microservice() = MessageBroker then
 9:                 publisherEvents = [event, event.getPublisher()]
10:             else
11:                 responseEvents[event] = event.getSQL()
12:             end if
13:         end for
14:         for each $response_e$ in responseEvents do
15:             $publisher_e$ = publisherEvents[1]
16:             $middleWare_e$ = publisherEvents[0]
17:             $sql_e$ = $response_e$.getSQL()
18:             allPaths.append([$publisher_e$, $middleware_e$, $response_e$, $sql_e$])
19:         end for
20:     end for
21:     return allPaths
22: end procedure
```

An example of Chains and their resulting paths is shown below:

```
CHAIN 1:
[’InvoiceService:RegisterCustomer’,
’NotificationService:RegisterCustomer’,
’Rabbitmq:CustomerRegistered’,
’WorkshopManagementEventHandler:RegisterCustomer’]

Paths from CHAIN 1:
[’CustomerManagementAPI:CustomerRegistered’, ’Rabbitmq:CustomerRegistered’,
’InvoiceService:Register customer’, ’InvoiceSQL:InvoiceRegisterCustomer’]
[’CustomerManagementAPI:CustomerRegistered’,’Rabbitmq:CustomerRegistered’,
```



```
 "@t": "2022-01-31T18:23:06.6688090Z",
"@mt": "Create RabbitMQ message-handler instance using config:\n - Hosts: rabbitmq\n - Port: 5672\n - UserName: rabbitmquser\n
- Password: ********************\n - Exchange: Pitstop\n - Queue: WorkshopManagement\n - RoutingKey: ",
"Application": "WorkshopManagementEventhandler",
"@@i": "c9ab637c"
```

```
"@t": "2022-01-31T18:23:06.6895011Z",
"@mt": "Create RabbitMQ message-handler instance using config:\n - Hosts: rabbitmq\n - Port: 5672\n - UserName: rabbitmquser\n
- Password: ********************\n - Exchange: Pitstop\n - Queue: Notifications\n - RoutingKey: ",
"Application": "NotificationService",
"@@i": "08687548“
```

```
"@t": "2022-01-31T18:23:06.6741709Z",
"@mt": "Create RabbitMQ message-handler instance using config:\n - Hosts: rabbitmq\n - Port: 5672\n - UserName: rabbitmquser\n
- Password: ********************\n - Exchange: Pitstop\n - Queue: Invoicing\n - RoutingKey: ",
"Application": "InvoiceService",
"@@i": "88dd9648"
```

Figure 6.2: Message Broker exchange initialization examples

```
'NotificationService:RegisterCustomer', 'NotificationSQL:RegisterCustomer']
['CustomerManagementAPI:CustomerRegistered','Rabbitmq:CustomerRegistered',
'WorkshopManagementEventHandler:RegisterCustomer', 'WorkshopmagementSQL:RegisterCustomer']

CHAIN 2:
['InvoiceService:RegisterMaintenanceJob',
'NotificationService:RegisterMaintenanceJob',
'Rabbitmq:MaintenanceJobPlanned',
'WorkshopManagementEventHandler:RegisterMaintenanceJob']

Paths from CHAIN 2:
['WorkshopManagementAPI:RegisterMaintenanceJob', 'Rabbitmq:CustomerRegistered',
'InvoiceService:RegisterMaintenanceJob', 'InvoiceSQL:RegisterCustomer']
['WorkshopManagementAPI:RegisterMaintenanceJob', 'Rabbitmq:CustomerRegistered',
'NotificationService:RegisterMaintenanceJob', 'NotificationSQL:RegisterCustomer']
['WorkshopManagementAPI:RegisterMaintenanceJob', 'Rabbitmq:CustomerRegistered',
'WorkshopManagementEventHandler:RegisterMaintenanceJob' 'WorkshopmagementSQL:RegisterCustomer']
```

In the example above, *CHAIN 1* contains three *Response* events and one *MessageBroker* event. The derived *PATHS* each contain the additional events generated in Algorithm 6.2, the first additional event is the *Publisher* event. In the example for *CHAIN 1*, the *MessageBroker* event is *'Rabbitmq:CustomerRegistered'* and its corresponding *Publisher* event is *'CustomerManagementAPI:CustomerRegistered'*. Thus, each of the *PATHS* will contain two events at the beginning. The *Response* events are next followed by their corresponding *SQL* event. In this example, the *Response* event *'InvoiceService:Register customer'* is followed by the *SQL* event *'InvoiceSQL:InvoiceRegisterCustomer'*. Similarly the remaining two *Response* events *'NotificationService:RegisterCustomer'* and *'WorkshopManagementEventHandler:RegisterCustomer'* are followed by their *SQL* events *'NotificationSQL:RegisterCustomer'* and *'WorkshopmagementSQL:RegisterCustomer'* respectively. Therefore the original *CHAIN 1* results in the creation of three separate but related *Paths*.

6.4 Micro Service Dependency Graph Creation

The *Paths* generated in Section 6.3, are used to develop the Microservice Dependency Graph. The process for generating the MDG is simple and is summarized in Algorithm 6.3. The process iterates through each path in the list of event paths, and for each event in a path it creates a source node and a target node (if not already created) along with an edge indicating the current event in the path. In this respect, nodes denote microservices, middleware components (pub/sub infrastructure, service busses), and data base servers, while edges denote data transfer or call relations.

Here, we focus mostly on data exchange dependencies between microservices as in most situations data transfer is a result of parameter passing on a call or a result being returned as a result of a request. Furthermore, compliance analysis can benefit more by analyzing data exchange dependencies, as these are pivotal for assessing privacy, and access control policy violations.

As it was discussed in Section 2 microservice call dependencies have been investigated in the related literature, by analyzing the source code of the microservices involved, or by analyzing the corresponding Docker configuration files. Here, we take a different approach and we consider only run-time information by analyzing system logs.

Algorithm 6.3 MDG Creation

```
Let $MDG$ = []
Let $allPaths$ be a list of paths derived from the chains in calculateChains()

procedure CREATEMDG($result$)
    for each $path$ in $allPaths$ do do
        for each event $e_i$ in $path$ do do
            $m_k$ = FindMicroserviceEmmitting($e_i$)
            $m_n$ = FindMicroServiceReceiving($e_j$) and $match(e_i, e_j)$
            $Node_k = CreateNode(m_k)$ /* if not already created
            $Node_n = CreateNode(m_n)$ /* if not already created
            $edgek, n, I, j = CreateEdge(m_k, m_n, e_i, e_j)$
            MDG = add($\langle Node_k$, $Node_n$, $edgek, n, I, j\rangle$)
        end for
    end for
end procedure
```

6.4.1 Microservice Dependency Graph Applications

The proposed technique to compile a Microservices Dependency Graph is the first step towards a bigger goal, that is to perform compliance analysis and audits on microservice architecture systems. We envision three main uses of the MDG.

MDG Traversal

The first use is to perform off-line audits. As the system runs, a MDG will be created and will become stable after a short period of time, assuming there are no changes in the source code of the microservices involved. At any point, the MDG can be examined off-line (i.e. audited) for non-compliant data exchanges between microservices, to reveal possible violations. These violations may related to privacy, or access control. An example policy algorithm is shown in Algorithm 6.4. This algorithm for example is used to ensure that a microservice-A only receives a specific list of data from other microservices, and if any non-compliant data exchanges are encountered the algorithm will traverse the MDG and determine if there have been any subsequent data propagation's.

Algorithm 6.4 is divided into two main sections. The first part of the algorithm focuses on determing if the specified node has any data exchange violations. This section of the algorithm begins in *line 5*, where *policyViolations* will store a list containing all the data exchange

Algorithm 6.4 MDG Compliance Policy-A

```
Let MDG represent the Microservice Dependency Graph
Let edgePolicy be a list of edgeTypes allowed to be received a given microserivce
Let nodePolicy be a list of nodes (microservices) allowed to receive a given edgeType
procedure PolicyA(MDG, edgePolicy, nodePolicy)
    policyViolations = []
    for each node in nodePolicy do
        edgeViolations = []
        inEdges = MDG.getNode(node).getInEdges()
        for each inEdge in inEdges do
            if inEdge not in edgePolicy then
                edgeViolations.append(inEdge)
            end if
        end for
        policyViolations.append([node, edgeViolations])
    end for
    propagationList = []
    for violation in policyViolations do
        $node_v$ = violation.getNode()
        $edgeList_v$ = violation.getEdgeList()
        for each $edge_v$ in $edgeList_v$ do
            paths = MDG.findPaths($node_v$. $edge_v$)
            propagationList.append(paths)
        end for
    end for
    return propagationList
end procedure
```

violations for each of the specified nodes in *nodePolicy*. Then the algorithm, (*lines 6-8*) iterates through the list of specificed nodes and calls on the auxiliary methods *getNode()* and *getInEdges()* to return a list of incoming edges to the specified node. Next, in *lines 9-13* the algorithm iterates through each of the incoming edges and cross reference them with the specified *edgePolicy*. The specified edge policy list contains a list of edge types that are allowed to transmit data to the node. If the incoming edge is not in the specified list then the algorithm stores the edge violation in *edgeViolations*. After each incoming edge has been cross referenced, the violations that were discovered are stored in *policyViolations* alongside their corresponding node, (*line 16*).

The second half of the algorithm focuses on taking the violations from the first part of the algorithm and finding if any of the data violations propagated elsewhere in the graph. In *lines 17-20* the algorithm iterates through each of the violations, and during each iteration it stores the source node *nodeV* and iterates through all the edge violations *edgeV* that occurred in that node. During each iteration of the node's edge violation, the *MDG* is called to extract all the paths with the source node *nodeV* and edge type *edgeV*. The resulting paths are then added onto the *propagationList* and the algorithm continues iterating through the remaining edges and violations, (*lines 21-22*). Lastly the algorithm returns *propagationList* containing a list of data exchange violation paths with each path containing the source node, the edge type and all nodes that the edge violation was propagated onto.

RMI Guard Implementation

The second use is to perform run-time compliance analysis. This can be achieved by attaching "guards" in the MDG edges. More specifically, "guards" will entail business logic as to what type of data a microservice can send to another, or in which context such data exchange is legal. For example, during a transaction between microservice *A* and microservice *B* the MDG can be consulted and if different type of events, than the ones encoded in the MDG, are observed, then an alarm can be raised or the transaction will not be allowed to proceed. Figure 6.3 illustrates the process interactions in a sample implementation of RMI Guards.

Figure 6.3 depicts a RMI Guard implementation consisting of six components, the *Controller, Edge, Policy Repo, Policy Proxy, Policy Handler* and *External Service*. The *Controller*

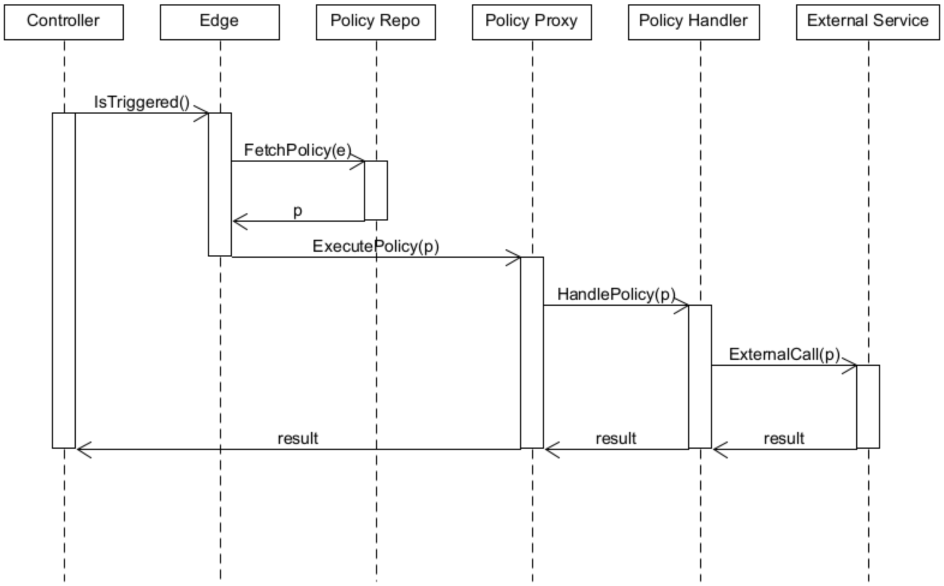


Figure 6.3: Sequence Diagram illustrating an RMI Guard implementation

initiates the process by calling on the *Edge* using the *isTriggered()*. Then, the *Edge* calls *Policy Repo* by using *FetchPolicy()* along with the parameter *'e'* , which represents an edge in the MDG. The *Policy Repo* contains a list of all the policies and their corresponding edges. The *Policy Repo* returns *'p'* which is the corresponding policy that correspond to the given edge *'e'*. After receiving the policy, *Edge* calls on *Policy Proxy* and sends the policy *'p'* using *ExecutePolicy()*. From there the *Policy Proxy* will call on *Policy Handler* using *HandlePolicy()* and giving it the policy *'p'* as a parameter. Then the *Policy Handler* takes the given policy and sends it to the corresponding component *External Service*. The *External Service* evaluates the given policy and returns a Boolean value *'result'* corresponding to the assessment of the policy. The Boolean value is then returned to *Policy Handler* which then returns it to *Policy Proxy* which will finally return the value to *Controller*. The *Controller* then determines the next course of action regarding the data propagation on the edge based on the returned value.

Development Aid

A third use of the system is during development. The development process of modern complex systems follows an agile methodology that calls for very short release cycles. This is referred to as *Continuous Software Engineering* and calls for *Continuous Integration* and *Con-*

tinuous Delivery (CI/CD). The challenge here is to minimize the failure risk despite the short release cycles and CI/CD processes. A MDG can help developers quickly run “what-if” scenarios and observe the impact the changes they introduced in the source code have on how data are exchanged between microservice, and whether any policies or constraints can be violated. Additionally the proposed MDG can be used for the detection of *Anti-Patterns*. As discussed in Section 2.6, there is already established research in the field of *Anti-Pattern Detection*. However most of the research revolves around the *Call Dependency Graph*. Therefore similar detection methods can be applied to our proposed MDG to detect potential *Ant-Patterns* in repects to data dependencies and not just call dependencies.

6.5 Summary

This chapter is the finalization of the proposed Microservice Dependency Graph framework. This chapter takes the associated event pairings that were developed in Chapter 5 and generates the MDG. At the start of the chapter in Section 6.1, a domain model alongside a description for its elements is provided in order to illustrate the structure of the MDG. The finalization of the MDG consists of three parts, event chain formation, path extraction, and MDG creation. The first part is in Section 6.2 in which all the event pairings from Chapter 5 are analyzed and a list of event chains are developed. The event pairs obtained through the MLN provide information regarding pairs of events that are associated with each other. However, in order to establish complete associations between events, the pairings must be cross-referenced with each other and compile event chains that represent all the data dependencies for an event flow. The process of developing event chains consisted of the implementation of the transitive property among event pairs. In the second section, Section 6.3, the event chains from the previous section are inspected and individual paths are extracted from each chain. Each path represents a directed flow of related data between events. In order to develop the complete paths corresponding to a single event flow, additional relational data was applied to the event chains. The additional relational data was derived from the event logs that established specific relationships between events. Specifically the relationship of *happenedBefore*, in which *EventA* happened before *EventB*. The combination of this relationship alongside the contents of the event chains are used to establish the event flow within each event chain. Lastly, in Section 6.4 the list of paths from the previous section are used to generate the Microservice Dependency Graph. Additionally, example usages of the developed MDG were described. These examples include, off-line audits using traversal algorithms, run-time compliance analysis using RMI Guard implementations, and as a development aid for *what-if* analysis and continuous software engineering.

Chapter 7

Experiments and Discussion

In this section we present the set up used for our experiments, the results obtained, and we discuss threats to validity.

7.1 Infrastructure Set up

The prototype system was developed in Python and Java and was applied to the PitStop open source microservice architecture system [60]. PitStop simulates the operations of a garage shop, from the customer registration phase to job planning and final invoicing phases. It comprises of 12 major components including a middleware component (RabbitMQ) and a MS-SQL database server. An overview of the system's solution architecture can be seen in Figure 7.1.

7.1.1 The Systems Microservice

As mentioned previously the system is comprised of 12 microservices, a description of each microservice alongside their responsibilities in terms of event handling is presented below

Web App

The *WebApp* is the front-end microservice, that provides users the ability to interact with the system. Through this service the users are able to manage vehicles, customers and workshop appointment setup. The *WebApp* service communicates strictly with the APIs in the system,

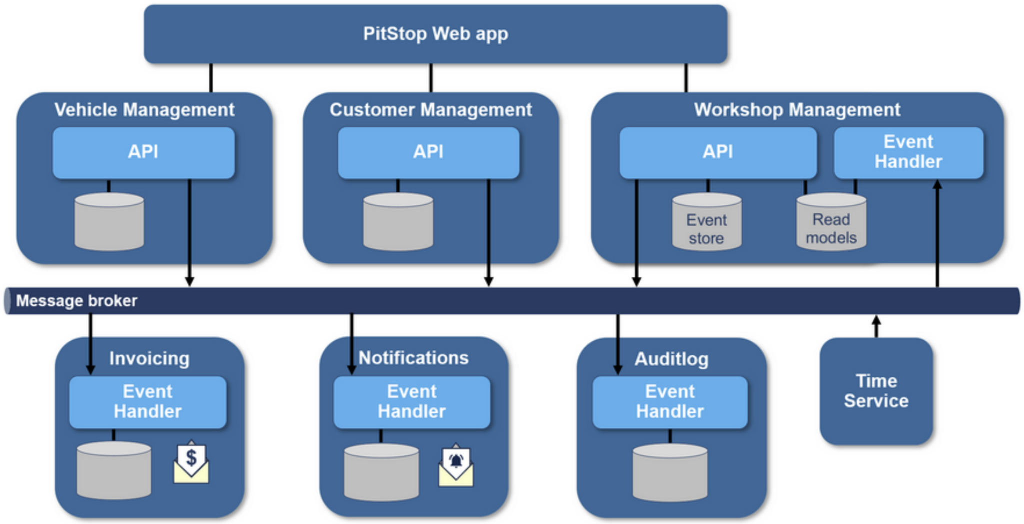


Figure 7.1: The solution architecture of the MSA system 'PitStop'

this includes *VehicleManagementAPI, CustomerManagementAPI* and *WorkshopManagementAPI*

Customer Management API

The *CustomerManagementAPI* microservice implements the API used by *WebApp* for the management of customers in the system. This microservice service is responsible for publishing the *CustomerRegistered* event into the *MessageBroker*

Vehicle Management API

The *VehiclerManagementAPI* microservice implements the API used by *WebApp* for the management of vehicles in the system. This microservice service is responsible for publishing the *VehicleRegistered* event into the *MessageBroker*

Workshop Management API

The *WorkshopManagementAPI* microservice implements the API used by *WebApp* for the management of workshop appointments in the system. This microservice service is responsible for publishing the *WorkshopPlanningCreated, MaintenanceJobPlanned* and *MaintenanceJobFinished* events into the *MessageBroker*

Message Broker

The *MessageBroker* microservice implements the RabbitMQ framework used in the system. This microservice is responsible for receiving events from producers and distributing events to the corresponding consumer queue. The exchange type of the established connections are all of type *Fanout*, meaning the message broker will distribute the incoming messages to all queues that have been bounded.

Auditlog Service

The *Auditlog* microservice is a logging microservice that captures all the events that occur in the *MessageBroker* and stores them.

Workshop Management Event Handler

The *WorkshopManagementEventHandler* microservice offers no API and is responsible for the handling of customer and vehicle data regarding workshop appointments. This microservice handles events strictly from the *MessageBroker*. The events handled by this microservice are *CustomerRegistered, VehicleRegistered, MaintenanceJobPlanned* and *MaintenanceJobFinished*.

Notification

The *Notification* microservice offers no API and is responsible for the notifying customers regarding their scheduled appointments. This microservice handles events strictly from the *MessageBroker*. The events handled by this microservice are *CustomerRegistered, DayHassPassed, MaintenanceJobPlanned* and *MaintenanceJobFinished*.

Invoice

The *Invoice* microservice offers no API and is responsible for the invoice creation regarding customers appointments. This microservice handles events strictly from the *MessageBroker*. The events handled by this microservice are *CustomerRegistered, DayHasPassed, MaintenanceJobPlanned* and *MaintenanceJobFinished*.

Time

The *Time* microservice offers no API and is only responsible for publishing the event *DayHasPassed* at a predefined time.

Mail

The *Mail* microservice implements MailDev to simulate an email. This simulation framework includes an SMTP server and a POP3 server alongside a front-end framework for user interaction.

SQL

The *SQL* microservice implements a single instance of MS SQL Server to host all databases used throughout all the microservices.

7.1.2 Systems Event Types

The system was ran under all possible scenarios and a log data set was collected. The logs gathered were generated by all the various components in the system that implemented a logging framework. The components that implemented logging frameworks include the message broker auditing service, the SQL database server, as well as a majority of the services in the system that either published or handled events. A description of each of the event types is provided below. The following descriptions are obtained from the documentation of the *PitStop* [60] application.

Customer Registered Event

This event is created and published to the *MessageBroker* by the *CustomerManagementAPI* microservice. This event corresponds to the registration of a new customer in the system. The customer data included in this event are the following, *CustomerId, Name, Address, PostralCode, City, TelephoneNumber* and *EmailAddress*.

Vehicle Registered Event

This event is created and published to the *MessageBroker* by the *VehicleManagementAPI* microservice. This event corresponds to the registration of a new vehicle in the system. The vehicle data included in this event are the following, *LicenseNumber, Brand, Type* and *OwnerId*

Workshop Planning Created Event

This event is created and published to the *MessageBroker* by the *WorkshopManagementAPI* microservice. This event corresponds to the intialization of an appointment event occurring. This event only contains one piece of information, which is *Date*.

Maintenance Job Planned Event

This event is created and published to the *MessageBroker* by the *WorkshopManagementAPI* microservice. This event corresponds to the scheduling of a new workshop appointment in the system. The workshop appointment data included in this event are the following, *JobId, StartTime, EndTime, CustomerId, Name, TelephoneNumber, License, Brand, Type* and *Description*.

Maintenance Job Finished Event

This event is created and published to the *MessageBroker* by the *WorkshopManagementAPI* microservice. This event corresponds to the finishing of a workshop appointment in the system. The workshop appointment data included in this event are the following, *JobId, StartTime, EndTime* and *Notes*.

Day Has Passed Event

This event is created and published to the *MessageBroker* by the *Time* microservice. This event contains no data due to the fact that it is used as a signalling event to trigger a reaction from other microservices.

7.2 Sample Run and Output

7.2.1 Data Collection

The PitStop system's loggers generate 26 different types of events. The Formal Concept Analysis of the data schemas considered all possible event types and their attributes and generated 23 concepts as depicted in Fig. 4.8 and in Table 4.1. In order to extract the log data set, we run the PitStop system 10 times under different operational scenarios. These sample runs generated 400 events related to data dependency, and each event was encoded in JSON format. These 400 events were analyzed and a total of 1,587 facts were generated. A breakdown of the distribution of events obtained throughout each operational scenario is depicted below.

Event Collection		
Operational Scenarios	Total Events	Data Dependency Events
Scenario 1	348	54
Scenario 2	471	65
Scenario 3	352	30
Scenario 4	641	61
Scenario 5	135	7
Scenario 6	446	20
Scenario 7	610	45
Scenario 8	212	23
Scenario 9	491	39
Scenario 10	720	56

7.2.2 Rule Training

In order to assign weights to the rules a training phase commenced. During the training phase we applied 864 facts to the rule base so that a Markov Logic Network can be created. Once the network was trained and rules were assigned weights, we have applied the trained rule-set to the rest 723 facts, in order to obtain ground *match(e1, e2)* type of facts, that indicate with a probability score whether two events *e1* and *e2* are associated (i.e they will belong to the same chain).

As discussed above, we have three rules deducing the *match(e1, e2)* ground facts. The first rule is applicable to events that share one pair of attributes with the same value, the second rule for the events that share two pairs and the third rule for events that share three pairs of attributes. Considering events that share more than three pairs of attributes makes the system less tractable, and generates *match(e1, e2)* ground facts with absolute certainty (probability 1), so we opted to limit the analysis to events that share three or less pairs of attributes. Our experiments assigned a weight of 1.51836 for the first rule (one common pair), a weight of 1.30469 for the second rule, and a weight of 1.22014. Even though for all practical purposes the rules have very similar weights, the training favours rules applicable to events sharing less attributes. This is happening because the number of events sharing three attributes is subsumed by the ones sharing two attributes, and similarly the events sharing two attributes are subsumed by the ones sharing one attribute. In this respect, this higher number of occurrences makes the weights differ in favour of fewer common attributes.

7.2.3 Fact Association

The application of rules in the 723 facts (approx.. 50% of the fact data set) yielded 676 pairs of matched events with various degrees of probability ranging from 12.01% to 99.99% of matching probability between two events. The standard deviation was 15.7%. Out of these 676 matched pairs, 586 had marching probability less than 20%, 64 had matching probability between 20.1% and 50%, none had probability between 50.1% and 89.9%, and 26 had matching probability between 90% and 99.9%. Out of the 64 in the range of 20.1% and 50%, none of them was a true match. This indicates a very clear classification of true positives and true negatives in the results obtained by the MLN reasoner. A breakdown of the event match probability distribution is depicted in the table below.

Event Match Probability Distribution	
Match Probability	Number of Matched Events
0%-20%	586
20.1%-50%	64
50.1%-89.9%	0
90%-99.9%	26

The experiments on this log data set also yielded 4 events out of 400 which are unaccounted for, that is they do not match with other events. Manual analysis indicated that these events where automatic responses by the middleware.

7.2.4 Microservice Dependency Graph Output

The Microservices Dependency Graph (MDG) which was compiled from this log data set is modelled using the JGraphT library and is visualized by the JGraphX library. It comprises of 12 nodes and 56 edges denoting different types of data transfer dependencies. As discussed above, in order to obtain the log data set used for our experiments, we have run the PitStop system 10 times under different scenarios. After a number of re-runs of different scenarios, the resulting MDG became stable so for this particular system there was no need to run the system under different scenarios for more than 10 times. The resulting MDG is shown in Figure 7.2.

7.3 Threats to Validity

There are four points we consider as threats to validity.

The first point deals with the complexity of the log schemas. In very large systems with many different logging systems, a manual analysis of the schemas to identify attribute synonyms may be a difficult task. In this respect, this step of the process should be automated by using schema matching techniques. An analysis of these techniques is important in order to identify their strengths and limitations when applied to event logs as opposed to general data base schemas.

The second point deals with the size and specificity of the rule-set. We have experimented

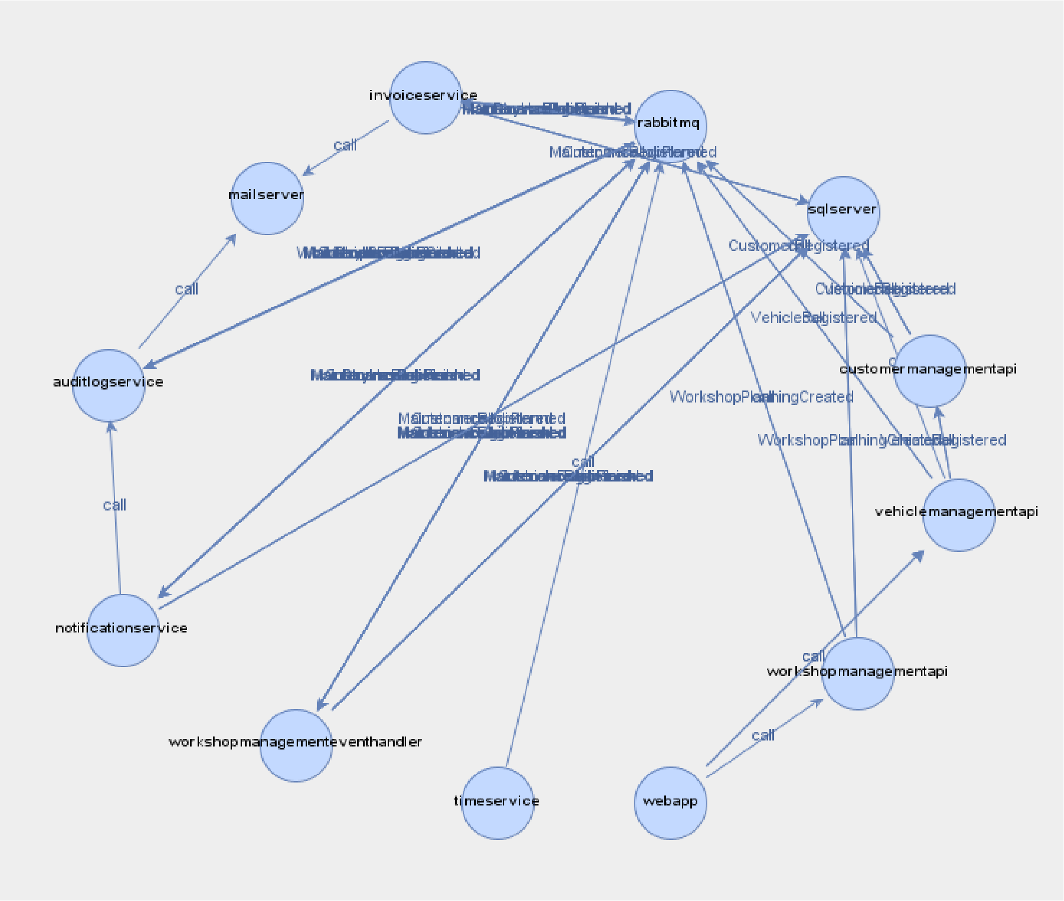


Figure 7.2: The MDG for the PitStop Application

as a proof-of-concept with simple rules. In large systems, there may corner cases and engineers have to draft the event matching rules carefully. Too many rules may make the system not tractable for real-time use (even though would be perfectly acceptable for off-line use i.e. auditing), and may limit recall, while too many general rules will hinder precision. Careful modeling and design of an optimal rule-set requires domain experts.

The third point deals with the availability of logs. In our system we have logs emitted from the middleware and the data base server. In deployments with limited logging capabilities the compilation of an MDG may not yield fully accurate results (i.e. missing dependencies). The approach is based on the assumption that adequate logging infrastructure is in place.

Finally, the fourth point deals with the MDG itself. We are extracting mostly data exchange type of dependencies between microservices, and we infer call information from these data exchanges. However, there may other types of dependencies that may not involve data exchanges, such as signals which may be issued by one microservice and trigger processing steps in another microservice. Currently, we are no able to capture these types of dependencies, and an extension of the MDG domain model along with log analysis techniques would be needed.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

In this thesis we presented a log analysis technique that allows for the extraction of data dependencies between microservices. The approach is based first on the reconciliation of schemas between events using Formal Concept Analysis, second on the association of events using Markov Logic Networks as a probabilistic inference engine, third on the identification of chains of related events in the log data set, fourth is the extraction of progressive data flow paths from the chains, and fifth on the compilation of a graph we refer to as Microservice Dependency Graph. Nodes in this graph correspond to microservices, or middleware components (pub/sub, service busses), or data base servers, while edges denote data exchanges between components. Data exchanges can take the form of data passing as call parameters or data that originate from external sources or data bases. The novelty of this work is that it utilizes domain logic in the form of a weighted rule-set, and a probabilistic reasoning engine to identify sets of related events in large log data sets.

The identification of dependencies between microservices is a pivotal first step towards the implementation of various frameworks. First, the MDG can be used to develop compliance analysis frameworks for microservice architectures. Compliance analysis may take the form of auditing a deployed system, to ensure that data are flowing from one component to another as specified (e.g. compliance with privacy policies), or data are not reaching or used by microservices not authorized to process these specific data (e.g. compliance with access control policies). Second, the MDG can be used to develop *what-if* analysis utilities. More specifically, as new features are added to the MSA system, an MDG can be compiled from test cases and the software engineers can identify what the interactions between the MSA components would be, if these features were finally released. This is an important aspect for minimizing the risk of unwanted interactions when the system is in production. Third, the MDG can be used to identify failure risk. A possible avenue of research would be to train a model to identify interaction patterns that are known to lead to failures. In this respect, when a new feature is added, the MDG interactions can be fed to the trained model and identify the failure risk proneness if this feature were to be released. This is an important utility for achieving continuous integration and continuous deployment (CI/CD).

8.2 Future Work

This work can be extended in several ways. One possible extension is to design a new or adapt an existing generic log schema for representing events. There exist many standards for event logging such as the Common Event Format (CEF), the NCSA Common Log Format, and Extended Log Format (ELF). Through a generic log schema, the MDG framework can be adapted to be capable of analyzing various different input streams form various types of microservices when creating the MDG. Another extension would be to harvest and analyze logs lower in the stack and in particular network traffic logs in order to disambiguate sources and targets of transactions or sequencing. Currently the MDG uses the MSA data event logs and SQL event logs to determine the source and target microservice, however a more reliable approach may be achieved with the usage of lower level network traffic logs. A third extension would be to develop a run-time monitor to identify interactions (i.e. edges) not currently in the MDG. Once the MDG and its corresponding edges have been established from the log files, a run time monitor can reveal new dependencies which are not currently modeled in the MDG. This may be related to one of two issues. First, this use case was not considered when the MDG was compiled, in which case the MDG is enhanced. The second issue is more sinister and may relate to the fact that the system has evolved and now is not compliant with its specification. A fourth extension would be to design and attach policies/guards to

MDG edges for compliance analysis. In Section 6.4.1, the possible application of guards were discussed. In this respect, future work for the MDG framework would involve implementing guards into the MDG model. The implementation of the guards would further improve the utility afforded by the run time monitor (future work previously stated). In combination with a run time monitor, any changes made to the system would be captured by the the run time monitor while the implemented guards would evaluate the changes as they are made. Finally, a fifth extension would be to associate MDG structure with Anti-Patterns, revealing design errors. Previous research in Anti-Pattern detection was discussed in Section 2.6. However, previous related research has focused on Anti-Pattern detection based on a call dependency graph. Using the MDG framework, additional analysis can be conducted to determine if any Anti-Patterns related to the data dependencies throughout the system.

Bibliography

[1] Alchemy: Open source ai. http://alchemy.cs.washington.edu/.

[2] How to become FedRAMP authorized. https://www.fedramp.gov/. Accessed: 2022-8-13.

[3] Hidenao Abe and Shusaku Tsumoto. Analyzing behavior of objective rule evaluation indices based on a correlation coefficient. In Ignac Lovrek, Robert J. Howlett, and Lakhmi C. Jain, editors, *Knowledge-Based Intelligent Information and Engineering Systems*, pages 758–765, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[4] arvindpdmn and devbot5S. Shift left. https://devopedia.org/shift-left, October 2018. Accessed: 2022-8-7.

[5] Leonard Peter Binamungu, Suzanne M Embury, and Nikolaos Konstantinou. Maintaining behaviour driven development specifications: Challenges and opportunities. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 175–184. IEEE, 2018.

[6] Grzegorz Blinowski, Anna Ojdowska, and Adam Przybyłek. Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access*, 10:20357–20374, 2022.

[7] André B Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd international workshop on Software and performance*, pages 195–203, 2000.

[8] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, et al. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 47–52, 2010.

[9] Richard H Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on scientific computing*, 16(5):1190–1208, 1995.

[10] Giuseppe Carenini. Lectures 30-32. `https://www.cs.ubc.ca/~carenini/TEACHING/CPSC422-17/index.html`.

[11] Pankaj Chejara and W. Wilfred Godfrey. Comparative analysis of community detection algorithms. In *2017 Conference on Information and Communication Technology (CICT)*, pages 1–5, 2017.

[12] Richard Cole, Thomas Tilley, and Jon Ducrou. Conceptual exploration of software structure: A collection of examples. In *CLA*, pages 135–148, 2005.

[13] Ward Cunningham. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, 1992.

[14] North Dan. Behavior modification. *Better Software*, 2006-03, 2006.

[15] Datalust. clef-tool. `https://github.com/datalust/clef-tool`.

[16] Datalust. Serilog. `https://datalust.co/seq`.

[17] Luc De Raedt and Luc Dehaspe. Clausal discovery. *Machine Learning*, 26(2):99–146, 1997.

[18] Kelley Dempsey, Victoria Yan Pillitteri, Chad Baer, Robert Niemeyer, Ron Rudman, and Susan Urban. Assessing information security continuous monitoring (iscm) programs. *NIST Special Publication*, 800:137A, 2020.

[19] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*, pages 195–216, 2017.

[20] Council of the European Union European Parliament. Council regulation (EU) no 2016/679, 2016.
`https://eur-lex.europa.eu/legal-content/EN/ALL/?uri=CELEX:32016R0679`.

[21] Robert Filepp, Constantin Adam, Milton Hernandez, Maja Vukovic, Nikos Anerousis, and Guan Qun Zhang. Continuous compliance: Experiences, challenges, and opportunities. In *2018 IEEE World Congress on Services (SERVICES)*, pages 31–32. IEEE, 2018.

[22] Brian Fitzgerald and Klaas-Jan Stol. Continuous software engineering and beyond: trends and challenges. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, pages 1–9, 2014.

[23] Brian Fitzgerald, Klaas-Jan Stol, Ryan O'Sullivan, and Donal O'Brien. Scaling agile methods to regulated environments: An industry case study. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 863–872. IEEE, 2013.

[24] Edgars Gaidels and Marite Kirikova. Service dependency graph analysis in microservice architecture. In Robert Andrei Buchmann, Andrea Polini, Björn Johansson, and Dimitris Karagiannis, editors, *Perspectives in Business Informatics Research*, pages 128–139, Cham, 2020. Springer International Publishing.

[25] Isuru Udara Piyadigama Gamage and Indika Perera. Using dependency graph and graph theory concepts to identify anti-patterns in a microservices system: A tool-based approach. In *2021 Moratuwa Engineering Research Conference (MERCon)*, pages 699–704, 2021.

[26] Bernhard Ganter and Rudolf Wille. *Formale Begriffsanalyse: mathematische grundlagen*. Springer-Verlag, 1996.

[27] Bernhard Ganter and Rudolf Wille. *Formal concept analysis: mathematical foundations*. Springer Science & Business Media, 2012.

[28] Javad Ghofrani and Daniel Lübke. Challenges of microservices architecture: A survey on the state of the practice. *ZEUS*, 2018:1–8, 2018.

[29] Pooyan Jamshidi, Claus Pahl, Nabor C Mendonça, James Lewis, and Stefan Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, 2018.

[30] Hiroshi Kazato, Shinpei Hayashi, Satoshi Okada, Shunsuke Miyata, Takashi Hoshino, and Motoshi Saeki. Feature location for multi-layer system based on formal concept analysis. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 429–434, 2012.

[31] G. Nagendra Kumar and Ch. Aswani Kumar. Generation of high level views in reverse engineering using formal concept analysis. In *2014 First International Conference on Networks & Soft Computing (ICNSC2014)*, pages 334–338, 2014.

[32] Anthony Kwan, Jonathon Wong, Hans-Arno Jacobsen, and Vinod Muthusamy. Hyscale: Hybrid and network scaling of dockerized microservices in cloud data centres. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 80–90, 2019.

[33] Yuanyuan Lan, Lei Fang, Mingzhu Zhang, Jianhua Su, Zhongguo Yang, and Han Li. Service dependency mining method based on service call chain analysis. In *2021 International Conference on Service Science (ICSS)*, pages 84–89, 2021.

[34] Ze Shi Li, Colin Werner, Neil Ernst, and Daniela Damian. Gdpr compliance in the context of continuous integration. *arXiv preprint arXiv:2002.06830*, 2020.

[35] Christian Lindig and Gregor Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of the 19th international conference on Software engineering*, pages 349–359, 1997.

[36] Dewei Liu, Chuan He, Xin Peng, Fan Lin, Chenxi Zhang, Shengfang Gong, Ziang Li, Jiayu Ou, and Zheshun Wu. Microhecl: High-efficient root cause localization in large-scale microservice systems. *CoRR*, abs/2103.01782, 2021.

[37] Haifeng Liu, Jinjun Zhang, Huasong Shan, Min Li, Yuan Chen, Xiaofeng He, and Xiaowei Li. Jcallgraph: Tracing microservices in very large scale container cloud platforms. In Dilma Da Silva, Qingyang Wang, and Liang-Jie Zhang, editors, *Cloud Computing – CLOUD 2019*, pages 287–302, Cham, 2019. Springer International Publishing.

[38] Peter Lutzeier. *Wort und Feld: Wortsemantische Fragestellungen mit besonderer Berücksichtigung des Wortfeldbegriffes*. PhD thesis.

[39] Shang-Pin Ma, Chen-Yuan Fan, Yen Chuang, Wen-Tin Lee, Shin-Jie Lee, and Nien-Lin Hsueh. Using service dependency graph to analyze and test microservices. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 02, pages 81–86, 2018.

[40] Shang-Pin Ma, Chen-Yuan Fan, Yen Chuang, I-Hsiu Liu, and Ci-Wei Lan. Graph-based and scenario-driven microservice analysis, retrieval, and testing. *Future Generation Computer Systems*, 100:724–735, 11 2019.

[41] Shang-Pin Ma, I-Hsiu Liu, Chun-Yu Chen, Jiun-Ting Lin, and Nien-Lin Hsueh. Version-based microservice analysis, monitoring, and visualization. pages 165–172, 12 2019.

[42] E Michael Maximilien and Laurie Williams. Assessing test-driven development at ibm. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 564–569. IEEE, 2003.

[43] Fabiola Moyon, Kristian Beckers, Sebastian Klepper, Philipp Lachberger, and Bernd Bruegge. Towards continuous security compliance in agile software development at scale. In *2018 IEEE/ACM 4th International Workshop on Rapid Continuous Software Engineering (RCoSE)*, pages 31–34. IEEE, 2018.

[44] Fabiola Moyón, Kristian Beckers, Sebastian Klepper, Philipp Lachberger, and Bernd Bruegge. Towards continuous security compliance in agile software development at scale. pages 31–34, 05 2018.

[45] Sam Newman. *Building Microservices*. O'Reilly Media, Inc., 2015.

[46] Feng Niu, Christopher Ré, AnHai Doan, and Jude Shavlik. Tuffy: Scaling up statistical inference in markov logic networks using an rdbms. *arXiv preprint arXiv:1104.3216*, 2011.

[47] Linda M Ott and Jeffrey J Thuss. The relationship between slices and module cohesion. In *Proceedings of the 11th International Conference on Software Engineering*, pages 198–204, 1989.

[48] Daniel Lowd Pedro Domingos. *Markov Logic*. Springer Cham, 2009.

[49] Ilaria Pigazzini, Francesca Arcelli Fontana, Valentina Lenarduzzi, and Davide Taibi. Towards microservice smells detection. 05 2020.

[50] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, Dec 2001.

[51] Akond Ashfaque Ur Rahman and Laurie Williams. Software security in devops: Synthesizing practitioners' perceptions and practices. In *2016 IEEE/ACM International Workshop on Continuous Software Evolution and Delivery (CSED)*, pages 70–76, 2016.

[52] Matthew Richardson and Pedro Domingos. Markov logic networks. *Machine learning*, 62(1):107–136, 2006.

[53] Tanvi Sahay, Ankita Mehta, and Shruti Jadon. Schema matching using machine learning. In *2020 7th International Conference on Signal Processing and Integrated Networks (SPIN)*, pages 359–366, 2020.

[54] Serilog. Serilog. `https://github.com/serilog/serilog`.

[55] Gregor Snelting and Frank Tip. Reengineering class hierarchies using concept analysis. *SIGSOFT Softw. Eng. Notes*, 23(6):99–110, nov 1998.

[56] Carlos Solis and Xiaofeng Wang. A study of the characteristics of behaviour driven development. In *2011 37th EUROMICRO conference on software engineering and advanced applications*, pages 383–387. IEEE, 2011.

[57] Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1:146–160, 1972.

[58] Edith Tom, Aybüke Aurum, and Richard Vidgen. An exploration of technical debt. *Journal of Systems and Software*, 86(6):1498–1516, 2013.

[59] Tom Tourwé and Kim Mens. Mining aspectual views using formal concept analysis. In *Source Code Analysis and Manipulation, Fourth IEEE International Workshop on*, pages 97–106. IEEE, 2004.

[60] Edwin van Wijk. Pitstop - garage management system. `https://github.com/EdwinVW/pitstop/wiki`.

[61] Janno von Stülpnagel, Jens Ortmann, and Joerg Schoenfisch. It risk management with markov logic networks. In Matthias Jarke, John Mylopoulos, Christoph Quix, Colette Rolland, Yannis Manolopoulos, Haralambos Mouratidis, and Jennifer Horkoff, editors, *Advanced Information Systems Engineering*, pages 301–315, Cham, 2014. Springer International Publishing.

[62] Eberhard Wolff. *Microservices: flexible software architecture*. Addison-Wesley Professional, 2016.

[63] Hamzeh Zawawy, Kostas Kontogiannis, John Mylopoulos, and Serge Mankovskii. Requirements-driven root cause analysis using markov logic networks. In Jolita Ralyté, Xavier Franch, Sjaak Brinkkemper, and Stanislaw Wrycza, editors, *Advanced Information Systems Engineering*, pages 350–365, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[64] Ciyou Zhu, Richard H Byrd, Peihuang Lu, and Jorge Nocedal. Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization. *ACM Transactions on mathematical software (TOMS)*, 23(4):550–560, 1997.

Curriculum Vitae

Name:	Andres Rodriguez Ishida
Degrees:	University of Western Ontario London, ON 2016 - 2020 BSc. Specialization in Computer Science University of Western Ontario London, ON 2020 - 2022 MSc. Software Engineering
Related Work:	Teaching Assistant, University of Western Ontario 2020-2022 CS2212 - Introduction to Software Engineering CS3357 - Computer Networks I CS4417 - Unstructured Data
Conference Presentations:	A. Rodriguez, ”Extracting Microservice Dependencies:A Log Based Approach”. In IBM CASCON ”Compliance by Design: Software Analytics and AIOps” Workshop 2021
Publications:	A. Rodriguez, K. Kontogiannis, C. Brealey, ”Extracting Micro Service Dependencies Using Log Analysis”. In Proceedings 29th IEEE Software Technology Conference 2022 (To Appear) IBM CASCON x EVOKE 2022, ”Framework for Extracting Microservice Dependencies Using Log Analysis” Exhibits Conference Track of WeaveSphere