Mono2Micro - From a Monolith to Microservices

The dynamic analysis of application in the JVM

Bernardo Rui dos Santos Andrade

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. António Manuel Ferreira Rito da Silva

Examination Committee

Chairperson: Prof. José Luís Brinquete Borbinha
Supervisor: Prof. António Manuel Ferreira Rito da Silva
Member of the Committee: Prof. Miguel Ângelo Marques de Matos

January 2021

Acknowledgments

I would like to thank, namely my family and closest friends, for the encouragement, support, caring and patience over these last five years. For always being there on my best and not-so-great moments without whom this project and other remarkable achievements would not have been possible.

I would also like to acknowledge my dissertation supervisor Prof. António Rito Silva and fellow colleagues for their insight, support and sharing of knowledge that have made this Thesis possible.

Last but not least, I'm grateful to the Advertio's team that helped me grow as a professional in such a friendly environment, gave me the first opportunity to materialize what I've been learning in recent years and changed my attitude towards tackling real-life problems on this area.

To each and every one of you – Thank you.

Abstract

The migration of monolith applications to a microservice architecture has long been a hot topic around major competitive business companies. By turning their complex systems into independently scalable services and managed by small agile software teams, the ideal of decreasing the time to market and increasing the availability of their services has become much more attainable.

This work leverages on a previous work which was focused on the problem of the complexity associated with software evolution when migrating a monolith to a microservices architecture by executing a static analysis on the monolith's codebase. Therefore, through a dynamic analysis, this work contributes the following research questions: (1) Does the information collected during run-time, for the same set of similarity measures, provide better results, in terms of the quality of the generated decompositions, when compared with the static information captured on the previous work? (2) Does a similarity measure, based on the dynamic behaviour of a system, generate decompositions which are performance optimized? (3) Are the used maintainability quality metrics correlated with performance?

To address each of these questions, a study was conducted on two monolith systems in which their behaviour was dynamically analysed. As result of the analysis we conclude that (i) neither of the analysis techniques, static and dynamic, outperforms the other, but the dynamic collection of data requires more effort (ii) the performance of a decomposition is correctly correlated with the maintainability quality metrics and no similarity measure provides better results than other.

Keywords

Monolithic Application; Microservices; Architecture Migration; Dynamic Code Analysis; Trace Compression; Maintainability quality metrics.

Resumo

A migração de aplicações monolíticas para uma arquitetura de microsserviços é, há muito tempo, um tema abordado pelas grandes empresas tecnológicas. Ao transformar os seus sistemas complexos em serviços escaláveis, independentes e geridos por pequenas equipas, o objetivo de reduzir o *time-to-market* dos seus produtos bem como o aumento da disponibilidade dos seus serviços tornou-se mais alcançável.

Este trabalho baseia-se num trabalho anterior que focou o problema da complexidade associada à evolução de software na migração de um monólito para uma arquitetura de microsserviços executando uma análise estática sobre o código do monólito. Portanto, através de análise dinâmica, este trabalho contribui com as seguintes questões de investigação: (1) As informações recolhidas em tempo de execução, para o mesmo conjunto de medidas de similaridade, fornecem melhores resultados, em termos de qualidade das decomposições geradas, quando comparadas com as informações estáticas capturadas no trabalho anterior? (2) Existe alguma medida de similaridade, baseada no comportamento dinâmico de um sistema, que gere decomposições otimizadas em termos de desempenho? (3) As métricas de qualidade de manutenção estão correlacionadas com desempenho?

Para responder a cada uma dessas questões, foi conduzido um estudo no qual se analisou a execução de dois sistemas monolíticos. Como resultado da análise, concluímos que (i) nenhuma das técnicas de análise, estática e dinâmica, supera a outra, mas a colheita dinâmica de dados requer mais esforço (ii) o desempenho de uma decomposição está corretamente correlacionado com as métricas de qualidade de manutenção e nenhuma medida de similaridade fornece melhores resultados que outra.

Palavras Chave

Monolíto; Microsserviços; Migração de Arquitetura; Análise dinâmica de código; Compressão de traços; Métricas de qualidade de manutenção.

Contents

List of Figures

List of Tables

Listings

Acronyms

CAP	Consistency Availability Partition tolerance
ORM	Object-Relational Mapper
MVC	Model-View-Controller
SCA	Static Code Analysis
DCA	Dynamic Code Analysis
MSA	Microservices Architecture
JVM	Java Virtual Machine
JSON	JavaScript Object Notation
LTW	Load-time Weaving
AOP	Aspect-oriented Programming
API	Application Programming Interface
MPSC	Multi-Producer-Single-Consumer
CSV	Comma-separated Values
RLE	Run-length Encoding
RTE	Reduced Trace Element
RAM	Random-access Memory
OLS	Ordinary Least Squares

1 Introduction

Contents

The concept of a microservices architecture has been acquiring numerous definitions over time in the industry and a broader attempt, presented in [1], describes it as an architectural style which end goal encompasses a set of cohesive and loosely coupled services, each one with its own independent responsibilities by functionally splitting an application. Initially employed by Amazon and Netflix, this architecture type emerged to solve the struggle of the software development processes not being able to keep up with the pace of business and customers demands and therefore it has been widely adopted by the software engineering and distributed systems communities.

However, in these days, numerous companies still live in a *monolithic hell*, possessing huge and complex legacy monolithic systems, where each change on the codebase makes it harder to understand and execute, leading to an increasingly maintenance overhead and a drastic decrease on the development productivity. Moreover, the horizontal scalability of a monolith is also compromised. The high concentration of services in the same executing component and the usage of a single shared database increases the overall system coupling and consequently turns it into a single point of failure (lack of reliability). Hence, breaking down a monolithic architecture into microservices can definitely help these companies by enabling continuous delivery and deployment of independent and scalable services that are easily maintained by autonomous agile teams.

Nevertheless, identifying the microservices boundaries, one of the major and most challenging software architects tasks, is still an obstacle when migrating a monolith system into microservices. Depending on the available architecturally significant requirements, an architect must reason about the inherent trade-offs when making such a complex and difficult partition. A wrong cut may have a direct impact on the distributed services' interfaces and, in the absence of a integrated development environment, the refactor becomes unmanageable.

As a consequence, a plethora of approaches have been proposed [2–5] to tackle this problem but the majority only focus on the domain model by identifying structural modules and their inter-relationships. Since persistent domain entities are highly coupled when accessed in the same transaction, the complexity of decomposing/relaxing the transactional behavior into several distributed transactions is being discarded, problem better known as the forgetting of the Consistency Availability Partition tolerance (CAP) Theorem migration smell identified by [6].

Each of these approaches differs from the other regarding the used techniques but they all have in common the same sequence of basic steps:(1) Collection: data is collected from the monolith's codebase or run-time behaviour; (2) Decomposition: a decomposition is generated by executing a clustering algorithm with the data collected in the first step according to a certain similarity measure (3) Analysis: a set of metrics is used to assess the generated decomposition quality.

In terms of collection, data is gathered from the monolith either by taking advantage of a static code analysis [4], by profiling its execution behaviour with dynamic analysis or by using both data collection techniques [7]. To generate a decomposition, different similarity measures are used where some only focus on the accesses to domain entities, others distinguish reads from writes, and others consider the sequence of accesses.

1.1 Previous work

To address the aforementioned limitations, the authors of [8,9] propose an analysis framework to support architects during the migration of monolith applications to a microservices architecture. This framework can be applied to any monolith that follows the Model-View-Controller (MVC) style and gives emphasis to the impact of the decomposition on the monolith business logic by identifying business applications transactional contexts. A business transaction (or functionality) should, ideally, be associated with a single transactional context, such that the decomposition of the monolith into a set of microservices does not have impact on the business logic. This impact is caused by the relaxation of the atomic transactional behavior of each functionality forcing its redesign and implementation by developers. Besides changing the inherent business logic, developers not only have to take into account the number of intermediate states but also the interactions between its intermediate states and the intermediate states of other functionalities and consequently, fault handling code must be written in order to compensate or recover from a plethora of possible interactions between functionalities. The more relaxed the consistency of a functionality, the more complex will be its implementation and therefore, the higher the cost - time and resources - of change of the system, also known as the Modifiability software quality attribute. To estimate this cost, the authors propose a complexity metric, formalized in the next chapter, based on information related to read and write sets and the invocation sequence of domain entities of each functionality.

The framework is also comprised by the same three steps:

- Collection: useful system data is collected from monolith systems, implemented using the Model-View-Controller architectural style, by using a static code analyser that captures transactional changes in the persistent model of domain entities triggered by the controllers, thus, corresponding to the monolith functionalities. As result of the collection, the functionalities accesses to the persistent entities are stored in JSON format. It consists in a key-value store that maps a controller name to a list of all the accesses that can possibly be made for that controller. An access is composed by the name of the accessed domain entity and the access type, either read or write;

- Decomposition: microservices candidates are identified using a hierarchical clustering (Python SciPy[1]) to process the collected data and, according to four similarity measures, generate a dendrogram of the domain entities. The generated dendrogram can be cut in order to produce different

[1] https://docs.scipy.org/doc/

decompositions, given the number of clusters. The decomposition tool also supports different combinations of similarity measures, for instance, it is possible to generate a decomposition with the following weights: 30% access, 30% read, 20% write, 20% sequence. These similarity measures are further discussed in the next chapter.

- Visualization and analysis: different views of the identified microservices and their inter-relationships are provided and also architects are allowed to interact with the tool to generate multiple decompositions, by varying the similarity measures weights and the number of clusters, and compare them according to the maintainability metrics: complexity, coupling and cohesion. Additionally, two different decompositions of the same system can be compared using the MoJoFM [10] distance metric. MoJoFM is a distance measure between two architectures expressed as a percentage. This measure is based on two key operations used to transform one decomposition into another: moves (Move) of entities between clusters, and merges (Join) of clusters. Given two decompositions, A and B, MoJoFM is defined as:

$$MoJoFM(A,B) = (1 - \frac{mno(A,B)}{max(mno(\forall A,B))}) \times 100\% \tag{1.1}$$

  where $mno(A,B)$ is the minimum number of Move and Join operations needed to transform A into B and $max(mno(\forall A,B))$ is the number of Move and Join operations needed to transform the most distant decomposition into B.

1.1.1 Problem

This work leverages the previous work [9], by studying whether the dynamic collection of the monolith's functionalities behavior can significantly improve the process of migrating a monolith to a microservices architecture.

The main limitation the previous work suffers from is the fact that only Static Code Analysis (SCA) is used to capture an abstracted model of the software state that may lead to some information loss (weaker and unreal results) [11]. This technique inevitably suffers from the intrinsic problem of identifying the run-time types when inheritance and polymorphism are used due to late binding.

1.2 Research Questions and Contributions

Therefore, this thesis studies two already known monolithic systems and presents an analysis of whether the Dynamic Code Analysis (DCA) technique provides significant differences when compared with the SCA in regards to the identification of candidate decompositions.

As a result, the following research questions are addressed:

RQ1: Does the information obtained using a dynamic code analyser, for the same set of similarity measures, provide better results, in terms of the quality of the generated decompositions, when compared with the static information captured on the previous work?

RQ2: Does a particular similarity measure provide better results in terms of the performance of the generated decompositions?

RQ3: Are the used maintainability quality metrics correlated with performance?

Simultaneously it tackles the existing limitations and improves the current analysis framework by:

- Employing DCA to collect data of the application run-time behaviour. New data artifacts were collected: the execution log/trace of the monolith's functionalities, containing the underlying dynamic types (when possible), and the execution time and frequency of each functionality.
- Refurbishing the analysis step internals to enable the analysis of larger volumes of data, by becoming less resource-hungry concerning time and memory.
- Introducing a performance metric: network communication latency. This metric simply calculates the number of hops between microservices during a distributed transaction. As one cannot predict the exact time taken by a request while travelling on the network or the execution time of a local transaction, it can be assumed that the more microservices a system has, the higher is the likelihood of increasing the latency of each functionality and consequently the overall system latency. Despite not being related with the main focus of the Mono2Micro project - maintainability - it was decided that this metric is a good fit to the current set of existing ones because it empowers the software architects reasoning with information of a different perspective whilst decomposing a system.

1.3 Organization of the Document

This thesis is organized and summarized as follows: Chapter 2 provides a thorough specification of the preceding work that serves as a fundamental basis and states what was accomplished. Chapter 3 presents the used DCA tool and explains each step of the data collection pipeline and corresponding artifacts. In Chapter 4, research questions are answered through the presentation of results of applying the analysis framework to the analysed systems. Chapter 5 introduces the state of the art in the microservices migration research and also describes the work that has been done in the field of how a dynamic code analysis can help in this migration process. Chapter 6 discusses the outcomes of this work and unveils potential future works. Finally, Chapter 7 concludes this thesis with a general synopsis of the work.

2

Background

Contents

This chapter starts by defining thoroughly the underlying key concepts of the Mono2Micro's framework. The second part of this chapter outlines what was accomplished by the previous work, which research answers were proposed, how they were addressed and what were the main findings.

2.1 Analysis framework formalization

A monolith is defined by its set of functionalities which execute in atomic transactional contexts and, due to the migration to the Microservices Architecture (MSA), have to be decoupled into a set of distributed transactions, each one executing in the context of a microservice.

Therefore, a monolith is defined as a triple (F, E, G), where F defines its set of functionalities, E the set of domain entities, and G a set of call graphs, one for each monolith functionality. A call graph is defined as a tuple (A, P), where $A = E \times M$ is a set of read and write of accesses to domain entities ($M = \{r, w\}$), and $P = A \times A$ a precedence relation between elements of A such that each access has zero or one immediate predecessors, $\forall_{a \in A} \#\{(a_1, a_2) \in P : a_1 = a\} \leq 1$, and there are no circularities, $\forall_{(a_1,a_2) \in P_T} (a_2, a_1) \notin P_T$, where P_T is the transitive closure of P. The precedence relation represents the sequences of accesses associated with a functionality.

2.1.1 Similarity Measures

The definition of similarity measures establishes the distance between domain entities. Domain entities that are closer, according to a particular similarity measure, should belong to the same microservice. Therefore, to minimize the amount of distributed transactions a functionality is decomposed in, domain entities will tend to be closer when accessed by the same functionalities. The authors define four similarity measures: access, read, write and sequence.

2.1.1.A Access similarity measure

The access similarity measure measures the distance between two domain entities, $e_1, e_2 \in E$, as:

$$sm_{access}(e_1, e_2) = \frac{\#(funct(e_1) \cap funct(e_2))}{\#funct(e_1)} \tag{2.1}$$

where $funct(e)$ denotes the set of functionalities in the monolith whose call graph has a read or write access to e. This measure takes a value in the interval 0..1. When all the functionalities that access e_1 also access e_2 then it takes the value 1.

2.1.1.B Read and Write similarity measures

These measures tend to include in the same microservice, domain entities that are read or written together, respectively. Since the cost of reading and writing is different in the context of distributed transactions, as writes introduce new intermediate states in the decomposition of a functionality, the next two similarity measures distinguish read from write accesses in order to reduce the number of write distributed transactions:

$$sm_{read}(e_1, e_2) = \frac{\#(funct(e_1, r) \cap funct(e_2, r))}{\#funct(e_1, r)} \tag{2.2}$$

$$sm_{write}(e_1, e_2) = \frac{\#(funct(e_1, w) \cap funct(e_2, w))}{\#funct(e_1, w)} \tag{2.3}$$

where $funct(e, m)$ denotes the set of functionalities in the monolith whose call graph has an access according to mode m, read or write, respectively.

2.1.1.C Sequence similarity measure

Finally, another similarity measure that is found in the literature groups domain entities that are frequently accessed in sequence, in order to reduce the number of remote invocations between microservices, i.e., the domain entities that are frequently accessed in sequence should be in the same microservice. Therefore, the sequence similarity measure is defined as:

$$sm_{sequence}(e_1, e_2) = \frac{sumPairs(e_1, e_2)}{maxPairs} \tag{2.4}$$

where $sumPairs(e_1, e_2) = \sum_{f \in F} \#\{(a_i, a_j) \in G_f.P : (a_i.e = e_1 \wedge a_j.e = e_2) \vee (a_i.e = e_2 \wedge a_j.e = e_1)\})$ is the number of consecutive accesses of e_1 and e_2, where $G_f.P$ is the precedence relation for functionality f, and $maxPairs = max_{e_i, e_j \in E}(sumPairs(e_i, e_j))$ is the max number of consecutive accesses for two domain entities in the monolith.

2.1.2 Metrics

A decomposition of a monolith is a partition of its domain entities set, where each element is included in exactly one subset, a cluster, and a partition of the call graph of each one of its functionalities. Therefore, given the call graph G_f of a functionality f, and a decomposition $D \subseteq 2^E$, the partition call graph of a functionality $partition(G_f, D) = (LT, RI)$ is defined by a set of local transactions LT and a set of remote invocations RI, where each local transaction

(i) is a subgraph of the functionality call graph, $\forall_{lt \in LT} : lt.A \subseteq G_f.A \wedge lt.P \subseteq G_f.P$;

(ii) contains only accesses in a single cluster of the domain entities decomposition, $\forall_{lt \in LT} \exists_{cinD} : lt.A.e \subseteq c$;

(iii) contains all consecutive accesses in the same cluster, $\forall_{a_i \in lt.A, a_j \in G_f.A} : ((a_i.e.c = a_j.e.c \wedge (a_i, a_j) \in G_f.P) \implies (a_i, a_j) \in lt.P) \vee ((a_i.e.c = a_j.e.c \wedge (a_j, a_i) \in G_f.P) \implies (a_j, a_i) \in lt.P)$.

From the definition of local transaction, results the definition of remote invocations, which are the elements in the precedence relation that belong to different clusters, $RI = \{(a_i, a_j) \in G_f.P : a_i.e.c \neq a_j.e.c\}$. Note that, in these definitions, the dot notation is used to refer to elements of a composite or one of its properties, e.g., in $a_j.e.c$, $.e$ denotes the domain entity in the access, and $.c$ the cluster the domain entity belongs to.

2.1.2.A Complexity Metric

The complexity of migrating a functionality in the context of a decomposition is translated to the required cognitive load effort the software developer has to address when redesigning a functionality. Splitting its transactional behavior into several distributed transactions, introduces multiple intermediate states due to the lack of isolation. Therefore, the impact on the functionality migration redesign effort is caused by the following factors:

- The number of local transactions, because each local transaction may introduce an intermediate state;
- The number of other functionalities that read domain entities written by the functionality, because it adds the need to consider the intermediate states between the execution of the different local transactions;
- The number of other functionalities that write domain entities read by the functionality, because the functionality redesign has to consider the different states these domain entities can be.

$$complexity(f, D) = \sum_{lt \in partition(G_f, D)} complexity(lt, D) \tag{2.5}$$

The complexity of a functionality is the sum of the complexities of its local transactions.

$$\begin{gathered} complexity(lt, D) = \#\cup_{a_i \in prune(lt)} \\ \{f_i \neq lt.f : dist(f_i, D) \wedge a_i^{-1} \in prune(f_i, D))\} \end{gathered} \tag{2.6}$$

The complexity of a local transaction is the number of other distributed functionalities that read, or write, domain entities, written, or read, respectively by the local transaction. The auxiliary function $dist$ identifies distributed functionalities, given the decomposition; a_i^{-1} denotes the inverse access, e.g.

$(e_1, r)^{-1} = (e_i, w)$; and $prune$ denotes the relevant accesses inside a local transaction, by removing repeated accesses of the same mode to a domain entity. If both read and write accesses occur inside the same local transaction, they are both considered if the read occurs before the write. Otherwise, only the write access is considered. These are the only accesses that have impact outside the local transaction.

2.1.2.B Coupling and Cohesion Metrics

Coupling and cohesion are important qualities of any software system, particularly, in a system implementing a microservices architecture, because one of its goals is to foster independent agile teams. Therefore, we intend to measure the coupling and cohesion of the monolith decomposition.

The cohesion of a cluster of domain entities depends on the percentage of its entities that is accessed by a functionality. If all the cluster's assigned entities are accessed each time a cluster is accessed, it means that the cluster is cohesive:

$$cohesion(c) = \frac{\sum_{f \in funct(c)} \frac{\#\{e \in c.e : e \in G_f.A.e\}}{\#c.e}}{\#funct(c)} \tag{2.7}$$

where $funct(c)$ denotes the functionalities that access the cluster c, and $G_f.A.e$ the entities that are accessed by functionality f.

The coupling between two clusters is defined by the percentage of entities one cluster exposes to other cluster. It can be defined in terms of the remote invocations between the two clusters.

$$coupling(c_i, c_j) = \frac{\#\{e \in c_j : \exists_{ri \in RI(c_i, c_j)} e = ri[2].e\}}{\#c_j.e} \tag{2.8}$$

where $RI(c_i, c_j)$ denotes the remote invocations from cluster c_i to cluster c_j

2.2 Previous results

The preceding work [9] also addressed the problem of microservices identification in a monolith while managing the impact that the relaxing of atomic transactional behavior has on the redesign and implementation of the functionality. Thereby the following two research questions were tackled:

RQ1: Is it possible to calculate the cost/complexity associated with the migration to a microservices architecture due to the introduction of relaxed consistency into the business behavior?

RQ2: Which similarity measures are more effective in the generation of a candidate microservices decomposition in terms of the cost of migration?

To validate their approach, three monolith systems were statically analysed. With respect to the first research question, the proposed complexity metric was evaluated through the analysis of its correlation with the cohesion and coupling metrics. Results show that:

- the complexity metric grows as the number of clusters increases. This was expected since with the increase of the number of clusters, the implementation of functionalities is split between more clusters;
- complexity has some negative correlation with cohesion. Given the high amount of outliers, it was concluded that the complexity of migrating the functionalities is not directly correlated with the cohesion;
- the higher the complexity the higher the coupling. Decompositions with high complexity tend to have a high number of distributed transactions for each functionality and, as a consequence, more interactions between clusters, which is what the coupling criteria measures.

Concerning the second research question, the similarity measures were evaluated against the complexity obtained for each combination and the achieved conclusions are that:

- there is no single combination of similarity measures that is more effective in the generation of candidate microservices decomposition in terms of the cost of migration;
- using a combination of measures provides better results than using only one measure in terms of complexity;
- the sequence similarity measure always provides the worst result in terms of complexity.

3 Dynamic Data Collection

Contents

This chapter starts by characterising the monolithic systems that were under profiling. Then a description of the DCA tool used to instrument the monoliths' run-time behaviour is presented detailing its internals, advantages and disadvantages which constitute the main reason of why this tool was adopted to eliminate the current technology limitation. Subsequently, an in-depth explanation is given regarding the data collection pipeline implementation on the Mono2Micro project. Figure 3.1 is also presented to facilitate the visualization of the entire pipeline, the relationships between steps and respective deliverables.

3.1 Monitored systems

Data was collected from the two already known monolith systems used in the previous work [9], LdoD[1] and Blended Workflow[2]. Both Java client-server web applications follow the MVC architectural style, where Spring-Boot[3] controllers process input events by triggering transactional changes in the model, thus, corresponding to the monolith functionalities. A monolith is designed considering its controllers as transactions that manipulate a persistent model of domain entities implemented using Fénix Framework[4] Object-Relational Mapper (ORM). The choice of only using these two systems was based on the leverage of the existing knowledge about the inner workings of both proprietary softwares. Not only there was access to their source code but also it was known how to execute and interact with the web applications which facilitated the collection of data and the integration with the DCA tool.

3.2 Dynamic analysis tool

The tool used to collect run-time data is Kieker[5] proposed by the authors of [12] as an extensible dynamic analysis tool designed for application performance monitoring and architecture discovery. Kieker is structured in two parts - monitoring and analysis - that together form a modular pipe-and-filter pipeline. Dynamic data is firstly gathered by monitoring probes that are programmed with Aspect-oriented Programming (AOP) and then converted into monitoring records that are written to a monitoring log (file system, database or a message oriented middleware). If an analysis is desired afterwards, already implemented readers can fetch the monitoring records and pass them to specific filters for further processing to obtain an envisioned use case metric. All the previous state components of the pipeline are fully customizable and can be extended from the existent ones. A dynamic trace analysis is available as a built-in feature which is in charge of recording information about the system operation executions and

[1] https://github.com/socialsoftware/edition
[2] https://github.com/socialsoftware/blended-workflow
[3] https://spring.io/projects/spring-boot
[4] https://fenix-framework.github.io/
[5] http://kieker-monitoring.net/

control flow traces and, as an add-on, visual architectural models are also provided such as call and dependency graphs and sequence diagrams for the associated traces. Another remarkable advantage that comes with this framework is that code instrumentation not only can be defined by annotations in the source code but also via an external XML configuration file thus automating the program profilling process. Regarding the introduced overhead, a quantitative evaluation is presented showing a small linear increase[6] in run-time which depends on the average number of activated monitoring points, in other words, the granularity of instrumentation. Just as importantly, Kieker is backed by an extensive user guide and its source code is present in a GitHub repository which exhibits recent signs of activity[7] (a total of 238 commits to the master branch during the whole year of 2020).

3.3 Dynamic analysis collection pipeline

The collection pipeline is comprised of three main steps - Setup, Monitoring and JavaScript Object Notation (JSON) File Generation - as illustrated by Figure 3.1. In general, only the first and last steps may require manual adjustments depending on the execution environment and the software in question, whereas the second is fully handled by Kieker and associated tools. In total, three types of deliverables are produced by the end of the pipeline: configuration files, created in the Setup step; raw data files - logs - extracted from the system while being monitored and finally, the desired outcome, the JSON file including semantic data prepared to be analysed.

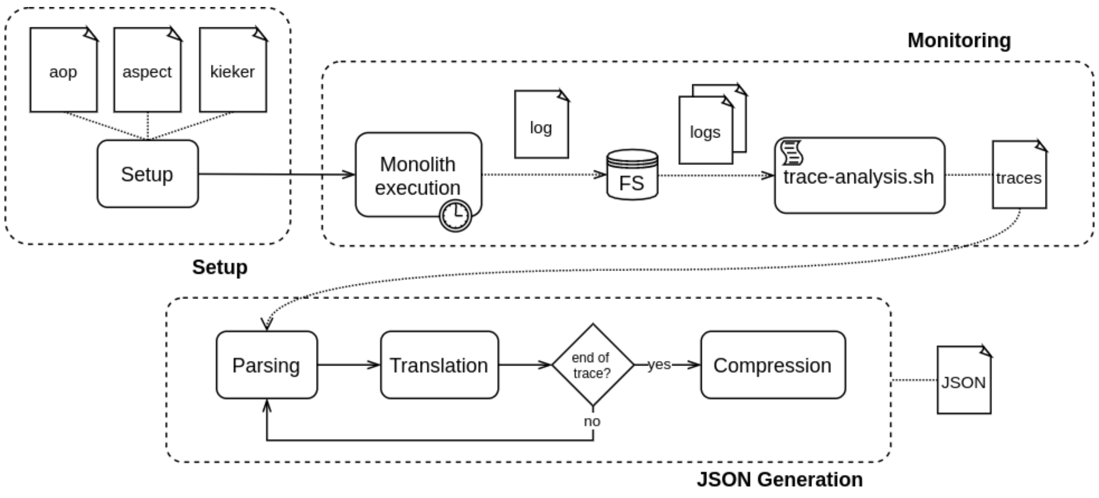

Figure 3.1: Data Collection pipeline

[6]the total constant overhead (probe triggering (ΔB) + data collection (ΔC) + writing data (ΔD)) for monitoring an operation with Kieker is under 2.5 microseconds

[7]https://github.com/kieker-monitoring/kieker/graphs/contributors

3.3.1 Setup

This step establishes the building blocks of the monitoring step and answers the questions of where to collect data from, what data should be collected and how it should be handled after being collected. The first two questions were answered with the help of AspectJ[8]. It was used to instrument the byte code of the Java applications and to develop new technology-specific probes as suggested in Kieker's user guide. Since it leverages the use of AOP, no manual modifications on the source code were required neither the insertion of annotations on intended methods. An additional concept to take into consideration is the chosen weaving type: Load-time Weaving (LTW). As one of the main goals of this work is to overcome the static analysis limitation regarding the identification of dynamic types when inheritance and polymorphism are involved, LTW had to be used because it postpones the weaving process until the class files are loaded into the JVM and, as a result, the concrete objects' types can be determined as well as the methods effectively called (also called as dynamic/late binding). The remaining weaving types explicitly supported by AspectJ - Compile-time and Post-compile weaving - operate exclusively during the static binding process and thus, do not fulfill the requirements.

To enable LTW, a jar-file containing the AspectJ weaver has to be explicitly provided by the run-time environment through the JVM weaving agent option as it can be observed in Listing 3.1. The authors of Kieker already provide a jar-file including the weaver bundled with their tool (kieker-1.14-aspectj.jar).

Listing 3.1: Example of an excerpt of a .pom file responsible for setting the project configuration details

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
        <agents>
            lib/kieker-1.14-aspectj.jar
        </agents>
    </configuration>
</plugin>
```

[8]The Eclipse Foundation (2011). The AspectJ Project. http://www.eclipse.org/aspectj/

After enabling LTW, the weaver must be configured with an *aop.xml* file in the classpath in the *META-INF* directory. An example is provided in Listing 3.2. The file contains two essential sections:

Listing 3.2: Example of an aop.xml file

```xml
<!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN" "http://www.aspectj.org/dtd/aspectj_1_5_0.dtd">
<aspectj>
    <weaver options="-Xset:weaveJavaxPackages=false">
        <include within="pt.ist..*"/>
    </weaver>

    <aspects>
      <aspect name="pt.ist.socialsoftware.edition.ldod.aspects.GettersAndSetters"/>
    </aspects>
</aspectj>
```

- **Weaver:** defines weaver options and specifies the classes to be considered for instrumentation and therefore, the high-level location(s) where the data will be collected. The adopted strategy[9] ensures that the weaver reaches all the executable source code, even in the absence of explicit accesses to persistent entities to account for possible background interactions of the framework.
- **Aspects:** defines one or more aspects to be woven into the classes. According to the AOP paradigm, an aspect is defined as the encapsulation of program logic (so-called concerns) that crosscuts multiple parts in a program[10]. This behavioural property not only provided a more fine-grained approach in terms of pinpointing the intended logic parts of the program but also allowed to specify what data should be extracted from these.

Before diving into the two developed aspects files, it's also crucial to take into account the following concepts that were a result of the work done by Kiczales *et al.* [13] on AOP which had a huge impact on the software engineering discipline:

> According to Wikipedia's[11] simple and self-explanatory definition: "An aspect can alter the behavior of the base code by applying an **Advice**[12] (additional behavior) at various **Join Points**[13] (points in a program) specified in a query called **Pointcut**[14] that detects whether a given join point matches."

[9]dependant on the project hierarchical folder structure
[10]Logging and security exemplify common crosscutting concerns
[11]https://en.wikipedia.org/wiki/Aspect-oriented_programming
[12]https://www.eclipse.org/aspectj/doc/next/progguide/semantics-advice.html
[13]https://www.eclipse.org/aspectj/doc/next/progguide/semantics-joinPoints.html
[14]https://www.eclipse.org/aspectj/doc/released/progguide/semantics-pointcuts.html

The implementation of aspects was driven primarily by the accuracy and performance of the collection process. Since extra behaviour is added during run-time, care must be taken so that the extra overhead does not jeopardize the user experience. The less precise the collection, the longer the system will be occupied writing meaningless data to the file system. Thus, to minimize the extra overhead, *pointcuts* (regular expressions) were designed as precise as possible and the logic inherent to the *Advice* had to be adapted to not degrade the overall performance of the system.

The persistence of domain entities is guaranteed by the FénixFramework ORM that auto-generates class files responsible for handling the intrinsic database logic. All these classes follow the same nomenclature pattern regarding their names and the names of their public access methods[15]. Classes' names are formatted as $< entityName >$*_Base* and the names of data access methods, the ones responsible for manipulating the respective entity's persistent state, are formatted as $\{get, set, add, remove, delete\} < attributeName >$. For instance, in the LdoD codebase exists a class called *Category* that, in order to be persisted, has to extend the already generated class *Category_Base* to access such methods as *getName*, *setName*, *addTag* and *removeTag*.

An excerpt of the fully fledged aspect responsible for defining all pointcuts is presented in Listing 3.3. The full version is available in Listing A.1.

Listing 3.3: Excerpt of the GettersAndSetters.java file containing the aspect's pointcuts

```
@Aspect
public class GettersAndSetters extends AbstractOperationExecutionAspect {
   // -------------------------------- BASE CLASS METHODS --------------------------------
   @Pointcut("execution(public void pt.ist..*.*_Base.add*(..)) && within(pt.ist..*.*_Base)")
   public void publicBaseAddMethods() {}
     // ------------------------------------- CONTROLLER METHODS -------------------------------------
   @Pointcut("@annotation(org.springframework.web.bind.annotation.PutMapping) && execution(* *(..))")
   public void controllerPutMethods() {}
   // -------------------------------------- KIEKER METHOD --------------------------------------
   @Pointcut("(cflow(controllerPutMethods()) && publicBaseAddMethods()) || controllerPutMethods()")
   public void monitoredOperation() {}
}
```

In simple terms, the goal of this aspect is to intercept the calls to the FenixFramework's data access methods aforementioned for a given controller execution so that the corresponding Advice can derive the necessary information from the method. To achieve this level of interception, four types of *Join Points*[16] were used:

- $within(C)$ - Matches when the executing code belongs to class C. Used to intercept code belonging only to *_Base* classes;

[15]Methods whose Java access modifier is *public*
[16]https://www.eclipse.org/aspectj/doc/next/progguide/language-joinPoints.html

- $execution(S)$ - Matches when a particular method body executes with signature S. Used to restrict the interception to only consider the execution of public methods, whose name starts with *get*, *set*, *remove* and *add*, and the execution of controller methods;
- $cflow(P)$ - Picks each join point in the control flow of the join points picked by pointcut P. Used to get the control flow of join points of each controller's method execution. Should not be used alone (without an intersection with other join point) because the focus is on specific join points.
- $@annotation(T)$ - Matches any element which has an annotation of type T.[17]. Used to get all controller methods that are Application Programming Interface (API) endpoints. These methods are annotated with: *RequestMapping, PostMapping, GetMapping, PatchMapping, PutMapping, MessageMapping* and *DeleteMapping*;

Regarding the second aspect, the plan was to adapt the already existing file, implemented by the authors of Kieker, so that the information of the executed method could be introduced in the monitoring record and consequently written to the file system. Listing 3.4 represents an excerpt of this file containing part of the logic introduced on the *Advice*. Full version available in Listing A.2.

This new information has to be, at least, the same one that static analysis gathers to determine the corresponding accesses during the Translation step. Thus, the implemented logic collects the following information:

1. **Declaring class type name** - The name of the class containing the definition of the method
2. **Method's name** - The name of the called method
3. **Target's type** - The type of the object that calls the method
4. **Method's arguments types** - The types of the objects passed as arguments
5. **Method's return type** - The type of the object returned by the method

As a result, in order to hold this data, the following compact format was adopted: **1:2:3:4:5**
For example, this data was captured from a log line:

$$Member\_Base : setRole : Member : ["MemberRole"] : void$$

Considering that this logic is executed during the collection process at run-time, it should account for two major contingencies: (i) objects may not have an explicit dynamic type, meaning the object can be $null$ (ii) Java generic types employed on iterables[18] are no longer available[19] and therefore, it's impossible to discover the iterable type;

[17] https://www.eclipse.org/aspectj/doc/next/adk15notebook/annotations-pointcuts-and-advice.html
[18] https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html
[19] https://docs.oracle.com/javase/tutorial/java/generics/erasure.html

The algorithm displayed in Listing 3.4, reflects the assumptions that were taken to address these contingencies:

1. Whenever a dynamic type can't be discovered, the static type is used. See lines 19-20. If neither of them can't be found the type is considered unknown and is sent in blank.

2. Whenever in the presence of an iterable object with at least one element, its type is considered to be the type of the first element (for performance reasons). Otherwise, if empty, the type is considered unknown and is sent in blank. See lines 4-13.

Listing 3.4: Excerpt of the AbstractOperationExecutionAspect.java file containing the Advice with the extra added behaviour, in particular, the part that figures out the type of the returned value

```java
1  // Getting the returned value type of the given method
2  retval = thisJoinPoint.proceed();
3  if (retval != null) {
4      if (retval.getClass().getSimpleName().equals("RelationList")) {
5          RelationList retValAux = (RelationList) retval;
6          Iterator<?> i = retValAux.iterator();
7
8          if (i.hasNext()) {
9              Object firstElement = i.next();
10             if (firstElement != null) {
11                 newSignature.append(firstElement.getClass().getSimpleName());
12             }
13         }
14     } else {
15         newSignature.append(retval.getClass().getSimpleName());
16     }
17 }
18 } else {
19     String returnedValueStaticType = ((MethodSignature) thisJoinPoint.getSignature()).getReturnType().
           getSimpleName();
20     newSignature.append(returnedValueStaticType);
21 }
```

At this time, only the question of how data should be handled after being collected remains open. Straight answer is: it depends on how the DCA tool, Kieker, is configured. The best way to setup Kieker is through the usage of a special configuration file called $kieker.monitoring.properties$.

An excerpt with the most impactful non-default properties on the monitoring process is presented in Listing 3.5. Full version available in Listing A.3. These properties allow to customize the Kieker's writing environment as follows:

RecordQueueFQN: The internal synchronized queue implementation to use. Before adding an

Multi-Producer-Single-Consumer (MPSC) queue implementation, the overall performance of the system, while being analysed, was surprisingly slow to the point where any action on the system became unfeasible.

RecordQueueSize: Defines the queue's capacity[20] in terms of the number of records. This property should be sensible to the amount of memory available on the hosting platform. A maximum capacity of five million records seemed reasonably appropriated given the available resources on both systems.

RecordQueueInsertBehavior: Specifies the action to take when there are incoming records and the queue is full. There are three possible actions: (i) Monitoring is terminated with an error (value 0) (ii) Writing is blocked until there is room for more records (value 1). This option has a critical caveat: if the writing process is blocked, it will affect drastically the system's performance because the process it's no longer asynchronous. Thus its use is discouraged when monitoring on a production/uncontrolled environment. (iii) New records are discarded until space is available (value 2). Since there are no more options available, this was the chosen one. It exchanges the loss of records by no system performance degradation.

maxEntriesInFile: The maximum number of records/lines in a (log) file. One million entries seemed reasonable given that a full log file weights around 132MB.

compression: The compression method used to compress each log file. In this case, each log file is written as zipped binary file. For instance, a zipped log file with one million entries ended up weighing just 17MB. Other compression methods can be used, such as *gzip*, but *zip* was chosen because it offers a good trade-off between compression speed and ratio. This property proved to be extremely valuable due to the file systems' little disk space.

Listing 3.5: Kieker's configuration properties that have the most impact on the writing process

```
kieker.monitoring.core.controller.WriterController.RecordQueueFQN=org.jctools.queues.
    MpscArrayQueue
kieker.monitoring.core.controller.WriterController.RecordQueueSize=5000000
kieker.monitoring.core.controller.WriterController.RecordQueueInsertBehavior=2
kieker.monitoring.writer.filesystem.FileWriter.maxEntriesInFile=1000000
kieker.monitoring.writer.filesystem.FileWriter.compression=pt.ist.socialsoftware.edition.
    ldod.compressors.ZipCompressor
```

[20] Depending on the used queue implementation (Bounded or Unbounded), this property can establish either the initial or maximum capacity

3.3.2 Monitoring

Having set up all the requirements, the monitoring process may start. During run-time, when a controller executes, Kieker will capture the accesses to the domain entities in the context of the execution of a controller and will store the Monitoring Records represented as Comma-separated Values (CSV). Listing 3.6 shows a sample file system Monitoring Log entry.

Listing 3.6: Example of a monitoring log entry

```
$1;1597411157437353328;;Category_Base:getTaxonomy:Category::Taxonomy;127;1597411157437348525
    ;1597411157437353202;;2;1
```

The entry not only contains the method's data according to the format previously explained, (*Category_Base:getTaxonomy:Category::Taxonomy*), but also context information such as timestamps, typically with nanosecond resolution, for the start and the end of an execution (from 1597411157437348525 to 1597411157437353202) and "efficient facilities to log the information needed to reconstruct (distributed) traces from the Monitoring Log reliably: an execution order index (2) and an execution stack size (1)". The first two values, $\$1$ and 1597411157437353328, are intrinsically related with Kieker's internals so their significance can be discarded.

After the monitoring has ended, one is incapable of telling in which order the calls to the ORM have happened in the context of controller execution. To achieve the desired representation of an ordered sequence of executions (trace), Kieker provides a built-in trace analysis tool that is able to (i) reconstruct traces from a set of log files, (ii) produce visual architectural models of the reconstructed traces and (iii) generate aggregate information regarding the whole reconstruction process such as the frequency and maximum number of executions of each trace.

Only the trace reconstruction feature was useful for the purpose of this work since there were no plans to improve the current Mono2Micro's visualization phase and the JSON Generation step already aggregates the necessary data.

3.3.2.A Trace Reconstruction

Traces can be reconstructed with two representations: execution traces and message traces. An execution trace representation of a trace is simply the ordered sequence of executions while a message trace describes a trace in terms of an ordered sequence of messages instead of executions. An execution represents the operation call which started the execution whereas a message can represent both start and end of an execution when the control flow is returned to the calling execution.

The execution trace representation was adopted as it mimics the same behaviour of the static analysis approach used in the previous work. This representation can be obtained by executing the *trace-analysis.sh* script, provided by Kieker, as shown in Listing 3.7 with the following options:

ignore-invalid-traces: The script execution will not abort on the occurrence of an invalid trace.

print-Execution-Traces: Execution trace representations of valid traces are saved into a text file

print-invalid-Execution-Traces: Execution trace representations of invalid traces are saved into a text file. Used for debugging purposes.

Listing 3.7: Example of the command to generate execution traces with Kieker's trace analysis tool

```
$ ./trace-analysis.sh \
    --ignore-invalid-traces \
    --inputdirs <path of input directory containing log files> \
    --outputdir <path of output directory> \
    --print-Execution-Traces \
    --print-invalid-Execution-Traces
```

As a result, the file *executionTraces.txt* is generated containing all the valid traces. Listing 3.8[21] shows an example of a valid execution trace representation. The first line of each representation indicates the trace's unique identifier (6), the timestamp at which the first execution started ($minTin$) and the timestamp at which the last execution ended ($maxTout$), respectively and, last but not least, the maximum execution stack size ($maxEss$) considering all the trace's executions. All the following lines follow the format of $< x[y,z]\ t1 - t2\ methodData >$ where x identifies the trace that the executions belongs to, y denotes the execution order index and z the execution stack size, $t1$ and $t2$ indicate the start and the end times of an execution respectively, and finally the method's data ($methodData$) according to the already mentioned format.

Listing 3.8: Excerpt of a trace from the file *executionTraces.txt*

```
TraceId 6 (minTin=1597411129007876729; maxTout=1597411129008257865; maxEss=1):
<6[0,0] 1597411129007876729-1597411129008257865 ReadingController:resetPreviousRecommendedFragments:::String>
<6[1,1] 1597411129008071866-1597411129008107197 FenixFramework:getDomainObject::["String"]:ExpertEditionInter>
<6[2,1] 1597411129008142154-1597411129008155507 FragInter_Base:getFragment:ExpertEditionInter::Fragment>
<6[3,1] 1597411129008168612-1597411129008177066 Fragment_Base:getXmlId:Fragment::String>
<6[4,1] 1597411129008195218-1597411129008202017 FragInter_Base:getXmlId:ExpertEditionInter::String>
```

[21] Irrelevant information removed to increase readability

3.3.3 JSON Generation

As the name suggests, this step is focused on producing a JSON file from the execution traces obtained in the previous step. This JSON file represents a mapping between functionality names and functionality objects, where each object has a traces field that consists in a list of trace objects. Each trace is characterized by a unique identifier and a compressed list of accesses observed for a specific functionality execution. An $Access$ is composed by the numeric identifier of the domain entity and the access type, either read or write. After completing the three sub-steps - Parsing, Translation and Compression - each trace of (compressed) accesses is prepared to be associated with its corresponding functionality if it proves to be unique.

3.3.3.A Parsing

On this phase, the $executionTraces.txt$ file is parsed line by line to obtain the functionality name, the trace's unique identifier and the methods' data of each trace.

Taking the first three lines of the trace presented in Listing 3.8 as an example, the functionality name would be $ReadingController.resetPreviousRecommendedFragments$, the trace identifier 6 and $FenixFramework : getDomainObject :: ["String"] : ExpertEditionInter$ the method's data. This data is further parsed to collected each respective semantic information - name and the types of the target, arguments and return objects - and subsequently used as input to the translation step.

3.3.3.B Translation

The goal of this sub-step is two-fold: (i) translate accessed entities names to numeric identifiers (ii) translate the method's data, received as an input, to the corresponding accesses to persistent domain entities.

The translation to unique numeric identifiers is maintained until all traces are parsed. When a new persistent domain entity is discovered, a counter is increased and its value is assigned to the entity. Hence uniqueness is ensured.

The translation algorithm behaves as follows:

- If the method's name starts with get it means that persistent entities were read. The accessed entities can be the target's type and the return type.
- Alternatively, if the method's name starts with set, add, $remove$ or $delete$ then persistent entities may have been written. In this case, the written entities can be the target's type and each argument type.

When translating a type, it's necessary to first check whether or not that it corresponds to a persistent entity. For instance, if there is a case where the target's type is unknown ($null$) and the return/argument is of type $String$, then no persistent entities were actually accessed. If it proves to be a persistent entity, then its numeric identifier is associated with the access type ($Read$ or $Write$) thus constituting an $Access$.

3.3.3.C Compression

One of the biggest challenges of this work, that arose from executing a dynamic analysis, was the massive amount of data generated allied to the difficulty of processing large traces due to time and space requirements. The explanation for this amount is quite simple: loops. Since code is executed in this type of analysis, a loop may repeat its body X amount of times and as a consequence, the size of execution traces may increase linearly. Therefore, after the whole execution trace has been translated into a list of accesses, this list is subject to compression.

This compression was achieved by leveraging the use of an enhanced version of the linear sequence compression algorithm called *Sequitur* [14] that can represent a sequence as a hierarchical structure. The authors of [15] were able to improve the compression ratio of *Sequitur*, and still preserving its linear performance, by combining it with Run-length Encoding (RLE). A compressed sequence is a new sequence composed of a new type of element. For ease of communication, let's call it Reduced Trace Element (RTE). A RTE can be seen as a container that attaches the information of how many times the element that it holds is repeated. It can only hold one of the following elements: a $Rule$ or an element of the original sequence. A $Rule$, identified with the symbol $\#$, is a special element that specifies how many of the next elements are going to be repeated.

For instance, the compressed version of the 27-length sequence shown in Listing 3.9, where letters maybe symbolize accesses, only contains 13 elements as illustrated in Listing 3.10. Although the original size has been reduced to more than half, this is still not the best possible compression, as discussed with one of the authors of [15]. The best compressed sequence can be observed in Listing 3.11 which contains only 12 elements. Unfortunately, no good solution was found to implement this last extra mile "without having at least $O(n^2)$ complexity" until this work is published.

In a small scale, depicted by this example, it may seem a minor improvement but assuming that traces can reach lengths of dozens of millions of accesses, the impact on the compressed size would be much more noticeable.

Listing 3.9: Example of a sequence

```
B, B, C, C, B, D, D, D, E, D, E, E, D, E, E, B, C, C, B, C, C, B, C, C, B, C, C
```

Listing 3.10: Example of the compressed sequence

```
(B)x2, (C)x2, (B)x1, (D)x3, (E)x1, (#1 (2))x2, (D)x1, (E)x2, (B)x1, (#4 (2))x3, (C)x2, (B)x1, (C)x2
```

Listing 3.11: Best compression

```
(B)x2, (C)x2, (B)x1, (D)x3, (E)x1, (#1 (2))x2, (D)x1, (E)x2, (#4 (2))x4, (B)x1, (C)x2
```

In the end, after fully crunching the $executionTraces.txt$, a JSON file is generated as it can be seen in Listing 3.12. In this example, three functionalities were discovered. The first one, $ReadingController.startReadingFromInter$, only has one trace while the remaining ones have two. As formerly mentioned, each functionality has a field t denoting their list of unique[22] trace objects. Each trace object has a unique identifier portrayed by the id field as well as the a field holding the compressed list of accesses. Each element of this list, a RTE, is serialized as a list whose content may change as follows:

- if the RTE holds an $Access$, then the numeric identifier of the domain entity is appended to the list after the access type, read ($"R"$) or write ($"W"$);
- if the RTE holds a $Rule$, then the amount of subsequent elements that belong to the same sequence is appended to the list;
- if the element held by the RTE is repeated more than one time, then this number is added at the end of the list;

For instance, the $ReadingController.startReadingFromInter$ functionality has a $Rule$, serialized as $[5, 2]$, that means the sequence of five elements next to it occurs two times. If the rule was decompressed, these six elements (rule plus next five consecutive elements) would produce 10 access elements. The same rationale is also applied to some of those access elements that possess a third element, for example $["R", 71, 2]$. The decompressed size of those six elements would be 26, almost eight times more than the compressed size just on this short example.

[22]Uniqueness is ensured by comparing the compressed list of accesses of a new trace, for the same functionality, against the list of the traces already discovered.

Listing 3.12: Excerpt of a JSON file

```
{
  "ReadingController.startReadingFromInter": {
    "t": [
      {
        "id": 0,
        "a": [["R",30,2],["R",20,5],[5,2],["R",71,2],["R",20,2],["R",71],["R"
            ,52,6],["R",70,2],["R",71],["R",57],["R",17,4]]
      },
    ]
  },
  "EditionController.getEditionTableOfContentsbyAcronym": {
    "t": [
      {
        "id": 0,
        "a": [["R",30,2],["R",16,5],["R",30],["R",70,3]]
      },
      {
        "id": 1,
        "a": [["R",30,2],["R",16,3]]
      }
    ]
  },
  "FragmentController.getFragment": {
    "t": [
      {
        "id": 0,
        "a": [["R",30,2],["R",20,4],["R",30],["R",20]]
      },
      {
        "id": 1,
        "a": [["R",30,2],["R",20,4]]
      }
    ]
  }
}
```

3.4 Implementation Optimisations

This section is dedicated to the work done in order to optimise the Mono2Micro analysis framework. Despite the crucial role of compressing traces when generating the JSON file, the amount of dynamic collected data was still significant to the point that its analysis became the bottleneck. Both time and memory resources were compromised. The worst observed case was a total of 42497 unique traces while some were reaching (compacted) sizes between five to six million elements. The time spent analysing the JSON file was ranging from 20 minutes to a couple of hours, an unpleasant slowness for personal use. The showstopper became the memory footprint since time was the least of concerns given the investigation purpose. All efforts were focused on the goal of allowing the analysis on a personal machine with at least 16 GB of Random-access Memory (RAM). Therefore, to preserve these resources, a few measures were put to practice.

3.5 Time

To decrease the amount of time taken during analysis, the following major changes were applied:

- Iterable data structures, such as $List$[23], were converted to $HashMaps$[24]. A $HashMap$ is a hashtable-based implementation class in which key-value pairs can be added. It is expected constant-time performance $O(1)$ for most operations like $add()$, $remove()$ and $contains()$. For instance, considering the function $Decomposition.getClusterByEntity(entityName)$ that had to iterate over the list of clusters of a Decomposition in linear time and was being called inside loops within loops, now it is expected to run instantly because the class $Decomposition$ holds an $HashMap$ responsible for mapping an entity to the respective cluster. Other example was the function $Cluster.containsEntity(entityName)$ that, by using a list of entity names, in the worst case, had a complexity of $O(n \cdot m)$ being n the number of entities of a cluster and m the entity name string size. Since its execution was taking place inside three nested loops, an enhancement became imperative. Now, its complexity is approximately $O(1)$ because it is backed by a $HashSet$[25] containing instead numeric identifiers for the entities.

- Refactored functions that can be executed in the same cycles. For example, the calculus of complexity and coupling of a cluster, implies that all controllers that access a cluster must be traversed. Instead of traversing those controllers twice to calculate each metric, now both metrics are calculated in the same loop.

- Decreased the number of String comparisons by turning almost every String object into a primitive number type[26]: $byte$, $short$ or int.

3.6 Memory

Memory consumption became an issue when analysing JSON files with size in the order of hundreds of megabytes until a few gigabytes (around 2 GB). This was mostly caused by keeping in memory too much information at once. Consequently, the amount of required *RAM* was reduced as follows:

- The JSON file is now read by chunks instead of keeping it all in memory which turned out to be

[23]https://docs.oracle.com/javase/8/docs/api/java/util/List.html
[24]https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html
[25]`https://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html`
[26]https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

impractical. This was achieved by using Jackson's Streaming Api[27], a library that enables the parsing of JSON content.

- When calculating metrics, only one controller and its respective local transactions graph are kept in memory at a time. This graph is the data structure that requires the most memory since it depends directly on the size of traces. Before introducing this optimization, all controllers were being kept in memory, each one with its own graph, before calculating the metrics.
- During the Analyser's execution[28], after the analysis of each decomposition, the result is written immediately to the file system. Before, the results of each decomposition were being saved in memory and only in the end, after analysing all decompositions, the results were written to disk;
- Unique identifiable $Strings$ (e.g, entity names, access modes and cluster names) are now mapped to JAVA's primitive data types[29]: $byte/short/int$. These types only consume up to four bytes whereas e.g, an empty $String$ object occupies more than thirty bytes[30].

[27]https://github.com/FasterXML/jackson-docs/wiki/JacksonStreamingApi
[28]A special feature of the framework that is capable of analysing multiple decompositions generated with different weight combinations of similarity measures.
[29]https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html
[30]dependent on the underlying Java Virtual Machine (JVM) configuration

4 Evaluation

Contents

The goal of evaluation is to answer the research questions by (i) assessing which technique, static or dynamic, provides the best results and the effort of applying the latter; (ii) qualifying the impact of each similarity measure on the performance of generated decompositions; (iii) discovering possible correlations between the maintainability quality metrics and performance.

The source of data came from the two already mentioned monolith systems used in the previous work, LdoD (122 controllers, 71 domain entities) and Blended Workflow (98 controllers, 52 domain entities), that were subject of dynamic analysis.

Regarding the LdoD system, it was monitored in three different environments: production, functional testing and simulation. The production monitoring lasted 3 weeks and a total of 490 GB worth of data was collected. Throughout this period, a tight supervision was necessary to oversee the impact the monitoring had on the performance of the system's functionalities. Since the server hosting the application had a small free disk space (around 20 GB) and a massive drop in performance was observed if it was full, it was mandatory to collect the generated logs from time to time (2-3 days) to not harm the user experience and to gather fresh logs instead of discarding them.

Table 4.1: Dynamically collected data

	LdoD			**Blended Workflow**
	Production	**Tests**	**Simulation**	**Simulation**
Covered Entities (%)	79	82	80	86
Covered Controllers (%)	44	96	84	68

Analyzing the collected data presented in Table 4.1, only 44% of the controllers were exercised in production. Therefore, further processing and evaluation of this data were abdicated due to the substantial effort required to process it and the relatively little coverage. Concerning functional testing, it was achieved by running a suite of 200 integration tests (4.207 lines of code) that exercised 96% of the controllers and 82% of the domain entities, generating a few megabytes ($<$200 MB) of data, while the instruction coverage, reported by *JaCoCo*[1], was 72% for domain entities and 82% for controllers. The reduced size of the collected data is explained by the usage of small subset of the original database's data and so, the traces associated with the execution of functionalities were much shorter. Finally, an expert of the system simulated, during one hour, the use of functionalities, using a database with a minimal set of data, and 200 MB of data was collected and 84% of the controllers and 80% of the domain entities were exercised.

In what concerns the Blended Workflow system, it was only simulated by an expert during an hour and 86% of entities and 68% of controllers were exercised. The reduced number of exercised controllers is justified by the deprecation of several controllers that are not reachable through the user interface.

Similarly to how data was evaluated in the previous work regarding the static analysis, for each system, and respective environment, a JSON was generated, after collection, and then served as input

[1] https://github.com/jacoco/jacoco

to the analyser feature of Mono2Micro that creates several dendrograms by varying the weights of the four existing similarity measures - Access, Write, Read and Sequence - in intervals of 10 in a scale of 0 to 100. Then several cuts were performed on each one. Each cut resulted in a candidate decomposition of the monolith with a specific number of clusters, varying from 3 to 10. For each generated decomposition, the values for the quality metrics were calculated. The complexity metric value had to be normalized in order to compare them among the monoliths, since they depend on the number of functionalities of each monolith.

The uniform complexity of a given decomposition d of a monolith is calculated by dividing the complexity of d by the $maxComplexity$. The $maxComplexity$ value is determined by calculating the complexity of a decomposition of the monolith where each cluster has a single domain entity. Therefore, the uniform complexity of any monolith decomposition is a value in the interval 0 to 1.

Analogously, to assess the correlation between each metric and the weights given to each similarity measure and the number of clusters, a linear regression model was employed using the Ordinary Least Squares (OLS) method. For instance, Equation (4.1) demonstrates the linear regression model that correlates the complexity of a decomposition with the weights of each similarity measure.

$$\begin{aligned} uComplexity(d) = {} & \beta_1 \cdot d.weight_A + \beta_2 \cdot d.weight_W \\ & + \beta_3 \cdot d.weight_R + \beta_4 \cdot d.weight_S \\ & + \beta_5 \cdot \#d.clusters + cons \end{aligned} \tag{4.1}$$

To test this regression, a hypotheses was defined as follows:

- H_0: $\beta_1 = \beta_2 = \beta_3 = \beta_4 = \beta_5 = 0$; meaning that the complexity of a decomposition does not have a relation with any of the five parameters
- H_1: $\beta_1 \neq 0 \vee \beta_2 \neq 0 \vee \beta_3 \neq 0 \vee \beta_4 \neq 0 \vee \beta_5 \neq 0$; meaning that the complexity of a decomposition does have a relation with at least one of the five parameters

All the obtained regression results in the next sections show enough evidence to reject the null hypothesis at a significance level of 0.05.

Henceforth, to reduce the amount of space occupied by the results depicted in tables, a terminology was also adopted where:

- N means the number of clusters;
- A, W, R and S stand for Access, Write, Read and Sequence similarity measures respectively;
- R^2 represents the statistical measure that indicates the variability in the dependent variable of the regression model.

- $Coef.$ refers to the Regression Coefficient (β_i) obtained for a given dependent variable X_i. It represents the effect of the dependent variable on the independent variable (Y). If β_i is positive, then as X_i increases, the value of Y will also increase. If β_i is negative, then as X_i increases, the value of Y will decrease.
- $95\%\ Conf.\ Interval$ depicts the 95% Confidence Interval. As the name suggests, it ensures that one can assume with 95% of confidence that there is a statistically significant correlation explained by the respective coefficient if zero does not belong to the interval.

4.1 Static and Dynamic Analysis Comparison

This section solely focus on comparing the results of both analysis techniques. The statically collected data was obtained from an on-going thesis occurring in parallel with this work. To assess if one technique outperforms the other, two distinct evaluations were conducted:

- **Similarity measures evaluation**: To determine whether the dynamic analysis is able to discover a combination of similarity measures that provides better decompositions, in terms of complexity, coupling and cohesion and specifically compare the conclusions about complexity from the previous work that used static analysis.
- **Decompositions evaluation**: To assess whether the dynamic analysis produces significantly different decompositions when compared to the ones statically generated and with a source of truth.

4.1.1 Similarity measures evaluation

In this segment, the assessment of the impact of the similarity measures is first demonstrated against the complexity metric, then coupling and finally cohesion. In each case, a comparison between the results from the static and dynamic analysis is shown across both systems and respective environments with the goal of identifying a combination of similarity measures that excels in providing better decompositions for each quality metric as well as to understand if the results from the dynamic technique either corroborate or refute the ones obtained from its counterpart.

4.1.1.A Complexity

The regression results concerning the impact of the combination of the similarity measures and number of clusters on the complexity metric show that, according to Table 4.2 and Table 4.3, the dynamic and static analysis confirm the statistically significant positive correlation with complexity for the coefficients of the number of clusters. Regarding the similarity measures, there is no strong unanimity in the results.

For instance, for the access measure, the static analysis always reports a positive correlation whereas, depending on the environment, the dynamic analysis reports the two opposite behaviours. The write and sequence measures coefficients seem to produce decompositions with higher complexity.

The obtained R^2 values were considerably high with the exception of functional testing environment with just 0.176. This means that, apart from this specific environment, the regression model explains most of the data-set (low variability) and yet, it's still hard to conclude that one combination of similarity measures can provide better results in terms of complexity as some confidence intervals contain the zero value and the magnitude of the coefficients is not pronounced.

This confirms the conclusion achieved from the previous work that "there isn't a unique solution to the values of similarity, that leads to the best decomposition in terms of the complexity metric."

Table 4.2: Comparison of the impact of similarity measures on Complexity between both analysis on the LdoD system

	Static analysis		**Tests**		**Simulation**	
	Coef.	**95% Conf. Interval**	**Coef.**	**95% Conf. Interval**	**Coef.**	**95% Conf. Interval**
N	0.0230	[0.021, 0.025]	0.0253	[0.023, 0.028]	0.0206	[0.019, 0.023]
A	0.0035	[0.003, 0.004]	-0.0003	[-0.001, -9.14e-05]	0.0017	[0.002, 0.002]
W	0.0041	[0.004, 0.004]	2.781e-05	[-0.000, 0.000]	0.0079	[0.008, 0.008]
R	0.0039	[0.004, 0.004]	-0.0002	[-0.000, 7.17e-05]	0.0007	[0.001, 0.001]
S	-0.0002	[-0.000, 4.66e-05]	0.0002	[-2.19e-05, 0.000]	0.0018	[0.002, 0.002]
R^2	0.434		0.176		0.682	

Table 4.3: Comparison of the impact of similarity measures on Complexity between both analysis on the Blended Workflow system

	Static analysis		**Simulation**	
	Coef.	**95% Conf. Interval**	**Coef.**	**95% Conf. Interval**
N	0.0439	[0.043, 0.045]	0.0277	[0.026, 0.029]
A	0.0014	[0.001, 0.002]	-0.0011	[-0.001, -0.001]
W	0.0019	[0.002, 0.002]	0.0002	[-9.17e-08, 0.000]
R	0.0016	[0.001, 0.002]	-0.0013	[-0.001, -0.001]
S	0.0021	[0.002, 0.002]	0.0019	[0.002, 0.002]
R^2	0.632		0.476	

4.1.1.B Coupling

In regards to the repercussions of the similarity measures and number of clusters on coupling, the results depicted in Table 4.4 and Table 4.5 indicate a mutual corroboration between both analysis and systems in the sense that the access, read, write and sequence weights have a statistically significant positive correlation while the number of clusters has a statistically significant negative correlation with coupling. All linear regressions have a considerably high R^2 value, greater than 0.45. Despite the small dimension of the coefficients (< 0.01), this low variability increases the confidence of claiming that the access and read measures may be the best duo of similarity measures to generate loosely coupled decompositions.

Table 4.4: Comparison of the impact of similarity measures on Coupling between both analysis on the LdoD system

	Static analysis		**Tests**		**Simulation**	
	Coef.	**95% Conf. Interval**	**Coef.**	**95% Conf. Interval**	**Coef.**	**95% Conf. Interval**
N	-0.0212	[-0.022, 0.020]	-0.0318	[-0.033, -0.031]	-0.0277	[-0.029, -0.027]
A	0.0034	[0.003, 0.003]	0.0039	[0.004, 0.004]	0.0037	[0.004, 0.004]
W	0.0038	[0.004, 0.004]	0.0043	[0.004, 0.004]	0.0052	[0.005, 0.005]
R	0.0032	[0.003, 0.003]	0.0038	[0.004, 0.004]	0.0039	[0.004, 0.004]
S	0.0041	[0.004, 0.004]	0.0046	[0.004, 0.005]	0.0045	[0.004, 0.005]
R^2	0.634		0.694		0.667	

Table 4.5: Comparison of the impact of similarity measures on Coupling between both analysis on the Blended Workflow system

	Static analysis		**Simulation**	
	Coef.	**95% Conf. Interval**	**Coef.**	**95% Conf. Interval**
N	-0.0209	[-0.022, -0.020]	-0.0298	[-0.031, -0.029]
A	0.0049	[0.005, 0.005]	0.0047	[0.005, 0.005]
W	0.0039	[0.004, 0.004]	0.0056	[0.006, 0.006]
R	0.0047	[0.005, 0.005]	0.0045	[0.004, 0.005]
S	0.0054	[0.005, 0.006]	0.0059	[0.006, 0.006]
R^2	0.459		0.564	

4.1.1.C Cohesion

Finally, an assessment was performed with the purpose of correlating the similarity measures and number of clusters with the cohesion. The regression results for the LdoD and Blended Workflow systems, expressed in Table 4.6 and Table 4.7 respectively, suggest that, once again, both analysis support one another: the access, read, write and sequence measures have a statistically significant positive correlation with cohesion as well as the number of clusters. The majority of the regression models were able to explain around three quarters of the data-sets given the elevated R^2 values with the exception of the static analysis executed on the Blended Workflow system. Again, all similarity coefficients have considerably close and low values, however, it appears that the access and read measures may be the best combination of similarity measures to generate high cohesive decompositions while the write measure tends to provide the opposite outcome.

Table 4.6: Comparison of the impact of similarity measures on Cohesion between both analysis on the LdoD system

	Static analysis		**Tests**		**Simulation**	
	Coef.	**95% Conf. Interval**	**Coef.**	**95% Conf. Interval**	**Coef.**	**95% Conf. Interval**
N	0.0383	[0.037, 0.039]	0.0420	[0.041, 0.043]	0.0380	[0.037, 0.039]
A	0.0043	[0.004, 0.004]	0.0027	[0.003, 0.003]	0.0040	[0.004, 0.004]
W	0.0024	[0.002, 0.003]	0.0024	[0.002, 0.003]	0.0019	[0.002, 0.002]
R	0.0032	[0.003, 0.003]	0.0039	[0.004, 0.004]	0.0040	[0.004, 0.004]
S	0.0032	[0.003, 0.003]	0.0027	[0.003, 0.003]	0.0035	[0.003, 0.004]
R^2	0.720		0.727		0.727	

Table 4.7: Comparison of the impact of similarity measures on Cohesion between both analysis on the Blended Workflow system

	Static analysis		Simulation	
	Coef.	**95% Conf. Interval**	**Coef.**	**95% Conf. Interval**
N	0.0003	[0.014, 0.015]	0.0383	[0.037, 0.039]
A	0.0063	[0.006, 0.006]	0.0051	[0.005, 0.005]
W	0.0066	[0.007, 0.007]	0.0032	[0.003, 0.003]
R	0.0065	[0.006, 0.007]	0.0045	[0.004, 0.005]
S	0.0062	[0.006, 0.006]	0.0039	[0.004, 0.004]
R^2	0.366		0.768	

4.1.2 Decompositions evaluation

Although, it seems that the data collected dynamically provides similar insight in terms of the correlation between the similarity measures and the quality metrics, it was also assessed whether these techniques produce significantly different decompositions. To perform this analysis the collected data was firstly compared as observed in Table 4.8.

Table 4.8: Collected data comparison

	LdoD				Blended Workflow	
	Static	**Tests**	**Static**	**Simulation**	**Static**	**Simulation**
AVG(Covered Entities per Controller) (%)	95	71	91	77	93	78

As expected, all controllers were captured by static analysis as opposed to dynamic analysis according to Table 4.1. On the other hand, for the coverage of the accesses to domain entities in the context of the controllers, in some cases, the dynamic analysis could identify accesses to domain entities, in the context of a controller execution, that the static collection could not, due to late binding. And, of course, the opposite also occurs, because depending on the inputs provided to controllers and data available in the database, some of the domain entities may not be accessed, both in tests and simulation.

To assess the results of the static and dynamic analysis approaches, the highest quality decompositions are compared, in terms of complexity, from each approach with a decomposition proposed by a domain expert, for both systems. In this analysis the expert decompositions are considered as a reference that is used to evaluate, using the MoJoFM metric, which approach provides closer results to it. Since both techniques may miss some domain entities during the collection phase, it was decided that all the unassigned entities would be put in the biggest cluster, as this strategy conforms with the incremental decomposition strategy rationale [1, Chapter 13].

The results from the comparisons are represented in Table 4.9, where each cell indicates the MoJoFM value between the lowest complexity decomposition with N clusters, using the column's collection technique, and the system's expert decomposition. Overall, the MoJo values obtained for each collection approach were very similar, for both systems, which leads to the conclusion that there isn't a collection technique that provides better results. However, note that, in both environments, the dynamic analysis

Table 4.9: Comparison of the highest quality decompositions with expert decompositions

		LdoD			Blended Workflow	
		Static	**Tests**	**Simulation**	**Static**	**Simulation**
N	3	62.12%	65.15%	68.18%	46.67%	44.44%
	4	60.61%	69.7%	66.67%	44.44%	46.67%
	5	56.06%	68.18%	66.67%	44.44%	60%
	6	78.79%	66.67%	66.67%	62.22%	57.78%
	7	77.27%	74.24%	68.18%	66.67%	64.44%
	8	83.33%	72.73%	59.09%	66.67%	62.22%
	9	81.82%	74.24%	57.58%	71.11%	62.22%
	10	45.45%	74.24%	56.06%	71.11%	62.22%
avg		68.18%	70.64%	63.64%	59.17%	57.5%

did not cover all controllers during the collection phase and also missed more entities than the static approach as shown in Table 4.1 and Table 4.8. Therefore, it was decided to assess if the dynamic analysis approach would generate decompositions closer to the expert decomposition than the ones generated by the static approach if only the common controllers and entities were considered.

To evaluate this scenario, static analysis was re-ran considering only the common controllers and domain entities, for each dynamic environment. The results, depicted in Table 4.10, reveal that, on average, both approaches continue to generate decompositions almost equally distant to the expert's, for both systems. The major noticeable difference is the drop from 68.18% to 62.88% on the average MoJo values obtained for the static approach when *evened* with the dynamic analysis executed on the simulation environment. This suggests that the missed controllers by the dynamic collection during the simulation (84%) compared to functional testing (96%), may be crucial to find the couplings between the domain entities that lead to a decomposition closer to the expert's, highlighting the impact of the variability among different dynamic collection environments. In addition, the values of the simulation environment on the Blended Workflow system remained the same because it was the only case where dynamic analysis could not discover new entities compared to static analysis. Nonetheless, it did, in fact, discover entities on some controllers that its static counterpart could not, hence the 93% coverage proved in Table 4.8.

Table 4.10: Comparison of the highest quality decompositions with expert decompositions, considering only the common controllers and entities

		LdoD				Blended Workflow	
		Static	**Tests**	**Static**	**Simulation**	**Static**	**Simulation**
N	3	65.15%	59.09%	63.64%	71.21%	46.67%	44.44%
	4	51.52%	69.7%	62.12%	71.21%	51.11%	46.67%
	5	72.73%	68.18%	63.64%	66.67%	53.33%	60%
	6	72.73%	54.55%	68.18%	66.67%	51.11%	57.78%
	7	75.76%	74.24%	63.64%	69.7%	68.89%	64.44%
	8	74.24%	72.73%	68.18%	59.09%	66.67%	62.22%
	9	72.73%	74.24%	57.58%	57.58%	68.89%	62.22%
	10	68.18%	72.73%	56.06%	56.06%	77.78%	62.22%
avg		69.13%	68.18%	62.88%	64.77%	60.56%	57.5%

Based on these results, for both systems, there are no significant differences between the lowest complexity decompositions obtained using statically and dynamically collected data and that none of the approaches achieve identical decompositions to the expert's, since the average MoJo values obtained vary around 55-70%.

Given the resemblance when compared to the expert, it was then assessed how far apart the static and dynamic decompositions were from each other, considering the common controllers and entities.

Table 4.11: Comparing static with dynamic decompositions, considering only the common controllers and entities

		LdoD		**Blended Workflow**
		Static and Tests	**Static and Simulation**	**Static and Simulation**
N	3	57.41%	84.62%	83.33%
	4	83.02%	82.35%	63.41%
	5	78.85%	80.39%	50%
	6	78.85%	78%	58.97%
	7	78.85%	76%	57.89%
	8	82.69%	46.94%	50%
	9	80.77%	48.98%	50%
	10	60%	42.86%	37.84%
avg		75.06%	67.52%	56.43%

Considering Table 4.11, for LdoD, the average MoJo between the *evened* static and tests approaches was 75%, while between the *evened* static and simulation approaches was 67%. For Blended Workflow, the average MoJo between the evened static and simulation approaches was 56%. This leads to the conclusion that, on average, although equally distant to the expert's decomposition, the static and dynamic approaches do not generate similar decompositions, which suggests that there is space for future research on these differences even when neither of the analysis techniques surpasses the other, according to the previous results.

4.2 Performance evaluation

This section is, instead, devoted to (1) understand the impact of each similarity measure in the performance of the generated decompositions and (2) apprehend potential correlations between the maintainability and performance metrics. and consequently provides an answer the last two research questions introduced in Section 1.2. The obtained results were exclusively produced from data collected with dynamic analysis.

The introduced performance metric calculates the number of hops between microservices during a distributed transaction, in other terms, the network communication latency of a distributed functionality. Leveraging the formalisation presented in Section 2.1, the performance of a functionality for a given decomposition is expressed as the average of the performance of its traces:

$$performance(f, D) = \frac{\sum_{t_f \in traces(partition(G_f, D))} performance(t_f) - 1}{\#traces(partition(G_f, D))} \tag{4.2}$$

where $traces(partition(G_f, D)$ denotes the unique sequences of local transactions in the partition call graph of functionality f, and $performance(t_f)$ denotes the number of local transactions in the sequence. Since the goal is to count the number of remote invocations between local transactions, minus one must be subtract. This subtraction assumes that each trace in the partition call graph has a at least one local transaction.

The performance of a decomposition is the average of the performance of its functionalities:

$$performance(D) = \frac{\sum_{f \in F} performance(f, D)}{\#F} \tag{4.3}$$

where $performance(f, D)$ corresponds to the performance of a functionality f in the context of a decomposition D.

The values obtained for this metric also had to be normalized given the same rationale applied to the complexity metric. By analogy, to assess the correlation between this metric and the weights given to each similarity measure and the number of clusters, the same linear regression model was employed as demonstrated in Equation (4.1)

4.2.1 The impact of Similarity Measures on Performance

To evaluate the impact of each similarity measure over performance, the same strategy was applied as in Section 4.1.1. The results, exhibited in Table 4.12, show a comparison across both systems and respective environments to determine a combination of similarity measures that provides low latency decompositions. As expected, there is a consensus indicating that the number of clusters have a statistically significant positive correlation with performance. A higher number of clusters/microservice candidates is presumed to increase the network communication latency of a functionality. However, in terms of similarity measures, there is no robust conclusion given that some confidence intervals include the zero value, results from different environments/systems disagree on the sign of the coefficient as well as the coefficient values are very close to zero (< 0.01) indicating a very weak correlation.

Table 4.12: Comparison of the impact of similarity measures on Performance

	LdoD Tests		**LdoD Simulation**		**Blended Workflow simulation**	
	Coef.	**95% Conf. Interval**	**Coef.**	**95% Conf. Interval**	**Coef.**	**95% Conf. Interval**
N	0.0207	[0.018, 0.023]	0.0088	[0.007, 0.01]	0.0223	[0.021, 0.024]
A	-0.0003	[-0.001, -8.24e-05]	0.0017	[0.001, 0.002]	-0.0008	[-0.001, -0.001]
W	-0.0004	[-0.001, -0.000]	0.0082	[0.008, 0.008]	0.0002	[5e-05, 0.000]
R	-0.0001	[-0.000, 0.000]	0.0002	[-3.33e-05, 0.000]	-0.0009	[-0.001, -0.001]
S	-0.0002	[-0.000, 6.46e-05]	0.0012	[0.001, 0.001]	0.0016	[0.001, 0.002]
R^2	0.129		0.640		0.460	

4.2.2 Correlation between Performance and Maintainability quality metrics

The regression results from Table 4.13 and Table 4.14 show that both complexity and coupling metrics have a statistically significant positive correlation, with considerably low variability ($R^2 > 0.6$), with performance which allows to conclude that decompositions with higher network latency tend to have worse values regarding the two maintainability metrics. This correlation is expected since decompositions with a high network latency tend to have a high number of distributed transactions for each functionality and inevitably more remote invocations between clusters. Note that, the correlation between latency and complexity is much stronger than with coupling, as can be observed by the magnitude of the coefficients that show almost a direct proportion with complexity.

Table 4.13: Comparison of the impact of Performance in Complexity

	LdoD Tests		**LdoD Simulation**		**Blended Workflow simulation**	
	Coef.	**95% Conf. Interval**	**Coef.**	**95% Conf. Interval**	**Coef.**	**95% Conf. Interval**
Performance	1.0014	[0.992, 1.011]	0.8680	[0.861, 0.875]	1.1896	[1.182, 1.198]
$\boldsymbol{R^2}$	0.955		0.962		0.980	

Table 4.14: Comparison of the impact of Performance in Coupling

	LdoD Tests		**LdoD Simulation**		**Blended Workflow simulation**	
	Coef.	**95% Conf. Interval**	**Coef.**	**95% Conf. Interval**	**Coef.**	**95% Conf. Interval**
Performance	0.1751	[0.160, 0.191]	0.1047	[0.094, 0.115]	0.5344	[0.512, 0.557]
$\boldsymbol{R^2}$	0.722		0.605		0.727	

Concerning cohesion, as shown in Table 4.15, the performance metric shows a negative correlation on both simulation environments whereas on the functional testing executed on the LdoD system, a correlation can not be established given the inclusion of 0 in the confidence interval. This lack of unanimity is, to some extend, expected because the number of remote invocations of a functionality is not directly correlated with the cohesion, a functionality does not need to access the entities of all clusters to have low latency.

Table 4.15: Comparison of the impact of Performance in Cohesion

	LdoD Tests		**LdoD Simulation**		**Blended Workflow simulation**	
	Coef.	**95% Conf. Interval**	**Coef.**	**95% Conf. Interval**	**Coef.**	**95% Conf. Interval**
Performance	0.0201	[-0.004, 0.045]	-0.1766	[-0.190, -0.164]	-0.2576	[-0.283, -0.233]
$\boldsymbol{R^2}$	0.628		0.672		0.704	

4.3 Lessons learned

Returning to the research questions defined in Section 1.2:

RQ1: Does the information obtained using a dynamic code analyser, for the same set of similarity measures, provide better results, in terms of the quality of the generated decompositions, when compared with the static information captured on the previous work?

RQ2: Does a particular similarity measure provide better results in terms of the performance of the generated decompositions?

RQ3: Are the used maintainability quality metrics correlated with performance?

It is now possible to provide a plausible answer to each one them. Starting on the first research question, it was concluded that, through the comparison of an expert decomposition with the ones generated using data from each analysis technique, they do not outperform each other. Moreover, the decompositions statically and dynamically generated were further compared together and the results show that they have significant differences despite having the same disparity degree towards the expert one. Finally, a study was also conducted on the impact of similarity measures over the quality metrics with the usage of dynamic analysis. No specific combination of similarity measures determines the generation of the best decomposition according to any of the three maintainability quality metrics.

The answer for the second research question follows the same conclusion as the first given that no robust result indicate the existence of a possible combination of similarity measures that could generate low latency decompositions.

Last but not least, this work was able to correctly calculate and correlate the introduced performance metric with the already existent maintainability metrics, thus constituting the answer to the third research question.

While addressing the research questions, other notable lessons emerged:

- The combination of the similarity measure values for the best decomposition according to the quality metrics may be dependent on specific characteristics of the monolith system. As such, the best approach is to run the analyser feature to find the best combinations for each monolith;
- The effort to collect data dynamically is significantly superior than the static collection, specially when collecting and evaluating data from production which resulted in a very low coverage. On the other hand, the use of integration tests, which achieved better coverage, has a higher development cost, because, contrary to unit tests that aim to have 100% coverage, integration tests, are harder to maintain since they are designed to verify the modules integration, not the execution of all paths;

5 Related Work

This chapter presents the most recent literature that was reviewed in order to establish a confident cornerstone for this work.

In recent years, a myriad of approaches to support the migration of monolith systems to microservices architectures have been proposed [3, 16–24], which use the monolith specification, codebase, services interfaces, run-time behavior, and project development data to recommend the best decompositions [25].

This work intends to address the approaches that use the monolith codebase and/or run-time behavior. Although they follow the same steps, they diverge on what is their main concern and, consequently, on the similarity measures that they use, such as accesses [5], reads [2, 4], writes [2, 4], and sequences [2]. On the other hand, some authors use execution traces to collect the behavior of the monolith, e.g. [5, 24], but there is no empirical evidence on whether it provides better data than the static mechanisms, and what is the required effort to collect the data. Recent work on the migration of microservices also integrates static and dynamic analysis techniques [7, 26], by complementing the data collected through static analysis with dynamic analysis collected data. So far, there is no work that evaluates or discusses the quality of data obtained with the use of static and dynamic analysis in the migration of monolith systems to a microservices architectures.

In [27], it is introduced an approach, DeMIMA, that semi-automatically identifies architecture design patterns (authors call them ”design motifs”) from source code and consequently increases developers program comprehension. This is achieved by creating an abstract model of the program by using a combination of static and dynamic analysis to calculate four language-independent properties - exclusivity, receiver type, lifetime and multiplicity - which allow to identify commonly UML relationships like association, aggregation, and composition that are used to describe design patterns. After the program's model is built, DeMIMA uses a constraint satisfaction solver to find the solution of the design motif identification problem where the variables are the elements of the design motif model which domain comprehends the entities of the program's model and the variables constraints are the relationships between the elements of the design motif. In regards to static and dynamic analysis, there is, unfortunately, little detailed information about their *modus operandi*.

The article of Jin *et al*. [5] proposes a framework to identify service candidates (including entity and interface identification) from a monolithic system through the extraction and processing of its execution traces and also provides a systematic way to measure quantitatively and consistently the independence of a functionality, modularity and independence of evolvability of the identified services candidates by using an evaluation suite composed of 8 metrics. Their framework is composed of three steps: Firstly, they use Kieker as the dynamic code analyser, the same tool that was used in this work. Executions traces are collected by using a pre-defined test suite and then a representative execution trace is extracted among the others thus ignoring methods invocation frequency which can be an important information to retain for other possible metrics. Secondly, they identify and group class entities by generating functional atoms (a minimal coherent unit, in which all the entities are responsible for the same functional logic) using a hierarchical clustering algorithm as we do. After the discovery of all functional atoms, they use a functional atom grouping algorithm, which maximizes the intra-connectivity inside service candidates (cohesion) while minimizes the inter-connectivity across candidate boundaries (coupling), such that each resulting group will constitute the set of entities of a service candidate. Lastly, to identify the interface of a service candidate, they pick the first method that appears on each execution trace and associate it with the respective owner class and that constitutes an interface method. Regarding their evaluation suite to assess the quality of service candidates, it's very likely that we might use their metrics related to the independence of functionality to evaluate and support our work because it was the first time that a dynamic analysis was integrated in the Mono2Micro project. For instance, the metric *cohesion at message level* that measures the cohesiveness of interfaces published by a service, deals with the similarity of input messages (parameters) and output messages (return values) which can be better calculated with data collected only during a system's run-time execution (if inheritance is used, a static analysis can't distinguish subclass objects if it is expecting the respective superclass). With this metric we will be able to correlate it with the one that we are using - *Cohesion of a Cluster* - because service interfaces are the mean used for functionalities to access the cluster. By relying on black-box testing to derive execution traces, their results depend not only on the existence of a pre-defined test suite (not always available) but also on its associated coverage level.

Another recent effort towards the decomposition of monolithic system into a microservice architecture using execution traces is present on [28]. The authors reveal a six step framework to help software architects identify different decomposition candidates reducing the inherent process subjectivity by also providing three metrics to assess the decomposition quality. Their approach assumes the existence of detailed log trace collected at run-time (any entry point of a system, classes, methods and database accesses must be traced) and thus they do not delve into the data collection subject. They simply recommend the usage of an instrumentation tool called Elastic APM[1], in case run-time data isn't being collected yet, to enable logging generation at the method level with a very low effort. The most important steps of their framework are the first two where a process-mining tool (DISCO[2]) is used to identify what are the most frequently traversed execution paths by inspecting the log file. It graphically represents the call graph of the system according to the collected execution traces and shows, at the same time, the frequency that each path has been traversed, something that we are already doing but did not use in this work. The remaining steps suffer from the lack of automation and detail because: 1) no specification is given about how the common execution paths are clustered (they just mention the clustering in the conclusion without referring the concept in the respective step); 2) the third step, Removal of Circular

[1] https://www.elastic.co/solutions/apm
[2] https://fluxicon.com/disco/

Dependencies, is done manually and they only refer that an algorithm to discover cycles can be used without giving an academic background to support their claim and finally; 3) the identification and quality assessment (metric calculation) of decomposition candidates, steps four and five respectively, as well as the selection of the best decomposition, step 6, also lack automation because everything is done manually by experts through the visualization of the graph generated by a third party tool. The metrics they suggest are: Coupling between microservices, Number of classes per microservices and Number of classes that need to be duplicated. The coupling metric is defined by the division of the number of calls to external services used in each class belonging to the microservice by the total number of classes inside the microservice. The usage of the number of classes that need to be duplicated metric is justified with the premise that the more duplicated classes between microservices, the bigger the system size and consequently the harder will be its maintenance, but we believe that this metric can be further improved (or even better justified) because the existence of duplicated entities between microservices can imply changes on the consistency level of certain functionalities.

In [29], the authors propose a method to assist the extraction of functionality-oriented microservices by monitoring the dynamic behaviour of software and clustering entities with similar business logic from execution traces. They also used *Kieker*, as their DCA, on four *JAVA* monolithic systems which, in total, collected just 204 unique traces. To evaluate the obtained microservice candidates, five metrics related to cohesion and coupling were used systematically from the perspective of maintainability. The only adopted environment to generate data was through the design and execution of 204 test cases. In general, their proposed execution-oriented clustering method can obtain microservices with better functional independence in comparison with three state-of-the-art methods, resulting in more loosely coupled microservices. Despite the similarities between this approach and the one developed in this work, in the scope of Mono2Micro, their clustering uses the typical class and method hierarchical naming structure to define the microservices boundaries from the static information obtained on traces such as the *JAVA* package and, as such, completely discarding the notion of late binding which is the main advantage of executing a dynamic analysis. Additionally, our work is a novelty in terms of comparing the potential and effort of both code analysis techniques and they assume that “program behaviors cannot be explicitly reflected in source code”, but according to our results, it’s not possible to corroborate this assumption because neither of the analysis surpassed the other.

In [30], static analysis is used to complement the incompleteness of dynamic analysis, in order to increase programming comprehensibility of large systems. It proposes a two-phase hybrid clustering approach that groups entities (classes) by combining their behavioral characteristics and structural relationships. The similarity between system entities is identified by the number of the alleged *software features* while we use our own set of similarity measures intrinsically related to state consistency. A *software feature* is essentially a use-case scenario that, when exercised, will generate execution traces (no descriptive or visual representation is provided), similar to our controller definition. The authors do not explain how they process the collected traces which was an important part of this work. From these traces, classes/entities are extracted that will serve as an input to the hierarquical clustering algorithm. In the first phase, backed by dynamic analysis, a skeleton decomposition is created from the said traces. Until the skeleton is created more steps have to be completed of which two should be highlighted and adapted to Mono2Micro:(i) Clustering of Omnipresent Classes (ii) Removal of Singleton Clusters . Omnipresent Classes, considered as *software utilities* and used in a large number of subsystems, are removed to improve the clustering result and grouped into a utility cluster. The removal of singleton clusters, the ones containing just one single class, constitutes the last step where these classes are added to a pool to be used in a orphan adoption algorithm, in the second phase, that analyzes their static dependencies (naming conventions and structural relations) with the already formed clusters. The authors evaluate their approach only with a case study where they had to insert probes at the entry and exit of every method. It's not clear whether they opted by manual or automatic probe insertion and in which environment they exercised the dynamic analysis. Once again, both analysis are used without any scrutiny. The effort of applying dynamic analysis and the impact that this analysis may have on the run-time performance of the analysed system are completely discarded.

In [26], an approach is proposed to migrate monolithic systems into microservices where service boundaries are primarily defined with data collected from static analysis and subsequently refined through the visualization of the system run-time behaviour. Overall, it's a non-automated approach which requires previous knowledge about the software's structure and behavior to derive an architectural view (use cases and domain contexts) of the system such that it can later be validated with statically collected data (source code packages) in order to identify potential ambiguities. Additionally, the authors just present one case study and do not execute an evaluation over their procedure. This work differs from [30] since the first uses dynamically collected data to refine the data that was collected with static analysis and the other does the inverse process.

In [7] it's described a systematic and automatic approach to refactor monolithic systems, based on the Django web framework, to a microservice architecture that combines the results from static and dynamic code analyses. First, static analysis, relies on the system's source code, is employed to create a graphical model of the system where nodes can be classes, packages and modules and edges the import statements and method calls. Each edge is assigned a weight to represent the strength of the respective dependency. This weight is calculated through the sum of the number of imports and method calls between two nodes. Then, the system's operational behaviour is profiled to gather information - executed API endpoints, database queries and class methods - to calculate a new weight for each edge of the graph based on the execution frequency of each method. The final edge weight is a function of the static and dynamic weights. This graph is then used as input into a clustering algorithm that groups nodes according to the weights of the edges to form the microservice candidates. Despite this insightful and novel approach, only a qualitative evaluation is presented to perceive the impact of the combination of both analyses in comparison with the static analysis provided by a different tool, Service Cutter[3]. Moreover, just one system was studied and the evaluated quality was subjectively achieved by questioning three developers responsible for that system. Finally, contrary to this work, no explanation is provided about how (much) data was processed after one week of monitoring and no analysis was executed over coverage percentage.

A recent approach described in [24], proposes an automated framework to extract microservices from legacy monolithic systems only requiring the executable file. Execution and performance logs are the data artifacts obtained through dynamic and performance (CPU and Memory) analysis executed by Kieker and Java VisualVM[4]. Their framework processes the execution logs to identify controller objects (*COs*) and subordinate objects (SOs) (entities) in a similar manner as described in Section 3.3). Then a matrix is created that associates a quantified relation to each pair $<$*CO*, *SOs*$>$. This relation is calculated based on the established functional metrics Invocation Strength and Invocation Frequency to create better high-cohesive-low-coupled microservices. These metrics that work as similarity measures, as opposed to ours, our solely focused on frequency (performance) while ours are focused on aggregating entities according to a specific change in state consistency (read, write, sequence). *Invocation Strength* is a constant that attributes a weight to the type of execution call. They consider three types of operations: CREATE, UPDATE and READ. Each type has its own weight while Mono2Micro does not yet associates a different weight to write and read operations neither differentiates between create and update operations. The Invocation Frequency is another constant that depends on the same operation type but it's also influenced by the number of invocations for the respective operation. After calculating the functional matrix, the performance values of CPU and Memory are matched with the respective entries. Then, a Genetic algorithm is used on the matrix to identify microservices according to specific heuristics: CPU and memory consumption and the aforementioned functional metrics). Finally, they evaluated their framework by designing and executing tests cases without considering the impact that coverage might have on the achieved results as well as no insight was provided about how they coped with the amount of data collected.

[3]https://servicecutter.github.io/
[4]https://visualvm.github.io/

6 Discussion

Contents

In this chapter it is discussed what was achieved to benefit the software engineering community, specifically in the context of the Mono2Micro project towards the migration of monolith systems to microservices architectures according to transactional contexts. Moreover, threats to the validity of this work are also presented as well as some matters that may be addressed in subsequent researches.

6.1 Accomplishments

Alongside the effort of providing an answer to each of the three research questions, other noteworthy collateral accomplishments arose, namely:

- Dynamic analysis was introduced as a new collection technique through the usage of a highly customizable and extensible tool;
- Mono2Micro's analysis framework was heavily optimized with the purpose of processing much larger quantities of data in the shortest time possible;
- A new performance metric was added in the framework to provide an additional perspective to software architects when evaluating the quality of generated decompositions;
- Cooperation with an external researcher from the area of dynamic analysis, specifically in the performance analysis of software systems, in order to apply his feasible sequence compression algorithm on the collected traces.

6.2 Threats to Validity

However, the obtained results and conclusions on the evaluation (Chapter 4) aren't flawless. Their validity is still threatened both internally and externally.

6.2.1 Internal Validity

Since dynamic analysis adds an extra layer of computation on top of the monitored systems run-time behaviour, the assumptions made on the instrumentation, to minimize the performance degradation perceived by end-users, are a clear bias on the obtained results given that: (i) an iterable object type is considered to be the type of the first element and (ii) new records are discarded when Kieker's queue is full .

The approach of placing the entities not found during the collection process into the biggest cluster, when comparing the static and dynamic decompositions with the expert's, may have biased the results, as there is a probability associated with the expert decomposition that may or may not contain those entities in the same cluster. However, comparisons were made using other approaches and achieved similar results, thus, there is enough confidence in discarding this as a threat.

6.2.2 External Validity

Due to the effort associated with the dynamic collection of data, only two systems were analysed, but from the comparison with the decompositions generated from statically collected data, it's possible to infer that the quality of one decomposition does not outperforms the other, though the dynamic analysis of more monoliths is necessary. Nevertheless, the conclusions about the incompleteness of data and required effort associated with the dynamic collection of data are evident and show that a cost/benefit relation may tend for the static analysis approaches.

6.3 Future work

As a consequence of the results of this thesis and the learned lessons, the following topics are suggested as future work:

- Experiment with machine learning techniques using the metrics as fitness functions to infer which combination of similarity measures is more adequate for a concrete monolith system.
- Further explore the results of the dynamic collection of data, in terms of the frequency of each of the functionalities, and define new similarity measures to verify if it can generate better decompositions as well as new metrics that can provide
- Analyse more monolith systems with different technology stacks using dynamic analysis to reduce the existing bias on the regression models.
- Investigate other sequence compression algorithms with the purpose of decreasing the JSON file size and also the time the Mono2Micro framework takes to process it.

6.4 Data availability

The data used and produced in this research is available at `https://github.com/socialsoftware/mono2micro/tree/master/data/dynamic`

7

Conclusions

The migration of monolith systems to the microservices architecture is a complex problem that software development teams have to address when systems become more complex and larger in scale. Therefore, it is necessary to develop the methods and tools that help and guide them on the migration process. One of the most challenging problems is the identification of microservices. Several approaches have been proposed to automate such identification, which, although following the same steps, use different monolith analysis techniques, similarity measures, and metrics to evaluate the quality of the system.

In this thesis, two monolith systems were analysed to study the impact of applying static and dynamic analysis on the quality of the automatically generated decompositions as well as whether a particular combination of similarity measures provides better decompositions.

From the results of this research, it was concluded that there is no particular similarity measure that generates the best decompositions, considering the quality metrics for complexity, coupling, cohesion, and performance and a hypothesis is raised that it may depend on particular characteristics of the monolith. Therefore, as a suggestion, the analyzer feature of the Mono2Micro's framework can be used to generate decompositions using an extensive combination of similarity measures to find the one that has the best quality.

Moreover, as result of the executed analysis, it was also concluded that the different monolith analysis techniques generate decompositions that do not outperform each other, but, it was clear that the effort required by the dynamic analysis is much superior and resulted in less coverage. Although the cost is much higher, both systems were extensively dynamically analyzed, and compared with the static analysis, which is a unique effort with no precedents. Future experiments can be done, but this work already contains, in this aspect, an important and novel contribution.

Lastly, a new metric concerning performance was also added to current suite of metrics of Mono2Micro which was proved to be correctly correlated with the already existent maintainability ones. This addition is unequivocally out of the context of Mono2Micro, which is focused on transactional contexts, nonetheless it also plays a key role in a microservices architecture.

A new performance metric was added in the framework to provide an additional perspective to software architects when evaluating the quality of generated decompositions;

As additional contributions, (i) the gathered data from the evaluated monolith systems, using both static and dynamic analysis, is publicly available and can be used by third parties to do further research; (ii) the current analysis framework is now prepared to feasibly process larger amounts of data; (iii) the dynamic data collection was implemented to be as configurable and extensible as possible such that it can handle a wider variety of code bases with different technology stacks that are built using Java in a long-term view .

Bibliography

[1] C. Richardson, *Microservices Patterns: With Examples in Java*. Manning Publications, 2019. [Online]. Available: https://books.google.pt/books?id=UeK1swEACAAJ

[2] M. J. Amiri, "Object-aware identification of microservices," in *2018 IEEE International Conference on Services Computing (SCC)*, 2018, pp. 253–256.

[3] J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner, "From monolith to microservices: A classification of refactoring approaches," in *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, J.-M. Bruel, M. Mazzara, and B. Meyer, Eds. Cham: Springer International Publishing, 2019, pp. 128–141.

[4] S. Tyszberowicz, R. Heinrich, B. Liu, and Z. Liu, "Identifying microservices using functional decomposition," in *Dependable Software Engineering. Theories, Tools, and Applications*, X. Feng, M. Müller-Olm, and Z. Yang, Eds. Cham: Springer International Publishing, 2018, pp. 50–65.

[5] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, and Q. Zheng, "Service candidate identification from monolithic systems based on execution traces," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[6] A. Carrasco, B. v. Bladel, and S. Demeyer, "Migrating towards microservices: Migration and architecture smells," in *Proceedings of the 2nd International Workshop on Refactoring*, ser. IWoR 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 1–6. [Online]. Available: https://doi.org/10.1145/3242163.3242164

[7] T. Matias, F. F. Correia, J. Fritzsch, J. Bogner, H. S. Ferreira, and A. Restivo, "Determining Microservice Boundaries: A Case Study Using Static and Dynamic Software Analysis," *arXiv e-prints*, p. arXiv:2007.05948, Jul. 2020.

[8] L. F. A. Nunes, N. A. V. Santos, and A. R. Silva, "From a monolith to a microservices architecture: An approach based on transactional contexts," in *European Conference on Software Architecture (ECSA)*, ser. LNCS, vol. 11681. Springer International Publishing, Sep. 2019, pp. 37–52.

[9] N. Santos and A. Rito Silva, "A complexity metric for microservices architecture migration," in *2020 IEEE International Conference on Software Architecture (ICSA)*, 2020, pp. 169–178.

[10] Z. Wen and V. Tzerpos, "An effectiveness measure for software clustering algorithms," in *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.* IEEE, 2004, pp. 194–203.

[11] M. D. Ernst, "Static and dynamic analysis: Synergy and duality," in *WODA 2003: Workshop on Dynamic Analysis*, Portland, OR, USA, May 2003, pp. 24–27.

[12] W. Hasselbring and A. van Hoorn, "Kieker: A monitoring framework for software engineering research," *Software Impacts*, vol. 5, p. 100019, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2665963820300063

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP'97 — Object-Oriented Programming*, M. Akşit and S. Matsuoka, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 220–242.

[14] C. G. Nevill-Manning and I. H. Witten, "Identifying hierarchical structure in sequences: A linear-time algorithm," *J. Artif. Int. Res.*, vol. 7, no. 1, p. 67–82, Sep. 1997.

[15] D. G. Reichelt, S. Kühne, and W. Hasselbring, "Peass: A tool for identifying performance changes at code level," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1146–1149.

[16] M. Ahmadvand and A. Ibrahim, "Requirements reconciliation for scalable and secure microservice (de)composition," in *2016 IEEE 24th International Requirements Engineering Conference Workshops (REW)*, Sep. 2016, pp. 68–73.

[17] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann, "Service cutter: A systematic approach to service decomposition," in *Service-Oriented and Cloud Computing*, M. Aiello, E. B. Johnsen, S. Dustdar, and I. Georgievski, Eds. Cham: Springer International Publishing, 2016, pp. 185–200.

[18] S. Hassan and R. Bahsoon, "Microservices and their design trade-offs: A self-adaptive roadmap," in *2016 IEEE International Conference on Services Computing (SCC)*, June 2016, pp. 813–818.

[19] L. Baresi, M. Garriga, and A. De Renzis, "Microservices identification through interface analysis," in *Service-Oriented and Cloud Computing*, F. De Paoli, S. Schulte, and E. Broch Johnsen, Eds. Cham: Springer International Publishing, 2017, pp. 19–33.

[20] S. Klock, J. M. E. M. V. D. Werf, J. P. Guelen, and S. Jansen, "Workload-based clustering of coherent feature sets in microservice architectures," in *2017 IEEE International Conference on Software Architecture (ICSA)*, April 2017, pp. 11–20.

[21] G. Mazlami, J. Cito, and P. Leitner, "Extraction of microservices from monolithic software architectures," in *Web Services (ICWS), 2017 IEEE International Conference on*. IEEE, 2017, pp. 524–531.

[22] L. De Lauretis, "From monolithic architecture to microservices architecture," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2019, pp. 93–96.

[23] A. Selmadji, A. Seriai, H. L. Bouziane, R. Oumarou Mahamane, P. Zaragoza, and C. Dony, "From monolithic architecture style to microservice one based on a semi-automatic approach," in *2020 IEEE International Conference on Software Architecture (ICSA)*, 2020, pp. 157–168.

[24] Y. Zhang, B. Liu, L. Dai, K. Chen, and X. Cao, "Automated microservice identification in legacy systems with functional and non-functional metrics," in *2020 IEEE International Conference on Software Architecture (ICSA)*, 2020, pp. 135–145.

[25] F. Ponce, G. Márquez, and H. Astudillo, "Migrating from monolithic architecture to microservices: A rapid review," in *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, 2019, pp. 1–7.

[26] A. Krause, C. Zirkelbach, W. Hasselbring, S. Lenga, and D. Kroger, "Microservice decomposition via static and dynamic analysis of the monolith," in *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2020, pp. 9–16.

[27] Y.-G. Guéhéneuc and G. Antoniol, "Demima: A multilayered approach for design pattern identification," *IEEE Trans. Software Eng.*, vol. 34, pp. 667–684, 09 2008.

[28] D. Taibi and K. Systä, "From monolithic systems to microservices: A decomposition framework based on process mining," in *CLOSER 2019 - Proceedings of the 9th International Conference on Cloud Computing and Services Science*, D. Ferguson, V. Munoz, M. Helfert, and C. Pahl, Eds. SCITEPRESS, 2019, pp. 153–164.

[29] W. Jin, T. Liu, Q. Zheng, D. Cui, and Y. Cai, "Functionality-oriented microservice extraction based on execution trace clustering," in *2018 IEEE International Conference on Web Services (ICWS)*, 2018, pp. 211–218.

[30] C. Patel, A. Hamou-Lhadj, and J. Rilling, "Software clustering using dynamic analysis and static dependencies," in *2009 13th European Conference on Software Maintenance and Reengineering*, 2009, pp. 27–36.

A

Code of Project

Listing A.1: Class file containing the aspect's Joint points and Pointcuts. It specifies the methods in which the Advice should be applied. Import statements are omitted

```
/**
 * @author Bernardo Andrade
 */
@Aspect
public class GettersAndSetters extends AbstractOperationExecutionAspect {
    public GettersAndSetters() {}
    // ----------------------------------------- EXCEPTIONS ------------------------------------------
    @Pointcut("execution(pt.ist.fenixframework..* get*(..))")
    public void gettersReturningFenixFrameworkObject() {}

    @Pointcut("execution(* set*(pt.ist.fenixframework..*, ..))")
    public void settersWithFenixFrameworkObjectArgument() {}

    @Pointcut("!gettersReturningFenixFrameworkObject() && !settersWithFenixFrameworkObjectArgument()")
    public void noGettersOrSettersWithFenixFramework() {}
```

```
// -------------------------------------------- _BASE CLASSE METHODS
    --------------------------------------------
@Pointcut("execution(public void pt.ist..*.*_Base.add*(..)) && within(pt.ist..*.*_Base)")
public void publicBaseAddMethods() {}

@Pointcut("execution(public void pt.ist..*.*_Base.remove*(..)) && within(pt.ist..*.*_Base)")
public void publicBaseRemoveMethods() {}

@Pointcut("execution(public * pt.ist..*.*_Base.get*(..)) && within(pt.ist..*.*_Base)")
public void publicBaseGetMethods() {}

@Pointcut("execution(public void pt.ist..*.*_Base.set*(..)) && within(pt.ist..*.*_Base)")
public void publicBaseSetMethods() {}

@Pointcut("publicBaseAddMethods() || publicBaseRemoveMethods() || publicBaseGetMethods() ||
    publicBaseSetMethods()")
public void baseClassMethods() {}
//---------------------------------------------------------------- FENIX FRAMEWORK
    ----------------------------------------------------------------
@Pointcut("execution(public * pt.ist.fenixframework.FenixFramework.getDomainObject(..))")
public void fenixFrameworkGetDomainObject() {}

@Pointcut("execution(public * pt.ist.fenixframework.DomainRoot_Base.get*(..)) || execution(public *
     pt.ist.fenixframework.DomainRoot_Base.set*(..))")
public void gettersAndSettersInDomainRootBase() {}

@Pointcut("gettersAndSettersInDomainRootBase() || fenixFrameworkGetDomainObject()")
public void fenixFrameworkGettersAndSetters() {}
// ---------------------------- OTHER CLASSES (AND METHODS) THAT MAY BE IMPORTANT TO CATCH
    ----------------------------
@Pointcut("execution(public * pt.ist.fenixframework.backend.jvstmojb.pstm.AbstractDomainObject.
    getExternalId(..))")
public void fenixFrameworkAbstractDomainObjectGetExternalId() {}

@Pointcut("execution(protected * pt.ist.fenixframework.backend.jvstmojb.pstm.OneBoxDomainObject.
    deleteDomainObject(..))")
public void fenixFrameworkAbstractDomainObjectDeleteDomainObject() {}

@Pointcut("fenixFrameworkAbstractDomainObjectGetExternalId() ||
    fenixFrameworkAbstractDomainObjectDeleteDomainObject()")
public void otherMethodsThatMayBeImportant() {}
// ----------------------------------------------------------------- CONTROLLER CLASSES
    -----------------------------------------------------------------
@Pointcut("(@target(org.springframework.stereotype.Controller) || @target(org.springframework.web.
    bind.annotation.RestController)) && execution(* *(..))")
public void controllerClasses() {}
// --------------------------------------------------------------------------- CONTROLLER METHODS
    ---------------------------------------------------------------------------
// Controller GET methods
@Pointcut("(@annotation(org.springframework.web.bind.annotation.RequestMapping) || @annotation(org.
    springframework.web.bind.annotation.GetMapping)) && execution(* *(..))")
public void controllerGetMethods() {}

// Controller POST methods
@Pointcut("(@annotation(org.springframework.web.bind.annotation.RequestMapping) || @annotation(org.
    springframework.web.bind.annotation.PostMapping)) && execution(* *(..))")
public void controllerPostMethods() {}

// Controller PATCH methods
@Pointcut("@annotation(org.springframework.web.bind.annotation.PatchMapping) && execution(* *(..))"
    )
public void controllerPatchMethods() {}

// Controller PUT methods
@Pointcut("@annotation(org.springframework.web.bind.annotation.PutMapping) && execution(* *(..))")
public void controllerPutMethods() {}

// Controller DELETE methods
@Pointcut("@annotation(org.springframework.web.bind.annotation.DeleteMapping) && execution(* *(..))
    ")
public void controllerDeleteMethods() {}

// Controller MESSAGE methods
@Pointcut("@annotation(org.springframework.messaging.handler.annotation.MessageMapping) &&
    execution(* *(..))")
public void controllerMessageMethods() {}

@Pointcut("controllerGetMethods() || controllerPostMethods() || controllerPatchMethods() ||
    controllerPutMethods() || controllerDeleteMethods() || controllerMessageMethods()")
public void controllerMethods() {}
// -------------------------------------------------------------------- KIEKER METHOD
    --------------------------------------------------------------------
@Pointcut("(baseClassMethods() && noGettersOrSettersWithFenixFramework()) ||
    fenixFrameworkGettersAndSetters() || otherMethodsThatMayBeImportant()")
public void domainAndORMMethods() {}

@Pointcut("(cflow(controllerMethods()) && domainAndORMMethods()) || controllerMethods()")
public void monitoredOperation() {}
```

Listing A.2: Class file which defines the Advice responsible for gathering all the information to compose the Monitoring Record. Lines 42-123 were added to the original file such that new method information could be added to the record. Import statements are omitted

```java
1  /**
2   * @author Andre van Hoorn, Jan Waller, Bernardo Andrade
3   */
4  @Aspect
5  public abstract class AbstractOperationExecutionAspect extends AbstractAspectJProbe {
6      private static final Logger LOGGER = LoggerFactory.getLogger(AbstractOperationExecutionAspect.class
           );
7      private static final IMonitoringController CTRLINST = MonitoringController.getInstance();
8      private static final ITimeSource TIME = CTRLINST.getTimeSource();
9      private static final ControlFlowRegistry CFREGISTRY = ControlFlowRegistry.INSTANCE;
10
11     @Pointcut
12     public abstract void monitoredOperation();
13
14     @Around("monitoredOperation() && notWithinKieker()")
15     public Object operation(final ProceedingJoinPoint thisJoinPoint) throws Throwable {
16         if (!CTRLINST.isMonitoringEnabled()) {
17             return thisJoinPoint.proceed();
18         }
19         final boolean entrypoint;
20         final int eoi; // this is executionOrderIndex-th execution in this trace
21         final int ess; // this is the height in the dynamic call tree of this execution
22         long traceId = CFREGISTRY.recallThreadLocalTraceId(); // traceId, -1 if entry point
23         if (traceId == -1) {
24             entrypoint = true;
25             traceId = CFREGISTRY.getAndStoreUniqueThreadLocalTraceId();
26             CFREGISTRY.storeThreadLocalEOI(0);
27             CFREGISTRY.storeThreadLocalESS(1); // next operation is ess + 1
28             eoi = 0;
29             ess = 0;
30         } else {
31             entrypoint = false;
32             eoi = CFREGISTRY.incrementAndRecallThreadLocalEOI();
33             ess = CFREGISTRY.recallAndIncrementThreadLocalESS();
34             if ((eoi == -1) || (ess == -1)) {
35                 LOGGER.error("eoi and/or ess have invalid values: eoi == {} ess == {}", eoi, ess);
36                 CTRLINST.terminateMonitoring();
37             }
38         }
39         final long tin = TIME.getTime();
40         // execution of the called method
```

```
        Object retval = null;
        final StringBuilder newSignature = new StringBuilder(128);
        try {
            // 1) Getting the entity declaring type that executes the given method
            try {
                newSignature.append(thisJoinPoint.getSignature().getDeclaringType().getSimpleName());
            } catch (Exception e) {
                throw e;
            } finally {
                newSignature.append(":"); // Separator
            }
            // 2) Getting the method name
            try {
                newSignature.append(thisJoinPoint.getSignature().getName());
            } catch (Exception e) {
                throw e;
            } finally {
                newSignature.append(":"); // Separator
            }
            // 3) Getting the entity runtime type that executes the given method
            try {
                Object target = thisJoinPoint.getTarget();
                if (target != null) {
                    newSignature.append(thisJoinPoint.getTarget().getClass().getSimpleName());
                }
            } catch (Exception e) {
                throw e;
            } finally {
                newSignature.append(":"); // Separator
            }
            // 4) Getting the arguments types of the given method
            try {
                if (thisJoinPoint.getArgs().length > 0) {
                    Object[] args = thisJoinPoint.getArgs();
                    String argTypes = "[";
                    for (int i = 0; i < args.length; i++) {
                        if (args[i] == null) {
                            String argStaticType = ((MethodSignature) thisJoinPoint.getSignature())
                                                                        .getParameterTypes()[i]
                                                                        .getSimpleName();
                            argTypes += "\"" + argStaticType + "\"";
                        } else {
                            String argDynamicType = args[i].getClass().getSimpleName();
                            argTypes += "\"" + argDynamicType + "\"";
                        }
                        if (i < args.length - 1) {
                            argTypes += ",";
                        }
                    }
                    argTypes += "]";
                    newSignature.append(argTypes);
                }
            } catch (Exception e) {
                throw e;
            } finally {
                newSignature.append(":"); // Separator
            }
            // 5) Getting the returned value type of the given method
            try {
                retval = thisJoinPoint.proceed();
                if (retval != null) {
                    if (retval.getClass().getSimpleName().equals("RelationList")) {
                        RelationList retValAux = (RelationList) retval;
                        Iterator<?> i = retValAux.iterator();
                        if (i.hasNext()) {
                            Object firstElement = i.next();
                            if (firstElement != null) {
                                // ASSUMPTION: returned value dynamic type is the dynamic type of the
                                    first element of the list
                                newSignature.append(firstElement.getClass().getSimpleName());
                            }
                        }
                    } else {
                        newSignature.append(retval.getClass().getSimpleName());
                    }
                } else {
                    String returnedValueStaticType = ((MethodSignature) thisJoinPoint.getSignature())
                                                                                     .getReturnType()
                                                                                     .getSimpleName();
                    newSignature.append(returnedValueStaticType);
                }
            } catch (Exception e) {
                throw e;
            }
        } finally {
            final long tout = TIME.getTime();
            CTRLINST.newMonitoringRecord(
                new OperationExecutionRecord(
                    "",
                    newSignature.toString(),
```

```java
                    traceId,
                    tin,
                    tout,
                    "",
                    eoi,
                    ess
                )
            );

            if (entrypoint) {
                CFREGISTRY.unsetThreadLocalTraceId();
                CFREGISTRY.unsetThreadLocalEOI();
                CFREGISTRY.unsetThreadLocalESS();
            } else {
                CFREGISTRY.storeThreadLocalESS(ess);
            }
        }
        return retval;
    }
}
```

Listing A.3: Kieker's configuration file: kieker.monitoring.properties

```
## The location of the file is passed to Kieker.Monitoring via the JVM parameter
## kieker.monitoring.configuration. For example, with a configuration file named
## my.kieker.monitoring.properties in the folder META-INF you would pass this location
## to the JVM when starting your application:
##
##  java -Dkieker.monitoring.configuration=META-INF/my.kieker.monitoring.properties [...]
##
## If no configuration file is passed, Kieker tries to use a configuration file in
## META-INF/kieker.monitoring.properties
## If this also fails, a default configuration is being used according to the values in
## this default file.

## The name of the Kieker instance.
kieker.monitoring.name=Dummy

## Whether a debug mode is activated.
## This changes a few internal id generation mechanisms to enable
## easier debugging. Additionally, it is possible to enable debug
## logging in the settings of the used logger.
## This setting should usually not be set to true.
kieker.monitoring.debug=true

## Enable/disable monitoring after startup (true|false; default: true)
## If monitoring is disabled, the MonitoringController simply pauses.
## Furthermore, probes should stop collecting new data and monitoring
## writers stop should stop writing existing data.
kieker.monitoring.enabled=true

## The name of the VM running Kieker. If empty the name will be determined
## automatically, else it will be set to the given value.
kieker.monitoring.hostname=

## The initial ID associated with all experiments. (currently not used)
kieker.monitoring.initialExperimentId=1

## Automatically add a metadata record to the monitoring log when writing
## the first monitoring record. The metadata record contains infromation
## on the configuration of the monitoring controller.
kieker.monitoring.metadata=true

## Enables/disable the automatic assignment of each record's logging timestamp.
## (true|false; default: true)
kieker.monitoring.setLoggingTimestamp=true

## Whether a shutdown hook should be registered.
## This ensures that necessary cleanup steps are finished and no
## information is lost due to asynchronous writers.
## This should usually not be set to false.
kieker.monitoring.useShutdownHook=true

## Whether any JMX functionality is available
kieker.monitoring.jmx=false
kieker.monitoring.jmx.domain=kieker.monitoring

## Enable/Disable the MonitoringController MBean
```

```
kieker.monitoring.jmx.MonitoringController=true
kieker.monitoring.jmx.MonitoringController.name=MonitoringController

## The size of the thread pool used to execute registered periodic sensor jobs.
## The thread pool is also used to periodically read the config file for adaptive
## monitoring.
## Set to 0 to deactivate scheduling.
kieker.monitoring.periodicSensorsExecutorPoolSize=0

## Enable or disable adaptive monitoring.
kieker.monitoring.adaptiveMonitoring.enabled=false

## Default location of the adaptive monitoring configuration File
kieker.monitoring.adaptiveMonitoring.configFile=META-INF/kieker.monitoring.
    adaptiveMonitoring.conf

## Enable/disable the updating of the pattern file by activating or deactivating
## probes through the api.
kieker.monitoring.adaptiveMonitoring.updateConfigFile=false

## The delay in seconds in which the pattern file is checked for changes.
## Requires kieker.monitoring.periodicSensorsExecutorPoolSize > 0.
## Set to 0 to disable the observation.
kieker.monitoring.adaptiveMonitoring.readInterval=0

## The maximal size of the signature cache. This is a weak limit, as the cache can exceed
    this size
## slightly in practical application.
## Set to -1 for an unbounded cache.
kieker.monitoring.adaptiveMonitoring.maxCacheSize=-1

## The behaviour of the signature cache, if the maximal size is bounded.
## 0: The cache ignores entries once the maximal size is reached.
## 1: The cache removes a (semi)random entry from the cache once the maximal size is
    exceeded.
## 2: The cache is completely cleared once the maximal size is reached.
kieker.monitoring.adaptiveMonitoring.boundedCacheBehaviour=0

```

```
############################
#######   TIMER   #######
############################
## Selection of the timer used by Kieker (classname)
## The value must be a fully-qualified classname of a class implementing
## kieker.monitoring.timer.ITimeSource and providing a constructor that
## accepts a single Configuration.
kieker.monitoring.timer=kieker.monitoring.timer.SystemNanoTimer

##
#kieker.monitoring.timer=kieker.monitoring.timer.SystemMilliTimer
## A timer with millisecond precision.
## The offset of the timer. The time returned is since 1970-1-1
## minus this offset. If the offset is empty it is set to the current
## time.
## The offset must be specified in milliseconds.
kieker.monitoring.timer.SystemMilliTimer.offset=0

## The timeunit used to report the timestamp.
## Accepted values:
##  0 - nanoseconds
##  1 - microseconds
##  2 - milliseconds
##  3 - seconds
kieker.monitoring.timer.SystemMilliTimer.unit=0
##kieker.monitoring.timer=kieker.monitoring.timer.SystemNanoTimer

## A timer with nanosecond precision.
## The offset of the timer. The time returned is since 1970-1-1
## minus this offset. If the offset is empty it is set to the current
## time.
## The offset must be specified in milliseconds.
kieker.monitoring.timer.SystemNanoTimer.offset=0
## The timeunit used to report the timestamp.
## Accepted values:
##  0 - nanoseconds
##  1 - microseconds
##  2 - milliseconds
##  3 - seconds
kieker.monitoring.timer.SystemNanoTimer.unit=0

###########################
#######   WRITER  #######
###########################

## The internal synchronized queue implementation to use.
## It must provide a constructor with a single int parameter which represents the queue's (initial) capacity.
## java.util.concurrent.ArrayBlockingQueue
## java.util.concurrent.LinkedBlockingQueue
kieker.monitoring.core.controller.WriterController.RecordQueueFQN=org.jctools.queues.MpscArrayQueue

## This parameter defines the synchronized queue's (initial) capacity in terms of the number of records.
## Note that the actual capacity can be increased depending on the queue implementation.
kieker.monitoring.core.controller.WriterController.RecordQueueSize=10000000

## 0: terminate Monitoring with an error (default)
## 1: writer blocks until queue capacity is available
## 2: writer discards new records until space is available
## Be careful when using the value '1' since then, the asynchronous writer
## is no longer decoupled from the monitored application.
kieker.monitoring.core.controller.WriterController.RecordQueueInsertBehavior=1

## Selection of monitoring data writer (classname)
## The value must be a fully-qualified classname of a class implementing
## kieker.monitoring.writer.IMonitoringWriter and providing a constructor that
## accepts a single Configuration.
kieker.monitoring.writer=kieker.monitoring.writer.filesystem.FileWriter

## When flushing is disabled, it could require a lot of events before finally any writing to the map file is done.
## In case of long running observations, this is the desired behavior. However, in shorter experiments and in cases when the application crashes,
## it is helpful to ensure all map entries have been written as soon as possible.
## To force flushing on the the map file, set the following property to true.
kieker.monitoring.writer.filesystem.TextMapFileHandler.flush=true

## As mentioned before, the FileWriter writes a map of all strings to a map file.
## This is usually done by the TextMapFileHandler, which is the default.You may choose
    another map file handler.
kieker.monitoring.writer.filesystem.FileWriter.mapFileHandler=kieker.monitoring.writer.
    filesystem.TextMapFileHandler

## Log file pool handler manages when files are written, how they are named and when they
    are removed.
## The default RotatingLogFilePoolHandler which supports a upper limit of log files which
    are kept.
kieker.monitoring.writer.filesystem.FileWriter.logFilePoolHandler=kieker.monitoring.writer.
    filesystem.RotatingLogFilePoolHandler

## The log stream handler writes the text output.
## Default is the TextLogStreamHandler (text serialization in standard Kieker format)
## Alternatively, you may use the BinaryLogStreamHandler which serializes the data in binary
     format.
## You may want to write your own LogStreamHandler, e.g., to support JSON as output format.
kieker.monitoring.writer.filesystem.FileWriter.logStreamHandler=kieker.monitoring.writer.
    filesystem.TextLogStreamHandler

## The FileWriter uses UTF-8 as default char set. However, you may want to specify another
    charset with
kieker.monitoring.writer.filesystem.FileWriter.charsetName=UTF-8

## As host file systems have limits on file length and to avoid losing all data when the log
     is corrupted,
## you can limit the maximal number of entries (events) per created file. The value must be
    greater than zero.
kieker.monitoring.writer.filesystem.FileWriter.maxEntriesInFile=1000000

## The maximal file size of the generated monitoring log. Older files will be
## deleted if this file size is exceeded. Given in MiB.
## At least one file will always remain, regardless of size!
## Use -1 to ignore this functionality.
```

```
kieker.monitoring.writer.filesystem.FileWriter.maxLogSize=-1

## The maximal number of log files generated. Older files will be deleted if this number is
    exceeded.
## At least one file will always remain, regardless of size!
## Use -1 to ignore this functionality.
kieker.monitoring.writer.filesystem.FileWriter.maxLogFiles=-1

## In order to use a custom directory, set customStoragePath as desired. Examples:
## /var/kieker or C:\\KiekerData (ensure the folder exists).
## Otherwise the default temporary directory will be used
kieker.monitoring.writer.filesystem.FileWriter.customStoragePath=

## When compression is enabled, each log file is written as zipped binary file.
## Depending on the libraries used alongside Kieker, you can user ZIP, GZIP and XZ
    compression.
## The corresponding classes are:
## NoneCompressionFilter (no compression)
## GZipCompressionFilter
## XZCompressionFilter
## ZipCompressionFilter
kieker.monitoring.writer.filesystem.FileWriter.compression=pt.ist.socialsoftware.edition.
    ldod.compressors.ZipCompressor

## When flushing is disabled, it could require a lot of records before
## finally any writing is done.
kieker.monitoring.writer.filesystem.FileWriter.flush=false

## When flushing is disabled, records are buffered in memory before written.
## This setting configures the size of the used buffer in bytes.
## 512 Kilobytes
kieker.monitoring.writer.filesystem.FileWriter.bufferSize=524288
```