Tampere University

Alexander Bakhtin

MICROSERVICE API PATTERN DETECTION

using business processes and call graphs

Master of Science Thesis
Faculty of Information Technology and Communication Sciences
Examiners: prof. Davide Taibi
Xiaozhou Li, PhD
November 2022

ABSTRACT

Alexander Bakhtin: Microservice API pattern detection
Master of Science Thesis
Tampere University
Master's Programme in Computing Sciences
November 2022

It is well recognized that design patterns improve system development and maintenance in many aspects. While we commonly recognize these patterns in monolithic systems, many patterns emerged for cloud computing, specifically microservices. Unfortunately, while various patterns have been proposed, available quality assessment tools often do not recognize many. This thesis performs a grey literature review to find and catalog available tools to detect microservice API patterns. It reasons about mechanisms that can be used to detect these patterns. Furthermore, the results indicate gaps and opportunities for improvements for quality assessment tools. There are available tools commonly used by practitioners that offer centralized logging, tracing, and metric collection for microservices. We assess the opportunity to combine current dynamic analysis tools with anomaly detection in the form of patterns and anti-patterns. We develop a tool prototype that we use to assess a large microservice system benchmark demonstrating the feasibility and potential of such an approach.

Keywords: microservice, microservice patterns, business process, telemetry, dynamic analysis

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application Programming Interface
DB	Database
IHR	Information Holder Resource
MAP	Microservice API Pattern
MLR	Multivocal Literature Review
MSA	Microservice Architecture
OSS	Open Source Software
RQ	Research Question
SDG	Service Dependency Graph

1. INTRODUCTION

Design patterns provide a generalized and reusable solution to common software design problems. They indicate that a system uses best practices. Patterns are well-recognized to improve software quality [1]. One of the reasons behind such impact is better software comprehension related to software documentation. Detection of patterns is a complex task [2, 3]. It can constitute static or dynamic system analysis and serves the purpose of quality assurance and quality indices of system design.

With the era of cloud systems, we must assume new patterns emerge that constitute possibly across multiple self-contained parts of the overall system. In the current context, cloud computing is fueled by the microservice architecture [4]. Microservices are small and autonomous services deployed independently, with a single and clearly defined purpose [5, 6]. Microservice architecture (MSA) has become the industry standard. MSA allows for efficient scaling, improved resiliency, and faster software delivery because of its loosely connected nature. It also enables teams to focus on a single task/module and produce reliable software faster by allowing them to use any programming language that suits the task. With the ability of agility and speed of development, MSA needs constant monitoring and analysis to tackle the complexity of the system. A microservice system's longevity and quality can be jeopardized without regular monitoring. We ask what techniques and state-of-the-art tools can detect microservice patterns and the current gaps in this context. Quality engineers, architects, and developers might need to know underlying code quality details to improve maintenance and support system comprehension.

The system can be subjected to static and/or dynamic (runtime) analysis. The static analysis uses software artifacts such as source code, deployment manifests, and API documentation. Dynamic analysis is the real-time testing or profiling of a system. Dynamic analysis can be applied to runtime data from a system in either a production or a staging/development environment. Users must use the system, or a script must replicate real-time user behavior, such as accessing all use cases and making reasonable user requests per minute, to generate runtime data. Data obtained in runtime includes application logs and telemetry data. Both these approaches have their specifics and suitability for certain tasks.

Many challenges need to be taken into account when detecting patterns in cloud-based

systems. For instance, the cloud-native development best practices [4] suggest separating microservice codebases to enable decentralized evolution. However, current static analysis tools operate on a single codebase [7] only. As a result, we cannot detect patterns that span across the whole system by concatenating analysis per codebase. Also, while static analysis can detect bugs, vulnerabilities, and smells [8, 9], before putting software into production, it also has the disadvantage of requiring language-specific analysis. It is challenging to find language-specific static analyzer tools unless the modules are written in a few popular languages. Otherwise, the developers need to create their tool, which will increase maintenance workload, or they must exclude those modules from analysis, which is also not a good option. Dynamic analysis can help solve this language-specific issue. Dynamic analysis tools can operate on the decentralized perspective [10]. However, the dynamic analysis does not reveal a comprehensive detail of the system (e.g., concerning the actual implementation of the service). Telemetry data is one form of dynamic analysis which provides a few key analysis perspectives, such as Service Dependency Graph (SDG), detecting architectural smells, and architectural evolution using the SDG.

This thesis considers a catalog of well-established Microservice-specific API design Patterns (MAP). It reports tools that can detect these patterns and mechanisms for the detection (i.e., static analysis, tracing, log mining). The goal is to identify current gaps in pattern mining and quality assurance automation tools and potentially fill them using the technique of reconstructing API call graphs through dynamic analysis and log mining.

To approach this, we adopted a Multivocal Literature Review (MLR) process [11] surveying the systematic gray literature. The key motivation for conducting an MLR and therefore including the grey literature is the strong interest of practitioners on the subject, and grey literature content creates a foundation for future research.

As a result, this work identifies a list of 46 MAPs and 59 pattern mining tools. Out of the 46 patterns, 34 have been addressed by found tools. These 34 MAPs can be discovered by 26 tools out of 59. Most importantly, we identified gaps in current tools to support MAP pattern identification. We further provide discussion for the reasons behind such a gap and what needs to be addressed to overcome the current hurdles. We also attempt to partially fill this gap - we use telemetry data to determine inter-service communication and build the Software Dependency Graph (call graph). Next, we use this information to detect three MAPs that are not currently covered by existing tools. Thus we provide a new tool that increases the coverage of detectable MAPs.

This work relies on the literature review we published in [12] for identification of patterns and existing tools and adapts the work we did in [13] for detecting the three MAPs using the call graphs (SDG).

The remainder of this paper is structured as follows. Section 2 presents the overall re-

search methodology for both the literature review and the detection of the patterns. Section 3 presents the background and related works. Section 4 provides detailed information on the gray literature review process we adopted. Section 5 presents the tool we develop to detect three new MAPs. Section 6 discusses implications for practitioners and researchers while finally, Section 7 draws the conclusions.

2. RESEARCH METHODS

This work is composed by two main parts.

The first part was initially published as a research paper [12], and in it we perform a Multivocal Literature Review to determine what tools currently exist to detect the patterns proposed in [14]. Since we are interested in considering the problem from the practitioner/developer side, we search for grey literature on the Internet with Google search and consider several types of sources.

Additionally, we analyse each pattern's description and consider if it can be theoretically detected by analysing one of the five attributes of a microservice system - AST call graph of the services, static analysis of the source code, studying the history of Git repositories, as well as mining of dynamically generated event and access logs. As a result we see the current situation of MAP coverage and well as the potential for future tools. We note the uneven distribution of tools by patterns, leading to some patterns being detected by many tools but some patterns currently having no corresponding tools.

In the second part we attempt to improve the situation by creating a tool that detects three patterns that are poorly covered according to our review in the first part. In particular, we are interested to detect the patterns using the approach we piloted in [13] - reconstructing the Service Dependency Graph between the services of the system using the access logs generated dynamically. We develop the tool using Python and its library NetworkX, since these are the resources already familiar to us. Our tool is applied in a case study of TrainTicket, a benchmark open-source microservice system.

We extend the approach used in [13] by considering several Business processes separately, which allows us to see if a certain pattern or its violation is only noticed in particular use cases, thus revealing potential inefficiencies that could be overlooked by developed and that are masked if a single SDG is reconstructed for all processes simultaneously.

3. BACKGROUND

In this section, we provide further insights into three key concepts that we use in our study of Microservice patterns.

3.1 Microservices

Microservices are becoming more and more popular. Big players such as Amazon, Netflix, Spotify as well as small and medium-sized enterprises are developing Microservices-based systems [15].

A good introduction to Microservice Architecture is provided in [16] and [5]. Web article of Martin Fowler [5] is considered seminal work on defining the term 'microservice' and many works studying microservices and MSA end up referencing it several times.

Since Microservices and Microservice systems are naturally compared to more classic Monolithic applications, it is makes sense to explain them first. Fowler provides such a summary:

... [The] monolithic style [is when] a monolithic application [is] built as a single unit. Enterprise Applications are often built in three main parts: a client-side user interface, ... a database ... and a server-side application. The server-side application will handle HTTP requests, execute domain logic, retrieve and update data from the database, and select and populate HTML views to be sent to the browser. This server-side application is a monolith - a single logical executable. Any changes to the system involve building and deploying a new version of the server-side application.

To explain how Microservices are different from Monolithic application, Taibi et al in [16] provide the following summary of Fowler's ideas and some other notions:

Microservices are autonomous services deployed independently, with a single and clearly defined purpose. Microservices propose vertically decomposing applications into a subset of business-driven independent services. Each service can be developed, deployed, and tested independently by different development teams and using different technology stacks. Microservices have a variety of different advantages. They can be developed in different programming languages, can scale independently from other services, and can be deployed on the hardware that best suits their needs. Moreover, because of their size,

they are easier to maintain and more fault-tolerant since the failure of one service will not disrupt the whole system, which could happen in a monolithic system. Various companies are adopting Microservices since they believe that it will facilitate their software maintenance. In addition, companies hope to improve the delegation of responsibilities among teams.

Business Processes *Business process* is a term that comes from the business/management sphere rather than from technology, however we adopt it here since good technology is developed by understanding the needs that it fulfills and expectations that are placed onto it.

In [17], Mathias Weske defines business process as *set of activities that are performed in coordination in an organizational and technical environment. These activities jointly realize a business goal. Each business process is enacted by one organization, but it may interact with business processes from other organizations.* Here he talks about activities in an organization in general, although he does mention that a share of activities performed for a business process are technical. Since in this work we will focus on the technological implementations of business process, we use a more specific definition provided in the glossary of Gartner[1]:

Business process is an *event-driven, end-to-end processing path that starts with a customer request and ends with a result for the customer.* To put simply, a business process is a special use case or a 'user story' that a certain product/technology might enable.

In this work we wish to study the detection of patterns in a microservice system while examining the system during several different business processes, since services might have been written bearing specific use cases in mind and thus be inefficient if later they are used in some other functionality which they nonetheless enable, thus a certain pattern or anti-pattern concerning a microservice will be more pronounced in the business processes that use that service and naturally invisible in processes that do not use it - in [16], authors note that *Reusability is amplified in Microservices. Therefore, systems that need to reuse the same business processes can benefit more from Microservices, while monolithic systems in which there is no need to reuse the same processes will not experience the same benefits.*

3.2 Microservice Patterns and their detection

One way to determine software quality is to analyze software for patterns and to verify that it doesn't contain anti-patterns [2, 3] or to calculate quality metrics such as coupling and cohesion [18]. Different tools have been proposed to perform automatic quality reviews using static analysis of code to operate on pattern detection [19]. Pattern mining has been

[1] https://www.gartner.com/en/information-technology/glossary/business-process

broadly researched [2, 20, 21] and it is a well-established domain, at least for monolithic applications.

Various microservices patterns have also been identified [4, 22, 23] for various tasks, such as porting from monoliths [23], supporting resilience [22], targeting good design practices [2, 24], among others. Development frameworks apply these patterns [22] to simplify development.

In this thesis, we consider current Microservice API Patterns (MAP further in the text) reported on the API-Pattern website [3, 14]. The API-Pattern website collects the vast majority of microservices patterns proposed by its creators in peer-reviewed literature [3, 25, 26, 27, 28, 29, 30]. The list of patterns is provided in Table 4.1 while the complete description of each pattern can be found on the API-pattern website [14]. We have, however, excluded the pattern *"Annotated Parameter Collection"* as it lacked a detailed description.

The major difference between monolith and cloud systems in regards to pattern mining, however, is the decentralized codebase, which likely introduces diversity, heterogeneity, and no obvious connection across codebases. Despite initial codebase convention efforts, these get easily lost with evolution and management diversity.

Because of possible diversity across microservices, practitioners often resort to assessing cloud systems through dynamic analysis [22, 31]. However, with such direction, we can recognize endpoints and calls but not internal microservice details often needed for pattern detection (since many patterns, including some MAP, have to do with internal implementation as well, e.g. *"Backend Integration"*).

When we consider other current analysis approaches [31], we can notice that static analysis of code or mining software repositories involves code parsing and conversion to syntax trees or various graph representations as an intermediate representation. Then it uses these intermediate representations to identify patterns. Other approaches consider log analysis. However, these approaches face difficulty with the non-structured format of log messages. While log clustering can be used, it is very challenging. For this reason, it is common to integrate event tracing, which adds logging statements to calls (i.e., via instrumentation) and collects additional information in log messages, including the originating microservice or the correlation ID to determine distributed transactions and related log messages [32].

Common tools exist for centralized logging, distributed tracing, and telemetry. In addition, various system-centric perspectives are available in such tools (i.e., Jaeger, Kiali, etc.), and it is possible to preview trace-reconstructed system topology or dependency graphs. While these views cannot be as detailed as when assessing the code, it gives sufficient abstraction on the running system and language agnosticism. Given the broad availability

of these tools, it is reasonable to consider the integration of detection of various patterns [2], anti-patterns [33], smells [34], or quality metrics (e.g. Coupling [18]) using dynamic analysis. While it might be assumed that development and operations (DevOps) engineers can easily see these indicators, Bento et al. [35] suggested that the current tools do not provide appropriate ways to abstract, navigate, filter, and analyze tracing data and do not automate or aid with trace analysis. Instead, the process relies on DevOps, but these might lack the expertise or the time necessary to determine the statistics in the ever-changing environment. When using traces, it can be seen that SDG is commonly used. For instance, Ma et al. [36] use it to analyze and test microservices through graph-based microservice analysis and testing. However, Ma et al. made the process dependent on DevOps manual efforts to detect anomalies by analyzing risky service invocation chains and tracing the linkages between services.

Another approach worth mentioning is program slicing [37], which combines log analysis with code analysis. This is accomplished by locating logging statements in the code and identifying logging templates in these statements; these are then matched with log messages found in logs to these code locations [38]. For instance, in Lprof [38], the authors used program slicing to profile distributed systems and optimize their performance. They matched log statements with log messages and performed a data-flow analysis of method parameters to identify if these parameters change across call paths. Unchanged parameters identified related log messages and could be used as a correlation ID similar to event tracing. Using this, they could recognize distributed transactions and their frequencies in logs. However, with event tracing (i.e., OpenTelemetry [13]), such tasks become much more simplified and commonly adopted by industry (establishing the correlation ID). Any of these techniques could be used to help with detecting MAPs.

4. MICROSERVICE API PATTERNS LITERATURE REVIEW

This chapter is adopted from our paper on "Survey on tools detecting microservice API patterns" [12]. The background information for this and further work has already been presented in section 3, so now we focus only on the review itself.

4.1 Review Design

This section describes the methods adopted to gather and classify the different tools to detect microservices best practices.

Since our goal is to map existing tools recommended and adopted by practitioners, we performed a systematic review of the grey literature. A review of peer-review literature would be biased toward academic opinions and would not clearly enable us to understand what practitioners can find when looking for such types of tools online.

In order to investigate the aforementioned goal, we formulated our research questions as:

RQ1: Which are the tools available to detect MAPs?

This RQ aims to find tools dedicated to mining-specific patterns described in the previous section. Even though some of them could be identified using general-purposes testing tools, the necessity to write tests for the specific patterns makes it unfeasible for the patterns to be adopted in the industry, while having a set of ready-made tools that could be incorporated into the CD/CI pipeline would facilitate the adoption of best practices and improve code and design quality.

RQ2: Which MAPs can be detected automatically with tools?

In this RQ, we map the tools identified in RQ1 to the patterns they detect.

RQ3: Which techniques can be used to detect MAPs?

In this RQ, we aim to understand whether MAPs can be detected using static or dynamic analysis tools. The possibility of detecting the pattern from static analysis tools would enable understanding of the patterns used without running the system. In contrast, the detection based on the dynamic execution of the system and the collection of log traces

would allow an understanding of how the system is actually behaving.

4.1.1 The Review Process

Grey literature Reviews and Multivocal Literature Reviews (MLR) proved to be the best choice for the research method due to the lack of maturity of the subject. MLR includes both academic and grey literature. However, since we are aimed at investigating the word of mouth of practitioners, we will perform a review of the only grey literature. The key motivation for the inclusion of grey literature is the strong interest of practitioners in the subject, and grey literature content creates a foundation for future research.

The process adopted is similar to the MLR, but doesn't include the peer-reviewed literature steps.

The process we adopted was based on these steps:

- Selection of keywords and search approach
- Initial search and creation of an initial pool of sources
- Snowballing
- Reading through material
- Application of inclusion / exclusion criteria
- Evaluation of the quality of the grey literature sources
- Creation of the final pool of sources

4.1.2 Literature Search Process

Since we are interested in finding tools to detect MAPs (Table 4.1), we created 53 query search strings. The search strings were as follows:

- `"pattern_name" api pattern detection tool`, where `pattern_name` was replaced with every pattern from table 4.1 (46 total searches, excluding *"Annotated Parameter Collection"* pattern)
- `api security analysis tool`
- `api parameter analysis tool`
- `api parameter discovery tool`
- `api documentation analysis tool`
- `api specification analysis tool`
- `semantic versioning identification tool`
- `microservice api pattern detection tool`

The latter strings were added because the first parametric search string did not produce many meaningful results, and at the same time, many patterns can be grouped together, as represented by other search strings. We understand that not all groups of patterns are represented by latter search strings, so partial bias is introduced; however, we could not think of search strings targeting other groups of patterns and, as stated before, not using them produced only a handful of results.

We applied the Search strings to the Google Search[1] engine, looking at 10 pages of results per search (excluding ads). The search was done with Incognito browser mode without logging into a personal Google account. The decision to use 10 pages of results was adopted after an informal piloting of the search, which showed that no relevant results appear on pages 9-10 of the search and that for some patterns (search strings) only a few results (2-3 pages, sometimes not enough even for 1 page) are returned.

[1] www.google.com

Table 4.1. *The list of Microservice API patterns [14]*

Foundation	**Quality**
Frontend integration	API Key
Backend integration	Rate Limit
Public API	Rate Plan
Community API	Service Level Agreement
Solution Internal API	Error Report
API Description	Conditional Request
Responsibility	Request Bundle
Processing Resource	Wish List
Information Holder Resource	Wish Template
Computation Function	Embedded Entity
State Creation Operation	**Structure**
Retrieval Operation	Atomic Parameter
State Transition Operation	Atomic Parameter List
Operational Data Holder	Parameter Tree
Master Data Holder	Parameter Forest
Reference Data Holder	Data Element
Data Transfer Resource	Id Element
Link Lookup Resource	Link Element
Evolution	Metadata Element
Version Identifier	Annotated Parameter Collection
Semantic Versioning	Context Representation
Two In Production	
Aggressive Obsolescence	
Experimental Preview	
Limited Lifetime Guarantee	
Eternal Lifetime Guarantee	

Search results consisted of blog posts (including blogs with lists of tools), websites, research papers, Github[2] repositories of tools, and Github repositories of lists of tools. It is good to note that StackOverflow[3] is a popular website for technical peer questions, and it could be expected to appear in the search results, but in reality, it did not. It could have been included in the study separately, but after piloting it, it did not provide meaningful results, so in this review, we decided to focus only on Google Search.

This search was performed between **10th and 21st of January 2022**.

4.1.3 Snowballing

We applied a backward snowballing to the retrieved literature in the following way:

- If the resource extracted from the search is a list of tools (as opposed to a page of just one tool), such as "Top 10 API security tools in 2021"[4], then we checked all of the referenced tools (and potentially other referenced lists)
- If the resource is a research paper, we checked if the paper cites other algorithms that fit our criteria and included them as well

4.1.4 Application of inclusion / exclusion criteria

Based on MLR guidelines [11], we defined our inclusion criteria:

- For pages of tools: the tool description directly contains one of the MAPs from 4.1
- For research papers: the paper proposes a new algorithm/tool to detect a pattern whose description in the abstract is similar to studied patterns

Moreover, we defined our exclusion criteria as:

- Exclusion criterion 1: Non-English results
- Exclusion criterion 2: Duplicated result
- Exclusion criterion 3: (for papers) The paper proposes a new algorithm/tool to detect a pattern; however, no source code for the tool is provided
- Exclusion criterion 4: (for commercial tools) The tool has no public documentation of functionality available
- Exclusion criterion 5: (for tools) The tool is an all-purpose security tool claiming to, e.g., "identify over 1000 different vulnerabilities", i.e., no particular reference to one of the MAP is given.

[2] www.github.com
[3] www.stackoverflow.com
[4] This is a made-up example

- Exclusion criterion 6: (for tools) The tool cannot automatically perform the analysis but requires the programmers to configure it and adapt a certain workflow first (e.g., configure the tool to detect certain words in commit messages and then use them in work), so that it cannot be used to retrospectively analyze the history of an existing project.

4.1.5 Evaluation of the quality and credibility of sources

In order to evaluate the credibility and quality of the selected grey literature sources and to decide whether to include a grey literature source or not, we extended and applied the quality criteria proposed by Garousi et al. [11], considering the authority of the producer, the methods applied, objectivity, date, novelty, impact, and outlet control. We adopted the same evaluation sheet adopted by Peltonen et al. [39].

Two authors assessed each source using the aforementioned criteria, with a binary or 3-point Likert scale, depending on the criteria themselves. In case of disagreement, we discussed the evaluation with the third author, which helped to provide the final assessment.

We finally calculated the average of the scores and rejected grey literature sources that scored lower than 0.5 on a scale that ranges from 0 to 1.

4.1.6 Creation of final pool of sources

Originally, 45 different resources were identified as relevant. After performing snowballing on papers and lists and filtering all papers and tools through the exclusion criteria, we had a list of 59 tools.

4.1.7 Data Extraction

In order to obtain the list of tools to detect MAPs (**RQ1**) we extracted the tool names from the selected sources.

To understand which pattern is detected by the tools (**RQ2**), firstly, we read through the tool documentation. There are two sorts of documents available: a dedicated website and a README file. Few tools have both. After reading the documentation, we investigated whether this tool recognizes any MAP and whether it is possible to obtain information about those patterns using this tool. We couldn't locate any tool that recognizes MAP on its own because these MAPs are not commonly well-established yet. However, a few tools disclose data that can be utilized to develop an insight into MAPs.

We examined each tool by reviewing its documentation to see if it exposed any information

that may be utilized to detect any MAP. When we locate a tool that can detect a pattern, we map that tool to that pattern.

Lastly, to understand which technique can be adapted to detect MAPs (**RQ3**), We manually analyzed each pattern, including those not discovered by tools, and mapped each pattern to the possible technique. For the techniques, we considered static and dynamic analysis. For static analysis, we considered plain code analysis, operation with call graphs, which might require more advanced algorithms, and mining software repositories which may include additional information. For dynamic analysis, we considered application log analysis considering rich logging in the system code (an ideal case) and event logs (i.e., received from instrumentation with correlation ID [22]). We did not consider program slicing.

The first two authors both independently examined and mapped each pattern to the various techniques using their own reasoning. After that, the differing viewpoints were resolved by consulting with each other as well as the other authors.

4.2 Results of review

This section reports the results we obtained following the research methodology highlighted in Section 4.1.

As for the tools available to detect MAPs (**RQ1**), we identified 59 tools from 45 sources. Tables 4.2 and 4.3 list the tools retrieved (Open Source and Commercial, respectively), together with their URL, license (for Open Source tools), languages supported for analysis, and date of the last update (for Open Source tools). As for the Licenses adopted by OSS projects, the license indicated in their repository is stated, such as MIT, Apache, etc.; 'OSS' refers to tools whose source code is openly available, but no license is added to the repository; 'N/A (Free)' refers to tools that are available for free (e.g., as a web service), but not in Open Source form, and thus might be subject to a custom license as well. Different tools can analyze MAPs from the perspective of different programming languages. Some tools support several languages. Other tools scan git commits to perform the analysis, and thus applicable to any language. While other tools either access the APIs under analysis using the provided endpoints or analyze API specifications in OpenAPI format, thus the language of implementation doesn't matter. The language reported in Table 4.2 as 'Any' refers to tools that parse source code in a language-agnostic manner and thus apply to any language.

There are 7 Commercial tools and 52 OSS tools to provide some summary statistics.

When it comes to OSS licenses, 33 of tools (56% of total tools) are using permissive licenses (Apache, MIT, BSD, etc.), and 5 more use no license at all, while 7 tools (11%) use 'copyleft' licenses (different versions of GPL license).

Table 4.2. *OSS tools to detect Microservice API Patterns (RQ1)*

ID	Tool	URL	License	Language supported	Updated
T1	API security tools audit	bit.ly/3UJImRv	N/A (Free)	REST API	N/A (Active)
T2	apidiff	bit.ly/3J0FyZN	MIT	Java	31.10.2021
T3	Arjun	bit.ly/3uFZoWv	GPL-3.0	REST API	29.08.2021
T4	Astra	bit.ly/3ontBp4	Apache-2.0	REST API	05.04.2019
T5	Brakeman	bit.ly/3LbQtSo	MIT	Ruby	30.01.2022
T6	Coala	bit.ly/3AXgKiw	AGPL-3.0	Any	11.06.2021
T7	code2flow	bit.ly/34w2MYM	MIT	Many (4)	27.12.2021
T8	git-secret	git-secret.io	MIT	git	01.02.2022
T9	git-semv	bit.ly/34yi1jM	MIT	git	17.06.2021
T10	GitVersion	bit.ly/3AVeUi6	MIT	git	31.01.2022
T11	go-semrel-gitlab	bit.ly/3upDINT	MIT	git	27.10.2019
T12	GraphQL FBC-CLI	bit.ly/333l57q	OSS	GraphQL	06.06.2018
T13	Hikaku	bit.ly/3Hubkhx	Apache-2.0	REST-API	19.08.2021
T14	jgitver	bit.ly/3glahV5	Apache-2.0	Java	30.03.2021
T15	Kiterunner	bit.ly/3J77eML	AGPL-3.0	REST API	10.05.2021
T16	LAPD	bit.ly/3gl6FlZ	N/A (Free)	Java	07.09.2013
T17	magento-semver	bit.ly/3gnWygi	OSL-3.0	N/A	19.01.2022
T18	microservices-antipatterns	bit.ly/3GwBwGJ	Apache-2.0	Python	17.12.2019
T19	modver	bit.ly/32WbY8k	MIT	Go	16.01.2022
T20	Mondrian	bit.ly/3rqHkNZ	OSS	PHP	16.09.2014
T21	MSA-nose	bit.ly/3sgODXF	OSS	Java	12.04.2021
T22	next-ver	bit.ly/3B5EWzq	MIT	git	09.02.2018
T23	NodeJS scan	bit.ly/3uqxHkm	GPL-3.0	Node.js	31.01.2022
T24	NoRegrets	bit.ly/3JjDAnF	Apache-2.0	JavaScript	02.07.2019
T25	OpenAPI diff	bit.ly/3oncymY	MIT	REST API	29.08.2017
T26	OpenAPI spec validator	bit.ly/3sogwxc	Apache-2.0	REST API	28.01.2022
T27	openapi-lint	bit.ly/3oqUJDA	BSD-3	REST API	12.08.2020
T28	openapilint	bit.ly/3J45Lqs	MIT	REST API	13.05.2019
T29	oval	bit.ly/3J466te	MIT	REST API	26.09.2018
T30	pact	bit.ly/3gp77Q8	MIT	REST-API	03.02.2022
T31	paramspider	bit.ly/3sf6JZZ	GPL-3.0	REST API	12.09.2021
T32	Prometheus	prometheus.io	Apache-2.0	Many (5)	02.02.2022
T33	prospector	bit.ly/3oo4Zg7	GPL-2.0	Python	01.02.2022
T34	Public API changes	bit.ly/3gjOKMF	Unlicense	C#	05.11.2017
T35	pycallgraph	bit.ly/3uqyZfc	GPL-2.0	Python	28.02.2018
T36	Pyramid OpenAPI3	bit.ly/3uqbZg5	MIT	Python	07.12.2021
T37	pyramid-swagger	bit.ly/3sfLKqb	BSD-3	Python	30.03.2020
T38	Python Semantic Release	bit.ly/3oo3uyg	MIT	Python	31.01.2022
T39	REST API Antip. Inspect.	bit.ly/3GkLhrG	MIT	REST API	31.03.2021
T40	schaapi	bit.ly/3J3BcB7	MIT	Java	11.02.2019
T41	secret-detection	bit.ly/3rqsM0N	OSS	Any	03.08.2020
T42	semantic-release	bit.ly/3usLkPQ	MIT	git	18.01.2022
T43	semantic-versioning-anal.	bit.ly/3HsvgBn	MIT	.NET	03.11.2021
T44	semver-config	bit.ly/3GnLpGN	OSS	git	26.09.2019
T45	semverbot	bit.ly/3GteoZI	MPL-2.0	git	03.01.2022
T46	Speccy	bit.ly/3uknqGe	MIT	REST API	02.10.2019
T47	Spectral	bit.ly/3gnIeo3	Apache-2.0	REST API	03.02.2022
T48	SpotBugs	bit.ly/3glbvQj	LGPL-2.1	Java	29.01.2022
T49	Standard Version	bit.ly/3uqgx64	ISC	Node.js	01.01.2022
T50	Vulture	bit.ly/3Gq9hd2	MIT	Python	03.01.2022
T51	wFuzz	bit.ly/3opt5Hg	GPL-2.0	REST API	28.11.2020
T52	Zally	bit.ly/34lLHkq	MIT	REST API	14.01.2022

Table 4.3. *Commercial tools to detect Microservice API Patterns (RQ1)*

ID	Tool	URL	Supported Languages
T53	Acunetix	acunetix.com	Web
T54	CheckMarx	checkmarx.com	Many (20)
T55	CodeClimate	bit.ly/3GrqB12	Many (11)
T56	Data Theorem API Secure	bit.ly/34ASaYM	N/A
T57	Dynatrace	dynatrace.com	Many
T58	SonarQube*	bit.ly/3GryR15	Many (29)
T59	Synopsys	bit.ly/3uqjhRd	REST API

*Dual-licensed, available both as commercial and open source (GPL).

In terms of languages, 16 tools (27%) are written in Python, 11 tools (19%) are written in JavaScript (some of them in TypeScript), Java and Go have 6 tools each (10% each); other represented languages are C++/C#, Kotlin, PHP and shell scripting (bash).

Supported languages/platforms involve 8 tools analyzing commits in Git (14%), another 8 tools targeting Python as the only language (13%), 6 tools (10%) targeting Java, and another 6 tools (10%) supporting several languages. Also, 19 tools (32%) target REST APIs directly, with 9 analyzing specifications and 10 using the endpoints dynamically.

Out of 52 OSS tools, 32 have been updated at least once since January 2021. Furthermore, 19 have been updated already in 2022.

It should also be noted that some tools are research prototypes (T2, T16, T18, T21), some grew out of research prototypes (T48), while many are projects done by hobbyists (T19, T34, T37 to name a few), so their quality and applicability to up-to-data languages and frameworks could be limited. The scope of this review is simply to identify existence of *some* tools to address MAPs and see the pattern coverage, assessing the actual quality and usefulness of the tools is a different, much more complicated endeavour.

As for the MAPs that can be detected automatically with tools (**RQ2**), we found 26 tools that expose information about the tools. These tools are listed as RQ2 column in Table 4.4. Our table shows that found tools detect a subset of all patterns, and a combination of tools is necessary to address broader coverage. For example, the pattern 'API Description' is detected by 9 tools in the Foundation category, while no tool targets the other 5 patterns in this category. Two tools, code2flow, and pycallgraph, which are both based on call graphs, can identify all of the patterns in the Responsibility category. Hikaku is a tool that can be used to detect all of the patterns in the Structure category. All of the patterns in this category, with the exception of 'Context representation,' can be discovered with four or more tools. There are three patterns in the quality section that no tool can detect. Rate Plan, Service Level Agreement, and Wish Template are examples of these patterns. At least one tool detects the rest of the patterns. Three patterns are not detected by any

tools in the Evolution category, whereas four patterns are detected by at least three tools.

The most promising techniques to detect MAPs (**RQ3**) stem from the static analysis involving source code analysis as detailed in Table 4.4. Most of the tools we found use static analysis. In particular, the static analysis might need to determine call graphs to detect certain patterns, and also, the detection process can use mining software repositories. Nevertheless, given the API level, dynamic analysis can also determine a large number of patterns. While application log analysis is one option, it is a challenging option dependent on the level of logging. It is more convenient to use event logs resulting from recent cloud-native frameworks and infrastructure advancements. All events are centralized and aggregated by the occurrence time, with correlation ID indicating message dependencies.

Table 4.4. *Toosl and possible Techniques for detecting microservice patterns*

Patterns	**RQ2** Detected by tool (ID)	**RQ3**				
		Static Analysis			Dynamic Analysis	
		Call Graph	Source Code	Repository	Event Log	Application log
Foundation						
Frontend integration		✓	✓		✓	✓
Backend integration		✓	✓			
Public API						
Community API						
Solution Internal API						
API Description	T25, T26, T27, T28, T29, T36, T37, T46, T52		✓			
Responsibility						
Processing resource	T7, T35	✓	✓		✓	✓
Information holder resource	T7, T35	✓	✓		✓	✓
Computation function	T7, T35	✓	✓		✓	✓
State recreation operation	T7, T35	✓	✓		✓	✓
Retrieval operation	T7, T35	✓	✓		✓	✓
State transition operation	T7, T35	✓	✓		✓	✓
Operational Data Holder	T7, T35	✓	✓		✓	✓
Master data holder	T7, T35	✓	✓		✓	✓
Reference data holder	T7, T35	✓	✓		✓	✓
Data transfer resource	T7, T35	✓	✓		✓	✓
Link Lookup resource	T7, T35	✓	✓		✓	✓
Structure						
Atomic parameter	T3, T6, T13, T31, T36, T37		✓			
Atomic parameter list	T3, T6, T13, T31, T36, T37		✓			
Parameter tree	T3, T6, T13, T31, T36, T37		✓			
Parameter forest	T3, T6, T13, T31, T36, T37		✓			
Data element	T6, T13, T31, T36, T37		✓			
Id element	T3, T6, T13, T31, T36, T37		✓			
Link element	T6, T13, T31, T36, T37		✓			
Metadata element	T13, T31, T36, T37		✓			
Context representation	T13		✓			
Pagination	T13, T31, T36, T37		✓		✓	✓
Quality						
API key	T13, T36, T37		✓		✓	✓
Rate limit	T36, T37, T53		✓		✓	✓
Rate plan						
Service level agreement						
Error report	T13, T36, T37	✓	✓		✓	✓
Conditional request	T13, T36, T37	✓	✓		✓	✓
Request bundle	T13	✓	✓		✓	✓
Wish list	T36, T37	✓	✓		✓	✓
Wish template		✓	✓		✓	✓
Embedded entity	T13	✓	✓		✓	✓
Linked information holder	T13	✓	✓		✓	✓
Evolution						
Version identifier	T6, T10, T13, T45		✓	✓	✓	
Semantic versioning	T2, T9, T13, T19, T38, T40, T42, T43, T44, T45		✓	✓	✓	
Two in production			✓	✓	✓	
Aggresive obsolesense		✓	✓	✓	✓	
Experimental preview			✓	✓	✓	
Limited lifetime guarantee	T2, T13, T40		✓			
Eternal lifetime guarantee	T2, T13, T40		✓			

5. DETECTING MICROSERVICES API PATTERNS

As discovered in the previous section, there is still a lot of room to develop tools to detect certain Microservice API patterns using particular techniques such as mining of application or event logs. In this section we make our own contribution to this area by describing a tool that can detect 3 MAPs, one of which has no tool coverage according to our review, and another one - only by one OSS tool.

5.1 Goal

The main goal of this thesis is to study the detection of Microservice API patterns using a call graph reconstructed from application logs while simulating several realistic business processes to see if some pattern only appears in parts of the system for specific use cases.

To achieve this goal we need to find an existing (benchmark) microservice system, a way to simulate load and business processes on it, a tool to collect logs as well as invent a way to reconstruct the dependency graph from these logs and define the selected detected patterns as a problem (property) on this graph.

Note: here and further we will use the term *call graph* to refer to a graph of dependencies between microservices obtained through dynamic analysis of a running system. Such a graph is also called SDG - Service Dependency Graph (see Section 1). This is unlike Section 4, where this term referred to a graph identified through static analysis of the source code.

Also, we assume that the reader is familiar with basic concepts from graph/network theory such as graph/network, vertex/node, edge/arc and degree (in_degree/out_degree) and do not provide an introduction to this topic here. More advanced concepts will not be necessary for our study of SDGs.

5.1.1 Research questions

We can formulate the ideas described in the above section as Research Questions. Since the patterns to identify have already been selected (see Section 5.1.2), the only remaining question are:

RQ4: How to implement the detection of the patterns from the reconstructed call graph?

After deliberation with the supervisors and coauthors of [12], it was agreed that the three selected patterns (section 5.1.2) are the ones that are 'in theory' detectable using a call graph (SDG). Now we need to actually decide how to formulate the detection in terms of graph's properties. The answer for each respective patterns is given in the same section 5.1.2.

RQ5: What will the detection reveal about the system that is used as a case study?

We want to run the detectors on data from a real-life systems to see if our algorithms (and underlying MAPs) indeed provide a useful insight into the system's architecture and operation. The selected system is TrainTicket (section 5.2.1).

5.1.2 Selected patterns

After we determined that we want to pursue the development of the MAP detectors using application logs and SDG, we have determined that the following MAPs are detectable in this way:

Frontend Integration[1]

The idea behind this patterns is that Frontend microservices need to be integrated into the system vertically, and all the business logic has to be put into other services, to which the Frontend service simply forwards the data (see figure 5.1). This mimics the idea used in design of the monolithic application, especially in Java programming language, which says that UI code and logic code have to be separated and *god classes* that do everything at once better be avoided.

We can formulate this patterns as a SDG property that *nodes corresponding to Frontend services should have in_degree* $= 0$ *and out_degree* ≥ 0.

The detection can be done in two ways

- Detect those service that fulfill the property above and list them as 'potential' Frontend services.

  This way the creator(s) of the system can see if there are some services that behave *as if* they were a Frontend service even though they are not (i.e. a service which is not a Frontend service, does not receive calls from other services but spontaneously calls other services might be an anomaly).
- Given a list of services designated as Frontend services, check if they fulfill the property listed above.

[1] https://www.microservice-api-patterns.org/patterns/foundation/FrontendIntegration

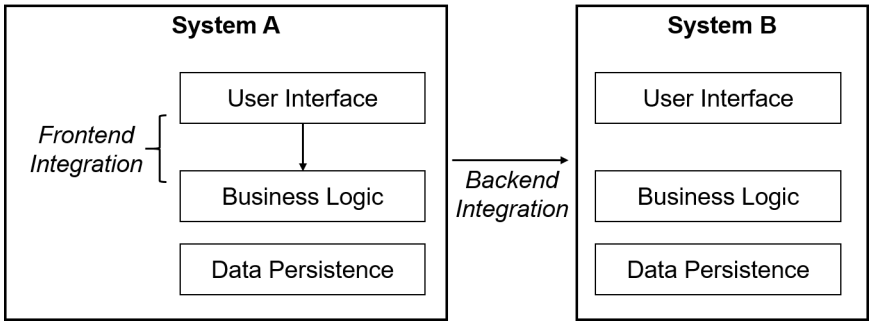

Figure 5.1. *Frontend is integrated vertically and simply provides all the data for services implementing Business Logic [14].*

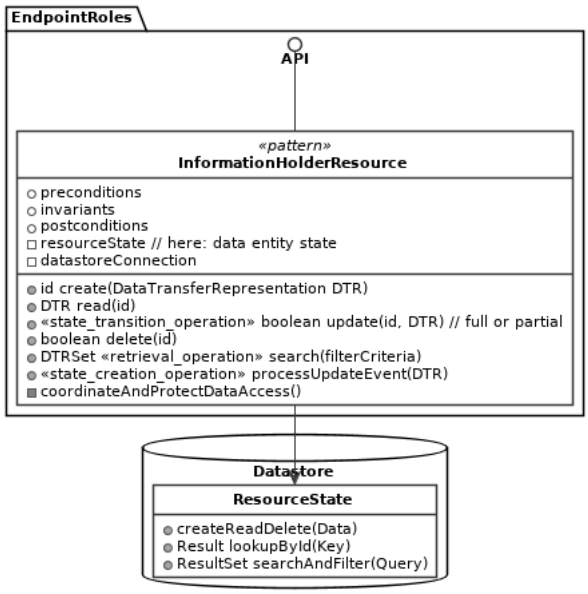

Figure 5.2. *Information Holder Resource provides an unchanging API for CRUD operations to the underlying database, whose implementation can change [14].*

This way the developers can check if their Frontend services behave as expected and no Business Logic which causes them to receive calls calls to provide some information has creeped into the codebase.

Our Literature Review did not find any tools to detect this pattern.

Information Holder Resource[2]

This pattern is based on the idea expressed by Newman in [6] that several services should not be accessing the same database directly using the SQL (or other language) queries. Instead, a separate service should be created whose only purpose is to expose the public API that would allow other services to request and modify the information in the database (so called CRUD operations - Create, Read, Update, Delete), see figure 5.2.

This will ensure services can use the same API even if the implementation of database

[2]https://www.microservice-api-patterns.org/patterns/responsibility/endpointRoles/InformationHolderResource

and its tables changes. Also, this way the Information Holder Resource can do all checks to ensure that the database is in a consistent state after each operation, instead of each Business Logic service having to implement these checks themselves for situations when they need to modify the database. This pattern also mimics the idea from general software engineering that data itself should be encapsulated within a class and only accessed and modified using the *'getter'* and *'setter'* methods.

We can formulate this as an SDG property in the following way:

A database service (DB) *and its Information Holder Resource* (IHR) *are a pair of services such that in SDG the node of DB only has incoming edges from the node of IHR, and the node of IHR only has outgoing edges into the node of DB. Additionally, the node of DB must have out_degree* $= 0$*.*

Several things can be detected about this property:

- Return a list of pairs of services (IHR, DB) that fulfill the above property.

  This will provide the developers/researchers the list of 'healthy' database-and-ihr implementations.

- Return a list of pairs of services (IHR, DB) that 'almost' fulfill the property in the following sense: the DB service only receives calls from the IHR service, but IHR service also sends calls to other services.

  This show the services that could be good IHRs for some DB if some functionality is refactored into other services.

- For a given list of services designed as Database Services (DB), return a list of services that call other services.

  This identifies the DB service that had some other Business Logic creeped into them that needs to be refactored.

- For a given list of services designated as Database Service (DB), find their respective (potential) IHRs and return them (items 1 and 2), and separately return the list of DBs that do not have a corresponding IHR.

  This provides a list of DB services for which the IHR implementation is still missing.

Our Literature review found only 2 tools (T7 and T35, table 4.2), both of which construct call graphs, however one of the projects has been frozen 8 years ago and the other only covers Python, JavaScript, Ruby and PHP. Thus our approach will provide a more language-agnostic way to detect the patterns through dynamic analysis, as explained in section 3.

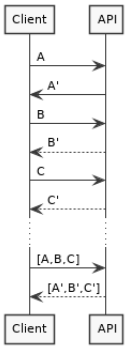

Figure 5.3. *Instead of sending several request to get homogeneous data (top), services use a Request Bundle to exchange all queries and responses during one call (bottom) [14].*

Request Bundle[3]

The idea of this pattern is that if one service A needs to receive several batches of homogeneous data from another service (endpoint) B, for examples several entries from the same database based on IDs, then instead of calling this service several times to obtain each piece of data, a mechanism should exist such that service B can receive a list of parameters from A in one request for the whole batch, and provide all data in a list in the same respective order. This list of request parameters and responses is call the Request Bundle (see figure 5.3). This is similar to how in certain numeric computations functions need to be 'vectorized', i.e. either receive one input and given out output, or receive a list (vector) of inputs and provide a list (vector) of outputs.

Due to how this pattern is formulated, it easier to detect the corresponding *anti*-pattern by negating the idea - find those situations where Request Bundle is missing or at least unused. Of course, if SDG shows the amount of times one service calls another as the weight of the corresponding edge, it is possible to detect potential violations of Request Bundle by noticing anomalously huge weights. However, this detection would not be rigorous and depend on subjective parameters - how big a weight is considered too big? how to make sure calls to the same service were consecutive if we do not store this information into the graph?

Instead of using the graph directly, we can, during the construction of the graph from the logs, put all the calls in a list and sort it by the timestamps of calls. This way we will see all the calls in chronological order. Then we just have to find consecutive calls between same pair of services, this will indicate the Request Bundle is not utilized during their communication. The detection can be done on two levels:

- Service level: Service A repeatedly calls service B
- Endpoint level: Service A repeatedly calls the same endpoint of service B

[3]https://www.microservice-api-patterns.org/patterns/quality/dataTransferParsimony/RequestBundle

Violation of Request Bundle on endpoint level should definitely be avoided - same endpoint provides same kind of data (homogeneous), so it is definitely possible to implement a list of queries/responses for this case.

On service level, of course one service could expose several types of information/functionality on different endpoints, so repeatedly calling the same service does not necessarily indicate violation of Request Bundle patterns. However developers/researchers could still benefit from monitoring this kind of situation, since according to good design principles each microservice should have clearly defined and limited responsibilities and functionalities, so it should be in theory possible to implement some kind of *master-endpoint* which receives parameters to execute several tasks of the other endpoints and thus functions as a Request Bundle.

Our Literature Review has identified only one tool (T13 - table 4.2), however this tool compares the implementation of API endpoint to its specification, so Request Bundles need to have been at least designed by the developers to be detected this way. Our approach allows to detect violation of Request Bundle pattern even if it had not been considered by the developers and indicate the interactions which would benefit from Request Bundles.

5.2 Case Study - TrainTicket

5.2.1 TrainTicket - a microservice benchmark system

Microservice systems are usually developed by big companies that can afford to have several teams developing several distinct microservices. Due to private nature of the corporate world, getting access to these systems and their data is problematic for researchers. Zhou et al have developed a benchmark microservice system called TrainTicket [40], which mimics a web application for purchasing train tickets (see figure 5.4). The system is an open source project, licensed under Apache-2.0 license and available on GitHub [41]. This project has already proven itself useful for researchers [42, 43, 44, 45, 46]

TrainTicket contains 41 microservices, both for the user-side functionalities (browse website, login, search for train, book ticket etc) and admin-side functionalities (modify user data, modify station, train, price data etc). The following languages are used in the services' codebases: Java, Node.js, Python, Go, MongoDB, MySQL, nginx. This represents a real-life situation when different teams on a project use different 'stacks' to build their services according to their skills and needs.

The system was deployed on Kubernetes [47], along with Istio [48]. The Istio was set up in such a way that it generates and outputs a JSON-formatted access log. We installed Kubernetes in a machine with 32GB of RAM and an Intel i9-8950HK processor using kind (Kubernetes-in-docker [49]). There were 6 CPU cores and 12 threads in the system.

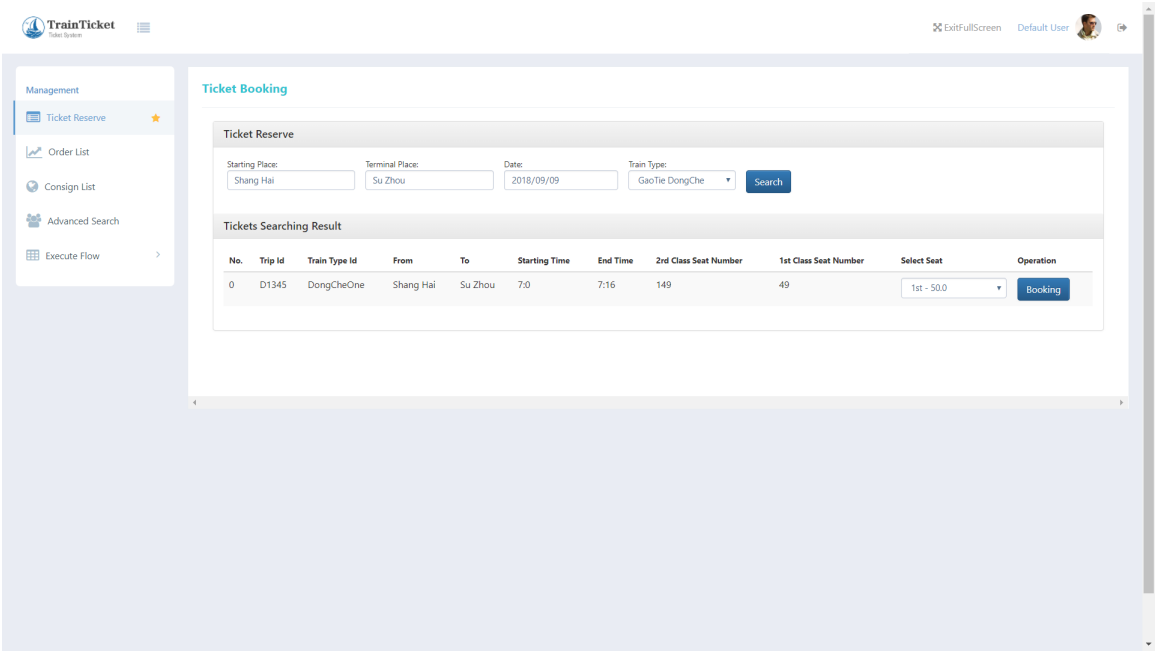

Figure 5.4. *User interface of deployed TrainTicket system[40, 41].*

Kubernetes' resource limit was set at 22GB of RAM and 8 threads of CPU.

5.2.2 Simulating business processes

The functionality of services described in TrainTicket documentation[4] allows us to define the following five business processes (we call them 'Users' here because each business process simulates a particular user's behavior; also classes to represent these processes in configuration codes are called Users):

- UserNoLogin
    1. Open the website
    2. Search for a train by departure
    3. Search for a train by return
- UserBooking
    1. Open the website
    2. Log in
    3. Search for a train by departure
    4. Book the ticket
- UserConsignTicket
    1. Open the website

[4] https://github.com/FudanSELab/train-ticket/wiki/Service-Guide-and-API-Reference

2. Log in
3. Search for a train by departure
4. Book the ticket
5. Consign the ticket[5]

- UserCancelNoRefund
  1. Open the website
  2. Log in
  3. Search for a train by departure
  4. Book the ticket
  5. Cancel the order without refund
- UserRefundVoucher
  1. Open the website
  2. Log in
  3. Search for a train by departure
  4. Book the ticket
  5. Cancel the order and get a voucher for the amount

The last three users expand the actions taken by UserBooking in different ways, which requires usage of different additional services in each case.

We used the ing tool PPTAM [50] to simulate a real-life user. It internally uses another tool called locust [51] which allows to define different Users for the studied system and well as parameters on how much and how often to spawn them. Both tools are implemented in Python, so Users are also defined as Python code in the following way:

- First, a Client class is defined that can send calls to the different endpoints in the system. Methods of this class are different atomic tasks that can be executed by the users (open home page, login with given username and password, search for a train with given parameters), each method performs necessary calls to the necessary endpoints.
- User classes are defined for each user (use case/business process) which perform the atomic tasks in the necessary order.
- User classes also have special attributes which define how locust will spawn them, such as `weight` and `wait_time`.

[5] Request to pick up the ticket from the office

In order to be able to cleanly separate activity and service calls due to each user, we configured the system to only use one user at a time and only spawn one instance of each at a time. Logs of locust then allow us to understand the interval of time in which that particular instance was running and map all system activity to it. More on this in the following sections.

During our experiment the following amount of users were spawned: UserNoLogin - 263, UserBooking - 70, UserConsignTicket - 92, UserCancelNoRefund - 199, UserRefundVoucher - 224.

5.2.3 Collecting logs

Distributed systems are notoriously difficult to manage, and adding traffic management to the mix makes things even more difficult. A service mesh comes with traffic management, observability, security, encryption, access control, rate limiting, and other features out of the box. As a result, service mesh has become an essential tool in Kubernetes-based systems. Each service module is deployed separately in this system, and each of these modules has a sidecar [52] proxy that receives and sends traffic on behalf of the microservice module. These proxy sidecars can control traffic and improve the service's observability and security.

Istio [48], linkerd [53], and consul connect [54] are the most popular open source service mesh tools, according to GitHub stargazers. In addition to these tools, most service mesh tools provide an access log (also known as audit logging or proxy log), containing information about each inbound and outbound request in a customizable format. Our method uses these access logs to determine which service is calling which service, which endpoint is being called, and how frequently a service is called in a given time period. We can use this information to reconstruct software architecture and automatically detect patterns and anti-patterns.

5.2.4 Parsing logs

During the execution of the simulation of users two kinds of logs are generated - locust logs about how Users are spawned and Istio access logs for each service about what calls the service makes.

The example of the locust logs can be seen in the figure 5.5. We have 5 log files like that for each User that was defined and run. We can use the first and last timestamps of the file (figure 5.5, lines 1 and 1857) to infer the time interval where this particular User was executing in the system. Locust was configured to spawn one instance of that User, let it execute its tasks and then spawn another one. Each instance of the User is reported in the log when it is spawned together with its UUID (figure 5.5, lines 10, 18), so we can

Table 5.1. *An example of an inbound and outbound request access log. Inbound requests have a-service as their destination, while outbound requests have a-service as their source and b-service as their destination [13].*

Request type	Sample Access Log

inbound request	{
	"start_time": "2022-05-26T06:22:02.661Z",
	"upstream_host": "10.244.0.65:12345",
	"downstream_local_address": "10.96.162.171:12345",
	"upstream_transport_failure_reason": null,
	"protocol": "HTTP/1.1",
	"upstream_service_time": "6",
	"authority": "b-service:12345",
	"requested_server_name": null,
	"response_code_details": "via_upstream",
	"connection_termination_details": null,
	"upstream_local_address": "10.244.0.41:33326",
	"downstream_remote_address": "10.244.0.41:48250",
	"path": "/api/v1/endpoint/",
	"bytes_sent": 44,
	"request_id": "4631dc4c-0a6e-9ad2-ba61-d257cdd6e50b",
	"bytes_received": 0,
	"route_name": "default",
	"duration": 7,
	"x_forwarded_for": null,
	"response_flags": "-",
	"response_code": 200,
	"method": "GET",
	"upstream_cluster": "outbound|12345||b-service.default.svc.cluster.local",
	"user_agent": "Apache-HttpClient/4.5.9 (Java/1.8.0_111)"
	}

outbound request	{
	"start_time": "2022-05-26T06:22:02.661Z",
	"upstream_host": "10.244.0.65:12345",
	"downstream_local_address": "10.96.162.171:12345",
	"upstream_transport_failure_reason": null,
	"protocol": "HTTP/1.1",
	"upstream_service_time": "6",
	"authority": "b-service:12345",
	"requested_server_name": null,
	"response_code_details": "via_upstream",
	"connection_termination_details": null,
	"upstream_local_address": "10.244.0.41:33326",
	"downstream_remote_address": "10.244.0.41:48250",
	"path": "/api/v1/endpoint/",
	"bytes_sent": 44,
	"request_id": "4631dc4c-0a6e-9ad2-ba61-d257cdd6e50b",
	"bytes_received": 0,
	"route_name": "default",
	"duration": 7,
	"x_forwarded_for": null,
	"response_flags": "-",
	"response_code": 200,
	"method": "GET",
	"upstream_cluster": "outbound|12345||b-service.default.svc.cluster.local",
	"user_agent": "Apache-HttpClient/4.5.9 (Java/1.8.0_111)"
	}

```
 1: [2022-06-13 11:58:06,590] ECS-DRTC-Resh03/INFO/locust.main: Run time limit set to 600 seconds
 2: [2022-06-13 11:58:06,591] ECS-DRTC-Resh03/INFO/locust.main: Starting Locust 2.8.6
 3: [2022-06-13 11:58:06,592] ECS-DRTC-Resh03/INFO/locust.runners: Ramping to 1 users at a rate of 1.00 per second
 4: [2022-06-13 11:58:06,593] ECS-DRTC-Resh03/DEBUG/locust.runners: Ramping to {"UserNoLogin": 1} (1 total users)
 5: [2022-06-13 11:58:06,593] ECS-DRTC-Resh03/DEBUG/locust.runners: Spawning additional {"UserNoLogin": 1}
                              ({"UserNoLogin": 0} already running)...
 6: [2022-06-13 11:58:06,594] ECS-DRTC-Resh03/DEBUG/locust.runners: 1 users spawned
 7: [2022-06-13 11:58:06,594] ECS-DRTC-Resh03/DEBUG/locust.runners: All users of class UserNoLogin spawned
 8: [2022-06-13 11:58:06,594] ECS-DRTC-Resh03/DEBUG/locust.runners: 0 users have been stopped, 1 still running
 9: [2022-06-13 11:58:06,594] ECS-DRTC-Resh03/INFO/locust.runners: All users spawned: {"UserNoLogin": 1} (1 total users)
10: [2022-06-13 11:58:06,594] ECS-DRTC-Resh03/DEBUG/root: Running user "no login" with id 6c4396b0-61fc-4ade-9253-e58b8c4aac30...
11: [2022-06-13 11:58:06,594] ECS-DRTC-Resh03/DEBUG/root: Performing task "home" for user 6c4396b0-61fc-4ade-9253-e58b8c4aac30...
12: [2022-06-13 11:58:06,612] ECS-DRTC-Resh03/DEBUG/urllib3.connectionpool: Starting new HTTP connection (1): localhost:32677
13: [2022-06-13 11:58:06,619] ECS-DRTC-Resh03/DEBUG/urllib3.connectionpool: http://localhost:32677
                              "GET /index.html HTTP/1.1" 200 18128
14: [2022-06-13 11:58:06,620] ECS-DRTC-Resh03/DEBUG/root: Performing task "search_departure" for user 6c4396b0-61fc-4ade-9253-e58b8c4aac30...
15: [2022-06-13 11:58:06,726] ECS-DRTC-Resh03/DEBUG/urllib3.connectionpool: http://localhost:32677
                              "POST /api/v1/travelservice/trips/left HTTP/1.1" 200 None
16: [2022-06-13 11:58:06,727] ECS-DRTC-Resh03/DEBUG/root: Performing task "search_return" for user 6c4396b0-61fc-4ade-9253-e58b8c4aac30...
17: [2022-06-13 11:58:06,840] ECS-DRTC-Resh03/DEBUG/urllib3.connectionpool: http://localhost:32677
                              "POST /api/v1/travelservice/trips/left HTTP/1.1" 200 None
18: [2022-06-13 11:58:07,841] ECS-DRTC-Resh03/DEBUG/root: Running user "no login" with id b9b0b541-6330-49fc-a128-f5e089fcbc01...
19: [2022-06-13 11:58:07,841] ECS-DRTC-Resh03/DEBUG/root: Performing task "home" for user b9b0b541-6330-49fc-a128-f5e089fcbc01...
20: [2022-06-13 11:58:07,847] ECS-DRTC-Resh03/DEBUG/urllib3.connectionpool: http://localhost:32677 "GET /index.html HTTP/1.1" 200 18128
21: [2022-06-13 11:58:07,847] ECS-DRTC-Resh03/DEBUG/root: Performing task "search_departure" for user b9b0b541-6330-49fc-a128-f5e089fcbc01...
22: [2022-06-13 11:58:08,220] ECS-DRTC-Resh03/DEBUG/urllib3.connectionpool: http://localhost:32677
                              "POST /api/v1/travelservice/trips/left HTTP/1.1" 200 None

...
1851: [2022-06-13 12:08:06,002] ECS-DRTC-Resh03/INFO/locust.main: --run-time limit reached. Stopping Locust
1852: [2022-06-13 12:08:06,003] ECS-DRTC-Resh03/DEBUG/locust.runners: Stopping all users
1853: [2022-06-13 12:08:06,005] ECS-DRTC-Resh03/DEBUG/locust.runners: Stopping Greenlet-0
1854: [2022-06-13 12:08:06,012] ECS-DRTC-Resh03/DEBUG/locust.runners: 1 users have been stopped, 0 still running
1855: [2022-06-13 12:08:06,015] ECS-DRTC-Resh03/DEBUG/locust.main: Running teardowns...
1856: [2022-06-13 12:08:06,015] ECS-DRTC-Resh03/INFO/locust.main: Shutting down (exit code 0)
1857: [2022-06-13 12:08:06,016] ECS-DRTC-Resh03/DEBUG/locust.main: Cleaning up runner...
```

Figure 5.5. *Example of a locust log of a process that for a given user class (line 4) spawns several instances of that user (lines 10, 18).*

also determine time intervals of activity for each User instance. Using this intervals we can map service calls from the access log to the corresponding User/User instance.

Table 5.1 contains a sample inbound and outbound request's access-log, where both logs are from service-a's proxy sidecar, and its service DNS inside the Kubernetes cluster is 'a-service.default.svc.cluster.local'.

The outbound request indicates that the event log contains upstream service information in DNS format, with the key 'upstream cluster' and 'path' indicating which upstream service endpoint is being requested. Also 'start_time' contains the time of the call, which we can use to map the call to a particular User instance. We can use this information to construct the SDG, in which a-service and b-service each represent a node, with a directed edge connecting a-service to b-service. We can also maintain the interaction count as edge-weight on this directed edge. We can analyze each access log of a distributed system and generate an edge in the graph for each interaction between two services within a time period. If the services already have an edge, we can increase the weight of the edge by one. Finally, we can use a visualizing tool to draw the graph automatically.

One thing to note is that while both the inbound and outbound logs contain downstream and upstream addresses in the form of IPs, IPs are ephemeral in Kubernetes-based systems because they rely on ephemeral 'Pods' [55]. The service's DNS, on the other hand, is stable, and it will forward traffic to a Pod's or a group of pods' updated IP address [55]. Another complication we have faced is the fact that by default the two systems (pptam/locust and Istio) log the events using different timezone, so a correction for time needs to

be done when mapping calls from access logs to time intervals inferred from the locust log.

For our tool we are using Python and in particular NetworkX library to store the resulting graphs. We use the `networkx.MultiDiGraph` class, i.e. directed multigraph class. Nodes are services and multi-edges go from service making the call to the service receiving it. For multi-edges, it is necessary in NetworkX to specify a unique key for each edge to distinguish several edges connecting the same pair of nodes. In our case the key is the endpoint being called. In addition, the total amount of such calls is recorded as the edge's weight. NetworkX has provided functions for drawing the graphs, which can be customized for a better layout.

The tool and the logs obtained in this work are available in the GitHub repository[6].

5.3 Results

Here we present examples of the obtained Service Dependency Graphs and discuss the results generated by each pattern detector.

5.3.1 Example call graphs

The call graphs for each User can be seen in figures 5.6 - 5.10.

Generally we notice that only a few services are used in UserNoLogin (figure 5.6), which only performs train searches. The activity of UserBooking (figure 5.7) is noticeably more complicated, however we observe that the the graph of UserNoLogin is a subgraph of UserBooking, which makes sense since UserBooking also performs a search first, thus it includes all activity of UserNoLogin in itself (we also verified it analytically through NetworkX).

UserConsignTicket, UserCancelNoRefund and UserRefundVoucher all extend UserBooking in three different ways, and we can see that the added functionality (consigning the ticket, cancelling the order, requesting the voucher) is cleanly separated to other service - the only addition to graph of UserBooking in the other three Users (figures 5.8 - 5.10), are the calls to `ts-consign- service`, `ts-cancel-service` and `ts-voucher-service`, respectively, and these services make other necessary calls to carry out the business logic.

[6] https://github.com/bakhtos/TrainTicketsBPE

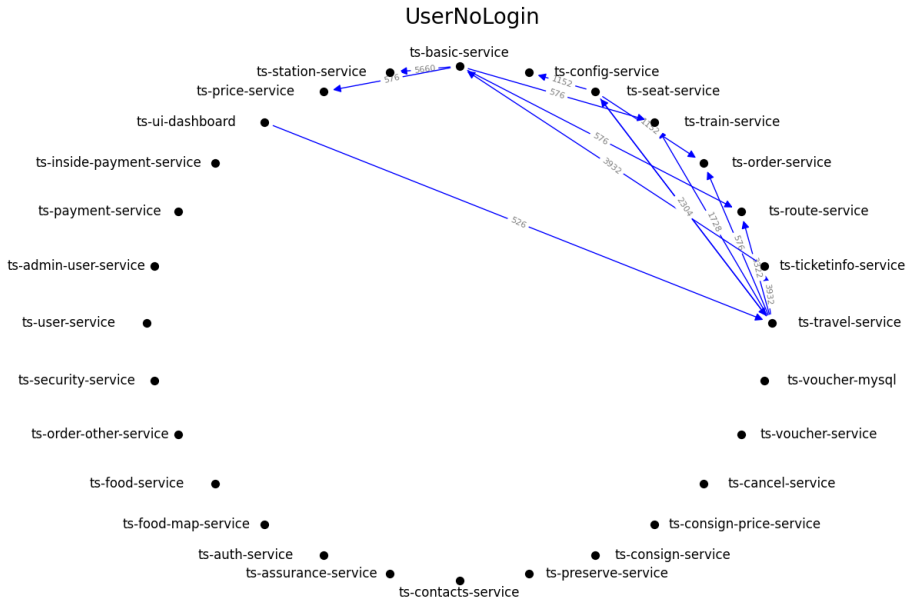

Figure 5.6. *Aggregated call graph of all instances of UserNoLogin*

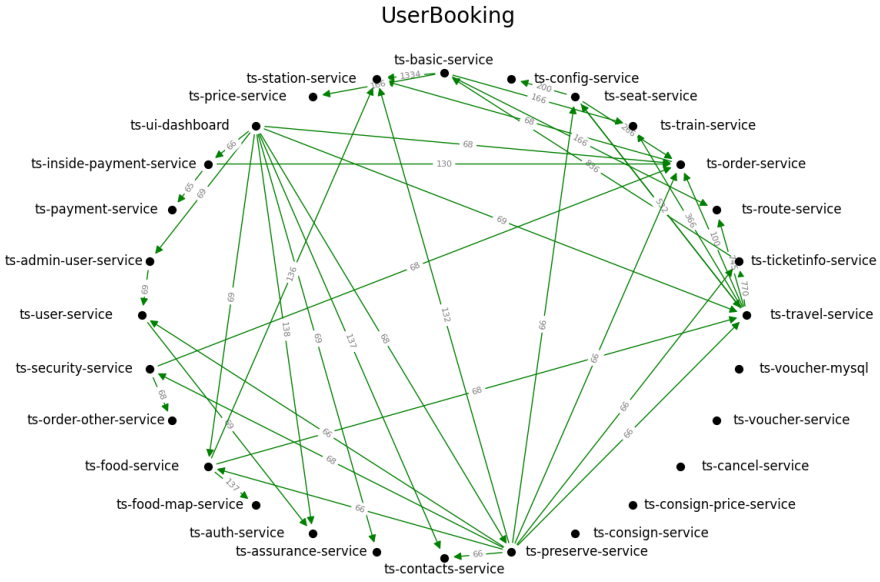

Figure 5.7. *Aggregated call graph of all instances of UserBooking*

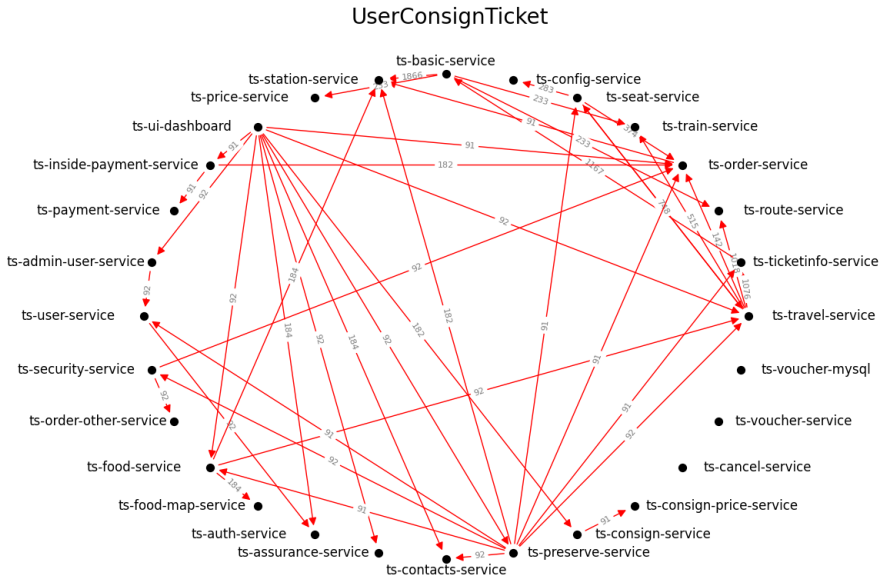

Figure 5.8. *Aggregated call graph of all instances of UserConsignTicket*

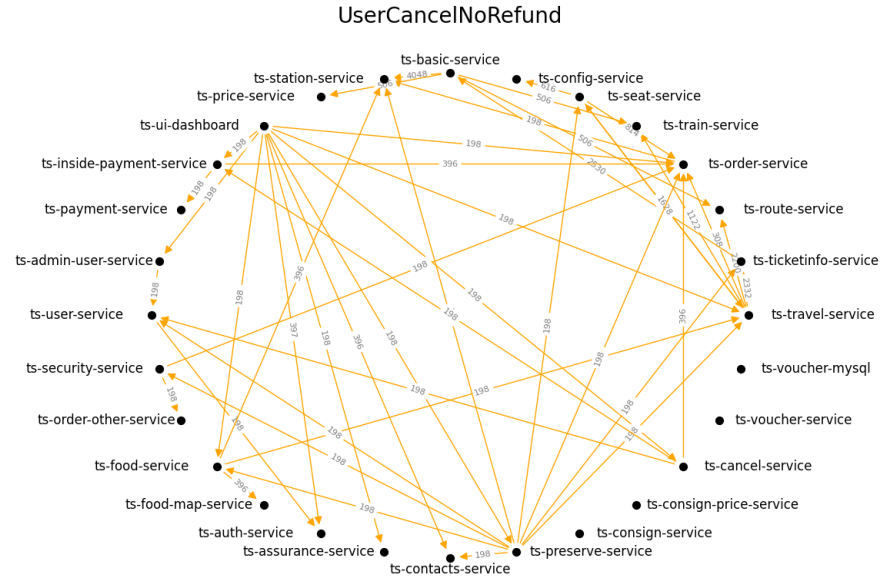

Figure 5.9. *Aggregated call graph of all instances of UserCancelNoRefund*

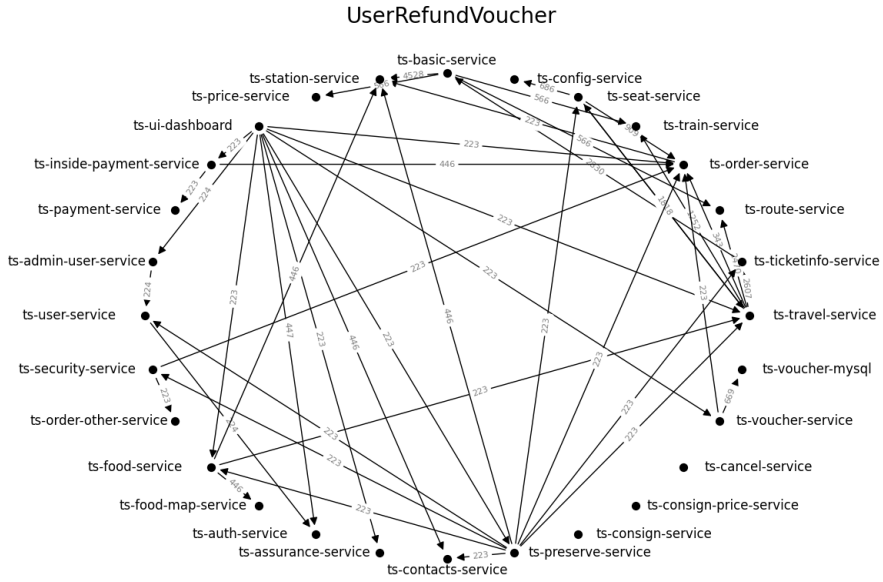

Figure 5.10. *Aggregated call graph of all instances of UserRefundVoucher*

5.3.2 Frontend integration - results

According to official documentation of all TrainTicket services on Github[7], `ts-ui-dashboard` is the only service responsible for the UI of TrainTicket, so the detector of Frontend Integration pattern was provided only with this service as 'service designated as frontend service' (see detector specification in section 5.1.2). The results for each business process/user are as follows:

UserBooking In all 70 run instances `ts-ui-dashboard` was successfully detected as potential Frontend service and *not* returned as Frontend Integration violator. No other services were reported as Frontend services (as expected), with the exception of `UserBooking_83ed2f64-c5a9-46b7-af7f-7a4f23259c71` instance - in that case 11 additional services were reported, but examination of locust log for that instance revealed that it was the last instance run because during its execution locust timed-out when it was performing the first task (logging in), so naturally most services were still unused.

UserNoLogin All 199 instances fulfill the pattern - only `ts-ui-dashboard` is reported as potential Frontend service and it is *not* a violator.

[7] https://github.com/FudanSELab/train-ticket/wiki/Service-Guide-and-API-Reference

UserConsignTicket All 92 instances fulfill the pattern - only `ts-ui-dashboard` is reported as potential Frontend service and it is *not* a violator.

UserCancelNoRefund All 199 instances fulfill the pattern - only `ts-ui-dashboard` is reported as potential Frontend service and it is *not* a violator.

UserRefundVoucher All 224 instances fulfill the pattern - only `ts-ui-dashboard` is reported as potential Frontend service and it is *not* a violator.

We should also recall that, as described in section 5.2.2, our User did not interact with the system using the User Interface, but sent call to endpoints that actually performed the tasks directly, so our graphs do not represent the full interactions of `ts-ui-dashboard` with other services. However, if for some reason some other service needed to call `ts-ui-dashboard` to carry out some business logic, than it would of course do it regardless of how the service's own operation was triggered - by a call from UI or by direct call from User, so detection of Frontend Integration pattern is still a valid task in this case.

5.3.3 Information Holder Resource - results

By going through a list of all services for which Istio has generated access logs, we notice that most of services have a corresponding `-mongo` service as well (in one case there is `-mysql` instead), which shows that most such services have a database associated with them. Thus it makes sense to provide all `-mongo` services to the detector as 'services designated as database services (see specification in section 5.1.2).

Since there are many database services and user instances, enumerating all detections is not feasible and so the detection results are summarized in tables 5.2 and 5.3 by presenting counts of each User's instance that detect a specific service/service pair.

The results are surprising. Most of pairs of IHR and DB services that we identified and expected to see in the reports have only been detected for one particular User instance of one particular User (UserBooking). Otherwise these services did not receive any calls and are not even part of the SDG for other User instances (see how those DB services that have 1 count in table 5.2 have 1 count missing in table 5.3, while all other DB services have full counts in table 5.3). The only exception to this trend is the pair of `ts-voucher-service` and `ts-voucher-mysql`, which appear to have calls between them in all instances of UserRefundVoucher.[8]

This could lead us to assume that the calls to `-mongo` services are some kind of initialization procedure which happens when the system is started if the User instance that

[8] In this context 'all' can mean 'all but one' because the last User Instance is most likely aborted early due to locust timeout, as explained before in section 5.3.2.

Table 5.2. *Results of IHR pattern detection: IHR-DB pairs (correct pairs have a check-mark).*

	Service		User				
	IHR	DB	NoLogin	Booking	ConsignTicket	CancelNoRefund	RefundVoucher
IHR-DB pattern confirmed							
✓	ts-config-service	ts-config-mongo	0/263	1/70	0/92	0/199	0/224
✓	ts-price-service	ts-price-mongo	0/263	1/70	0/92	0/199	0/224
✓	ts-inside-payment-service	ts-inside-payment-mongo	0/263	1/70	0/92	0/199	0/224
✓	ts-travel2-service	ts-travel2-mongo	0/263	1/70	0/92	0/199	0/224
✓	ts-food-map-service	ts-food-map-mongo	0/263	1/70	0/92	0/199	0/224
✓	ts-station-service	ts-station-mongo	0/263	1/70	0/92	0/199	0/224
✓	ts-payment-service	ts-payment-mongo	0/263	1/70	0/92	0/199	0/224
✓	ts-order-service	ts-order-mongo	0/263	1/70	0/92	0/199	0/224
✓	ts-route-service	ts-route-mongo	0/263	1/70	0/92	0/199	0/224
✓	ts-security-service	ts-security-mongo	0/263	1/70	0/92	0/199	0/224
✓	ts-auth-service	ts-auth-mongo	0/263	1/70	0/92	0/199	0/224
✓	ts-contacts-service	ts-contacts-mongo	0/263	1/70	0/92	0/199	0/224
✓	ts-order-other-service	ts-order-other-mongo	0/263	1/70	0/92	0/199	0/224
✓	ts-ticket-office-service	ts-ticket-office-mongo	0/263	1/70	0/92	0/199	0/224
✓	ts-consign-price-service	ts-consign-price-mongo	0/263	1/70	0/92	0/199	0/224
✓	ts-train-service	ts-train-mongo	0/263	1/70	0/92	0/199	0/224
✓	ts-user-service	ts-user-mongo	0/263	1/70	0/92	0/199	0/224
✓	ts-voucher-service	ts-voucher-mysql	0/263	1/70	0/92	0/199	0/224
	ts-food-service	ts-food-map-service	0/263	1/70	0/92	0/199	0/224
	ts-consign-service	ts-consign-price-service	0/263	0/70	91/92	0/199	0/224
	ts-ui-dashboard	ts-auth-service	0/263	0/70	0/92	1/199	0/224
	ts-basic-service	ts-station-service	56/263	0/70	0/92	0/199	0/224
IHR-DB pattern violated							
✓	ts-voucher-service	ts-voucher-mysql	0/263	0/70	0/92	0/199	223/224
✓	ts-travel-service	ts-travel-mongo	0/263	1/70	0/92	0/199	0/224
	ts-travel-service	ts-route-service	56/263	1/70	0/92	0/199	0/224
	ts-ui-dashboard	ts-verification-code-service	0/263	1/70	0/92	0/199	0/224
	ts-ui-dashboard	ts-assurance-service	0/263	69/70	92/92	198/199	223/224
	ts-ui-dashboard	ts-contacts-service	0/263	3/70	0/92	0/199	0/224
	ts-basic-service	ts-price-service	207/263	68/70	92/92	198/199	223/224
	ts-basic-service	ts-station-service	207/263	1/70	0/92	0/199	0/224
	ts-seat-service	ts-config-service	207/263	68/70	92/92	198/199	223/224
	ts-preserve-service	ts-food-service	0/263	1/70	0/92	0/199	0/224
	ts-security-service	ts-order-other-service	0/263	68/70	92/92	198/199	223/224
	ts-food-service	ts-food-map-service	0/263	68/70	92/92	198/199	223/224
	ts-inside-payment-service	ts-payment-service	0/263	65/70	91/92	198/199	223/224

Table 5.3. *Results of IHR pattern detection: DB services with no IHR.*

DB Service	User				
	NoLogin	Booking	ConsignTicket	CancelNoRefund	RefundVoucher
ts-delivery-mongo	263/263	70/70	92/92	199/199	224/224
ts-notification-mongo	263/263	70/70	92/92	199/199	224/224
ts-assurance-mongo	263/263	70/70	92/92	199/199	224/224
ts-consign-mongo	263/263	70/70	92/92	199/199	224/224
ts-food-mongo	263/263	70/70	92/92	199/199	224/224
ts-config-mongo	263/263	69/70	92/92	199/199	224/224
ts-user-mongo	263/263	69/70	92/92	199/199	224/224
ts-order-mongo	263/263	69/70	92/92	199/199	224/224
ts-voucher-mysql	263/263	69/70	92/92	199/199	1/224
ts-travel2-mongo	263/263	69/70	92/92	199/199	224/224
ts-price-mongo	263/263	69/70	92/92	199/199	224/224
ts-train-mongo	263/263	69/70	92/92	199/199	224/224
ts-station-mongo	263/263	69/70	92/92	199/199	224/224
ts-order-other-mongo	263/263	69/70	92/92	199/199	224/224
ts-contacts-mongo	263/263	69/70	92/92	199/199	224/224
ts-route-mongo	263/263	69/70	92/92	199/199	224/224
ts-inside-payment-mongo	263/263	69/70	92/92	199/199	224/224
ts-consign-price-mongo	263/263	69/70	92/92	199/199	224/224
ts-food-map-mongo	263/263	69/70	92/92	199/199	224/224
ts-security-mongo	263/263	69/70	92/92	199/199	224/224
ts-ticket-office-mongo	263/263	69/70	92/92	199/199	224/224
ts-travel-mongo	263/263	69/70	92/92	199/199	224/224
ts-auth-mongo	263/263	69/70	92/92	199/199	224/224
ts-payment-mongo	263/263	69/70	92/92	199/199	224/224

detected these calls was the very first instance that was run in the whole experiment. However, although UserBooking was indeed the first User processed by locust, this particular User instance of UserBooking was actually the last spawned one, during which locust terminated the execution of UserBooking. Thus it appears that these DB services are spawned but actually not used. These numbers match our results from [13], where Users were run simultaneously, but most `-mongo` services also had only 1 or 2 calls throughout the whole load test as well. We intend to expand the work from this thesis into a paper in the near future, and so we will investigate this result further.

We can also note that this detector will inevitably have false positives - a call chain has to terminate somewhere (otherwise it forms a cycle, which is a more serious violation that is not studied in this thesis), so the last and second-to-last services in a chain will satisfy our graph definition of IHR-DB pair. These are the entries in table 5.2 that are not marked with a checkmark. Even though they are not an actual IHR-DB pair, we can still note that there is no 'surprising' activity in this pairs: for example, `ts-consign-service` only calls `ts-consign-price-service` during UserConsignTicket, similarly `ts-inside-payment-service` calls `ts-payment-service` in all Users except UserNoLogin, since it is the only User that does not perform booking.

Since we also designed this pattern detector to check if DB services make calls, we shall note now that no DB services made calls in this experiment.

5.3.4 Request Bundle - results

As described in detector specification in Section 5.1.2, Request Bundle detection can be performed on Service level or Endpoint level. Since detections of Endpoint level would be a subset of Service Level detections, we note now that for this experiment these sets were equal and only present Endpoint level results for TrainTicket.

There are many Request Bundles and so they are divided among table 5.4 and 5.5. First column shows the calling service, second the called service and the called endpoint, third shows the amount of consecutive calls that make up the bundle, forth tells how many bundles of such size were detected in total (in some case there were several during a single User Instance), further columns tell how many instances of each User detected the bundle.

There are many pairs of services that break the pattern and have bundles of calls that happen many times during the business process execution. Several bundles occur in most of the Users and User Instances, such as `ts-travel-service` calling `ts-route-service` two or five times in a row, which is observed several times for each User Instance; `ts-basic-service` calling `ts-station-service` twice in row; `ts-preserve-service` and `ts-food-service` each calling `ts-station-service` twice in a row.

We notice that while `ts-travel-service` called `ts-route-service` twice in a row during some instances of UserNoLogin, in all other Users this call happens 5 times in a row, so either logging in or making an order is implemented inefficiently due to absence of Request Bundle in the second service. There are also several bundles in 5.5 that occur only in Users that perform logging in and booking, however they occur in all such Users, so these bundles also represent calls that can be optimized. In addition, `ts-consign-service` is implemented inefficiently, since all instances of UserConsignTicket have a request bundle with this service.

We assume that most bundles have only two consecutive calls due to the fact that these two calls are related to departure and arrival stations, or to forward and return journeys.

Also, most bundles that appear in the results have been caused by one instance of UserBooking - this is the same instance the gave surprising outputs for Information Holder Resource. Apparently is was some kind of anomalous User instance that caused some unexpected behavior in the system. Interestingly, User instances of other Users which were aborted at timeout did not cause similar behavior.

Table 5.4. *Results of Request Bundle pattern detection: Part I*

a-service	b-service endpoint	#calls	#bundles	User				
				NoLogin	Booking	ConsignTicket	CancelNoRefund	RefundVoucher
ts-travel-service	ts-ticketinfo-service /api/v1/ticketinfoservice/ticketinfo	2	115	0/263	1/70	0/92	1/199	0/224
ts-travel-service	ts-ticketinfo-service /api/v1/ticketinfoservice/ticketinfo	3	18	0/263	1/70	0/92	0/199	0/224
ts-travel-service	ts-ticketinfo-service /api/v1/ticketinfoservice/ticketinfo	5	6	0/263	1/70	0/92	0/199	0/224
ts-travel-service	ts-ticketinfo-service /api/v1/ticketinfoservice/ticketinfo	4	4	0/263	1/70	0/92	0/199	0/224
ts-travel-service	ts-route-service /api/v1/routeservice/routes	5	1115	261/263	70/70	91/92	198/199	222/224
ts-travel-service	ts-route-service /api/v1/routeservice/routes	2	400	147/263	1/70	0/92	0/199	0/224
ts-travel-service	ts-route-service /api/v1/routeservice/routes	20	1	0/263	1/70	0/92	0/199	0/224
ts-travel-service	ts-route-service /api/v1/routeservice/routes	4	9	0/263	1/70	1/92	0/199	0/224
ts-travel-service	ts-train-service /api/v1/trainservice/trains	2	29	0/263	1/70	0/92	0/199	0/224
ts-travel-service	ts-train-service /api/v1/trainservice/trains	3	2	0/263	1/70	0/92	0/199	0/224
ts-travel-service	ts-train-service /api/v1/trainservice/trains	5	1	0/263	1/70	0/92	0/199	0/224
ts-travel-service	ts-seat-service /api/v1/seatservice/seats	2	14	0/263	1/70	0/92	0/199	0/224
ts-travel-service	ts-seat-service /api/v1/seatservice/seats	3	1	0/263	1/70	0/92	0/199	0/224
ts-travel-service	ts-seat-service /api/v1/seatservice/seats	4	2	0/263	1/70	0/92	0/199	0/224
ts-travel-service	ts-order-service /api/v1/orderservice/order	2	4	0/263	1/70	0/92	0/199	0/224
ts-travel-service	ts-order-service /api/v1/orderservice/order	5	1	0/263	1/70	0/92	0/199	0/224
ts-seat-service	ts-travel-service /api/v1/travelservice/routes	5	1	0/263	1/70	0/92	0/199	0/224
ts-seat-service	ts-travel-service /api/v1/travelservice/routes	2	20	0/263	1/70	0/92	0/199	0/224
ts-seat-service	ts-travel-service /api/v1/travelservice/routes	3	1	0/263	1/70	0/92	0/199	0/224
ts-seat-service	ts-travel-service /api/v1/travelservice/routes	4	1	0/263	1/70	0/92	0/199	0/224
ts-seat-service	ts-travel-service /api/v1/travelservice/train_types	2	24	0/263	1/70	0/92	0/199	0/224
ts-seat-service	ts-travel-service /api/v1/travelservice/train_types	3	1	0/263	1/70	0/92	0/199	0/224
ts-seat-service	ts-travel-service /api/v1/travelservice/train_types	5	1	0/263	1/70	0/92	0/199	0/224
ts-seat-service	ts-config-service /api/v1/configservice/configs	2	9	0/263	1/70	0/92	0/199	0/224
ts-seat-service	ts-config-service /api/v1/configservice/configs	3	1	0/263	1/70	0/92	0/199	0/224
ts-seat-service	ts-config-service /api/v1/configservice/configs	5	1	0/263	1/70	0/92	0/199	0/224
ts-seat-service	ts-order-service /api/v1/orderservice/order	2	20	0/263	1/70	0/92	0/199	0/224
ts-seat-service	ts-order-service /api/v1/orderservice/order	5	1	0/263	1/70	0/92	0/199	0/224

Table 5.5. *Results of Request Bundle pattern detection: Part II*

a-service	b-service endpoint	#calls	#bundles	User				
				NoLogin	Booking	ConsignTicket	CancelNoRefund	RefundVoucher
ts-ui-dashboard	ts-travel-service /api/v1/travelservice/trips	4	1	0/263	1/70	0/92	0/199	0/224
ts-ui-dashboard	ts-travel-service /api/v1/travelservice/trips	5	2	0/263	1/70	0/92	0/199	0/224
ts-ui-dashboard	ts-travel-service /api/v1/travelservice/trips	2	2	0/263	1/70	0/92	0/199	0/224
ts-ui-dashboard	ts-contacts-service /api/v1/contactservice/contacts	2	581	0/263	68/70	92/92	198/199	223/224
ts-ui-dashboard	ts-consign-service /api/v1/consignservice/consigns	2	91	0/263	0/70	91/92	0/199	0/224
ts-ui-dashboard	ts-auth-service /api/v1/users/login	2	1	0/263	1/70	0/92	0/199	0/224
ts-basic-service	ts-station-service /api/v1/stationservice/stations	2	5533	207/263	69/70	92/92	198/199	223/224
ts-basic-service	ts-station-service /api/v1/stationservice/stations	3	54	0/263	1/70	0/92	0/199	0/224
ts-basic-service	ts-station-service /api/v1/stationservice/stations	4	14	0/263	1/70	0/92	0/199	0/224
ts-basic-service	ts-station-service /api/v1/stationservice/stations	10	1	0/263	1/70	0/92	0/199	0/224
ts-basic-service	ts-station-service /api/v1/stationservice/stations	5	7	0/263	1/70	0/92	0/199	0/224
ts-basic-service	ts-route-service /api/v1/routeservice/routes	5	1	0/263	1/70	0/92	0/199	0/224
ts-basic-service	ts-route-service /api/v1/routeservice/routes	2	2	0/263	1/70	0/92	0/199	0/224
ts-basic-service	ts-price-service /api/v1/priceservice/prices	2	3	0/263	1/70	0/92	0/199	0/224
ts-basic-service	ts-price-service /api/v1/priceservice/prices	5	1	0/263	1/70	0/92	0/199	0/224
ts-basic-service	ts-train-service /api/v1/trainservice/trains	2	6	0/263	1/70	0/92	0/199	0/224
ts-ticketinfo-service	ts-basic-service /api/v1/basicservice/basic	2	119	0/263	1/70	0/92	1/199	0/224
ts-ticketinfo-service	ts-basic-service /api/v1/basicservice/basic	3	14	0/263	1/70	0/92	0/199	0/224
ts-ticketinfo-service	ts-basic-service /api/v1/basicservice/basic	5	4	0/263	1/70	0/92	0/199	0/224
ts-ticketinfo-service	ts-basic-service /api/v1/basicservice/basic	4	1	0/263	1/70	0/92	0/199	0/224
ts-preserve-service	ts-station-service /api/v1/stationservice/stations	2	579	0/263	67/70	91/92	198/199	223/224
ts-food-service	ts-station-service /api/v1/stationservice/stations	2	581	0/263	68/70	92/92	198/199	223/224
ts-cancel-service	ts-order-service /api/v1/orderservice/order	2	198	0/263	0/70	0/92	198/199	0/224
ts-order-service	ts-order-mongo /	2	1	0/263	1/70	0/92	0/199	0/224
ts-station-service	ts-station-mongo /	4	1	0/263	1/70	0/92	0/199	0/224
ts-train-service	ts-train-mongo /	2	1	0/263	1/70	0/92	0/199	0/224
ts-price-service	ts-price-mongo /	3	1	0/263	1/70	0/92	0/199	0/224
ts-travel-service	ts-route-service /api/v1/routeservice/routes	3	21	1/263	1/70	0/92	0/199	1/224
ts-route-service	ts-route-mongo /	3	1	0/263	1/70	0/92	0/199	0/224
ts-config-service	ts-config-mongo /	3	1	0/263	1/70	0/92	0/199	0/224

6. DISCUSSION

It is interesting to note that the vast majority of MAPs can be detected by tools, even if there are no tools that analyze them all. Foundation patterns are the only group of MAPs where no tools implemented their detection, except for the "API description." However, few other patterns could be technically detected, and therefore tools could implement them. Other pattern categories have rather reasonable coverage by tools.

Regarding **RQ1**, we found 59 tools available for the detection task. We listed these tools in Tables 4.2 and 4.3. These tables also divided the tools based on open-source availability. We further categorized these tools based on patterns they identify in Table 4.4. However, not all tools had available information on which patterns they could detect.

With regards to **RQ2**, almost no tool identifies the patterns directly. The extracted results must be post-interpreted by users to identify these patterns from the provided information. For example, tools discovering the pattern 'Semantic Versioning Identifier' do not simply tell if the project has followed the SemVer specification[1], but instead tell the correct identifier (increment) based on changes in the source, and it is up to the developer/researcher to compare it to the one actually used. This is a missing step to better terminology unification, establishment, and automation in the domain, which some practitioners could desire. Still, there are notable gaps and improvement opportunities. Despite 59 tools found, there is no outstanding tool with respect to the detection coverage of a number of patterns. Tools must be combined to address different types of patterns. Table 4.4 outlines identified gaps that quality assurance tools should fill to provide better quality measures through integration into a single solution. Some conventional API testing tools could be used to detect specific patterns, but we excluded such tools as they need explicit scripting. As an example, in Postman [56], it is possible to extract request/response headers from HTTP calls, and from the headers, users can write tests to see if it contains API Key or Version information, etc.

Related to **RQ3**, identified tools are predominantly based on static code analysis, and more advanced techniques might be necessary to detect some patterns as depicted by Table 4.4. This table also shows that some patterns that could be detected by the techniques we identified are not yet recognized by the pool of tools we found, which opens

[1] www.semver.org

opportunities. Some of the patterns currently require manual input to be determined. However, this opens questions about whether other techniques could be considered to address these patterns. For instance, *"Rate Plan"* and *"SLA"* are about the legal use of the API, and perhaps organizational policies need to be taken into account. Still, these policies are not in a machine-readable format, and there is no guarantee these are enforced; thus, more advanced mechanisms would need to be considered to allow combinations of static or dynamic analysis with organizational policies.

When it comes to **RQ4**, we successfully implemented a tool that detects the three selected patterns - Frontend Integration, Information Holder Resource and Request Bundle. We have used Python and its library NetworkX. We notice that part of code actually processing the graphs is relatively small, and the part reading and parsing the logs is the more complicated and longer part of the code.

We also performed an experiment on pattern detection on a microservice system TrainTicket to answer **RQ5**. We achieved several insights into the system. Firstly, the Frontend Integration pattern is conformed to, with no exceptional cases. Secondly, Information Holder Resource and Request Bundle patterns are not followed, with almost every service spawning a MongoDB counterpart which appears to not be used (or at least not logged properly) and only functioning database being `ts-voucher-mysql`, as well as several service pairs having inefficient consecutive calls. Most of such pairs of calls seem to be related to activity of logging in or booking a ticket given the detection distribution among Users. We also witnessed an anomalous User instance, which, due to its timeout, appears to have caused a lot of activity in the system that got detected by our tool in form of IHR and Request Bundle pattern violations. We could not investigate the activity of that instance closely in the scope of this thesis, however we note that having a bug in one of the services that can cause many unnecessary calls to happen all over the system is one of the worst situations for developers to find themselves in, since if this bug occurs in a deployed system, it could overload it and lead to temporary shutdown, which could lead to disrupted business, money and customer loss.

The results of this work can be useful for researchers that can investigate different techniques for detecting patterns. This review's outcomes could also be useful to practitioners who can access the list of tools and our own tool that automatically detect patterns and eventually integrate them in continuous quality control models [57]. Finally, results might be beneficial to tool providers that might extend their solutions to detect a large number of patterns or integrate them into DevOps pipelines [58].

7. CONCLUSION

This thesis considered Microservice API patterns (MAP) and their recognition by available quality assurance tools. We performed a Multivocal Literature Review to identify existing tools to detect MAPs, as well as considered the potential of developing such tools using several techniques. We have identified 59 tools that address 34 MAPs out of 46. We did not find a specific tool that would surpass others, and thus a combination of tools is necessary to cover a broader spectrum of MAPs. Yet, not complete coverage exists considering our search results.

We attempted to bridge the gap by providing a tool that reconstructs a Service Dependency Graph from telemetry logs to detect three MAPs - Frontend Integration, Information Holder Resource and Request Bundle. We also performed a case study on a microservice benchmark system TrainTicket. Results of the case study provided important insights into the systems' flawed architecture.

REFERENCES

[1] Wedyan, F. and Abufakher, S. Impact of design patterns on software quality: a systematic literature review. *IET Software* 14.1 (2020), pp. 1–17.

[2] Taibi, D., Lenarduzzi, V. and Pahl, C. Architectural Patterns for Microservices: A Systematic Mapping Study. *Proceedings of the 8th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,* INSTICC. SciTePress, 2018, pp. 221–232. ISBN: 978-989-758-295-0. DOI: 10.5220/0006798302210232.

[3] Zimmermann, O., Stocker, M., Lübke, D., Pautasso, C. and Zdun, U. Introduction to Microservice API Patterns (MAP). *International Conference on Microservices (Microservices 2019)*. 2019. DOI: 10.4230/OASIcs.Microservices.2017/2019.4.

[4] Wiggins, A. *The Twelve-Factor App*. URL: https://12factor.net/ (visited on 11/11/2022).

[5] Lewis, J. and Fowler, M. *MicroServices*. Mar. 2014. URL: www.martinfowler.com/articles/microservices.html (visited on 11/11/2022).

[6] Newman, S. *Building Microservices*. O'Reilly Media, Inc., 2015. ISBN: 9781491950357.

[7] Cerny, T., Svacina, J., Das, D., Bushong, V., Bures, M., Tisnovsky, P., Frajtak, K., Shin, D. and Huang, J. On Code Analysis Opportunities and Challenges for Enterprise Systems and Microservices. *IEEE Access* (2020), pp. 1–22. DOI: 10.1109/ACCESS.2020.3019985.

[8] Walker, A., Das, D. and Cerný, T. Automated Code-Smell Detection in Microservices Through Static Analysis: A Case Study. *Applied Sciences* (2020).

[9] Saarimäki, N., Baldassarre, M. T., Lenarduzzi, V. and Romano, S. On the Accuracy of SonarQube Technical Debt Remediation Time. *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (2019), pp. 317–324.

[10] Soldani, J., Muntoni, G., Neri, D. and Brogi, A. The μTOSCA toolchain: Mining, analyzing, and refactoring microservice-based architectures. *Software: Practice and Experience* 51 (2021), pp. 1591–1621.

[11] Garousi, V., Felderer, M. and Mäntylä, M. V. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology* 106 (2019), pp. 101–121. ISSN: 0950-5849.

[12] Bakhtin, A., Al Maruf, A., Cerny, T. and Taibi, D. Survey on Tools and Techniques Detecting Microservice API Patterns. *2022 IEEE International Conference on Services Computing (SCC)*. 2022, pp. 31–38. DOI: 10.1109/SCC55611.2022.00018.

[13] Maruf, A. A., Bakhtin, A., Cerný, T. and Taibi, D. Using Microservice Telemetry Data for System Dynamic Analysis. *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)* (2022), pp. 29–38.

[14] Zimmermann, O. *Microservice API patterns*. URL: https://www.microservice-api-patterns.org/ (visited on 02/04/2022).

[15] Taibi, D., Lenarduzzi, V. and Pahl, C. Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation. *IEEE Cloud Computing* 4.5 (Sept. 2017), pp. 22–32. ISSN: 2325-6095. DOI: 10.1109/MCC.2017.4250931.

[16] Auer, F., Lenarduzzi, V., Felderer, M. and Taibi, D. From monolithic systems to Microservices: An assessment framework. *Information and Software Technology* 137 (2021), p. 106600. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2021.106600. URL: https://www.sciencedirect.com/science/article/pii/S0950584921000793.

[17] Weske, M. *Business Process Management: Concepts, Languages, Architectures*. Springer Berlin Heidelberg, 2012. ISBN: 9783642286162. URL: https://books.google.fi/books?id=-D5tpT5Xz8oC.

[18] Panichella, S., Imranur, M. R. and Taibi, D. Structural Coupling for Microservices. *11th International Conference on Cloud Computing and Services Science*. Apr. 2021.

[19] Pigazzini, I., Arcelli Fontana, F., Lenarduzzi, V. and Taibi, D. Towards Microservice Smells Detection. *Proceedings of the 3rd International Conference on Technical Debt*. TechDebt '20. Seoul, Republic of Korea, 2020, pp. 92–97. ISBN: 9781450379601. DOI: 10.1145/3387906.3388625.

[20] Dwivedi, A. K., Tirkey, A. and Rath, S. K. Software design pattern mining using classification-based techniques. *Frontiers of Computer Science* 12.5 (2018), pp. 908–922.

[21] F. Farias, M. A. de, Novais, R., Júnior, M. C., Silva Carvalho, L. P. da, Mendonça, M. and Spınola, R. O. A Systematic Mapping Study on Mining Software Repositories. *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. SAC '16. Pisa, Italy, 2016, pp. 1472–1479. ISBN: 9781450337397.

[22] Carnell, J. and Sánchez, I. H. *Spring microservices in action*. Simon and Schuster, 2021.

[23] Balalaie, A., Heydarnoori, A., Jamshidi, P., Tamburri, D. A. and Lynn, T. Microservices migration patterns. *Software: Practice and Experience* 48.11 (2018), pp. 2019–2042.

[24] Taibi, D., El Ioini, N., Claus, P. and Niederkofler, J. R. S. Patterns for Serverless Functions (Function-as-a-Service): A Multivocal Literature Review. *10th International Conference on Cloud Computing and Services Science (CLOSER)*. 2020, pp. 181–192. ISBN: 978-989-758-424-4. DOI: 10.5220/0009578501810192.

[25] Zimmermann, O., Stocker, M., Lübke, D., Pautasso, C. and Zdun, U. Introduction to Microservice API Patterns (MAP). *Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019)*. 2020, 4:1–4:17.

[26] Zimmermann, O., Lübke, D., Zdun, U., Pautasso, C. and Stocker, M. Interface Responsibility Patterns: Processing Resources and Operation Responsibilities. *European Conference on Pattern Languages of Programs 2020*. EuroPLoP '20. 2020.

[27] Zimmermann, O., Pautasso, C., Lübke, D., Zdun, U. and Stocker, M. Data-Oriented Interface Responsibility Patterns: Types of Information Holder Resources. EuroPLoP '20. 2020.

[28] Lübke, D., Zimmermann, O., Pautasso, C., Zdun, U. and Stocker, M. Interface Evolution Patterns: Balancing Compatibility and Extensibility across Service Life Cycles. *Proceedings of the 24th European Conference on Pattern Languages of Programs*. EuroPLop '19. Irsee, Germany: Association for Computing Machinery, 2019. ISBN: 9781450362061.

[29] Stocker, M., Zimmermann, O., Lübke, D., Zdun, U. and Pautasso, C. Interface Quality Patterns – Communicating and Improving the Quality of Microservices APIs. *23rd European Conference on Pattern Languages of Programs 2018*. July 2018. DOI: 10.1145/3282308.3282319.

[30] Zimmermann, O., Stocker, M., Lübke, D. and Zdun, U. Interface Representation Patterns - Crafting and Consuming Message-Based Remote APIs. *22nd European Conference on Pattern Languages of Programs (EuroPLoP 2017)*. July 2017, pp. 1–36. DOI: 10.1145/3147704.3147734.

[31] Bushong, V., Abdelfattah, A. S., Maruf, A. A., Das, D., Lehman, A., Jaroszewski, E., Coffey, M., Cerny, T., Frajtak, K., Tisnovsky, P. and Bures, M. On Microservice Analysis and Architecture Evolution: A Systematic Mapping Study. *Applied Sciences* 11.17 (2021). DOI: 10.3390/app11177856.

[32] Taibi, D. and Systä, K. From Monolithic Systems to Microservices: A Decomposition Framework based on Process Mining. *Proceedings of the 9th International Conference on Cloud Computing and Services Science (CLOSER)*. INSTICC. SciTePress, 2019, pp. 153–164. DOI: 10.5220/0007755901530164.

[33] Taibi, D., Lenarduzzi, V. and Pahl, C. Microservices Anti-patterns: A Taxonomy. *Microservices: Science and Engineering*. Cham: Springer International Publishing, 2020, pp. 111–128. DOI: 10.1007/978-3-030-31646-4_5.

[34] Taibi, D. and Lenarduzzi, V. On the Definition of Microservice Bad Smells. *IEEE Software* 35.3 (2018), pp. 56–62. DOI: 10.1109/MS.2018.2141031.

[35] Bento, A., Correia, J., Filipe, R., Araújo, F. and Cardoso, J. S. Automated Analysis of Distributed Tracing: Challenges and Research Directions. *Journal of Grid Computing* 19 (2021), pp. 1–15.

[36] Ma, S.-P., Fan, C.-Y., Chuang, Y., Lee, W.-T., Lee, S.-J. and Hsueh, N.-L. Using Service Dependency Graph to Analyze and Test Microservices. *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)* 02 (2018), pp. 81–86.

[37] Bushong, V., Sanders, R., Curtis, J., Du, M., Cerny, T., Frajtak, K., Tisnovsky, P. and Shin, D. On Log Analysis and Stack Trace Use to Improve Program Slicing. *Information Science and Applications*. Springer Singapore, Dec. 2021, (in print). DOI: Acceptedforpublication.

[38] Zhao, X., Zhang, Y., Lion, D., Ullah, M. F., Luo, Y., Yuan, D. and Stumm, M. lprof: A Non-intrusive Request Flow Profiler for Distributed Systems. *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO, 2014, pp. 629–644. ISBN: 978-1-931971-16-4.

[39] Peltonen, S., Mezzalira, L. and Taibi, D. Motivations, benefits, and issues for adopting Micro-Frontends: A Multivocal Literature Review. *Information and Software Technology* 136 (2021), p. 106571. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2021.106571.

[40] Zhou, X., Peng, X., Xie, T., Sun, J., Xu, C., Ji, C. and Zhao, W. Benchmarking Microservice Systems for Software Engineering Research. *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ICSE '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 323–324. ISBN: 9781450356633. DOI: 10.1145/3183440.3194991. URL: https://doi.org/10.1145/3183440.3194991.

[41] *GitHub - FudanSELab/train-ticket*. URL: https://github.com/FudanSELab/train-ticket/ (visited on 11/11/2022).

[42] Walker, A., Das, D. and Černý, T. Automated Code-Smell Detection in Microservices Through Static Analysis: A Case Study. *Applied Sciences* 10 (Nov. 2020). DOI: 10.3390/app10217800.

[43] Černý, T. and Taibi, D. Static analysis tools in the era of cloud-native systems. May 2022.

[44] Černý, T., Abdelfattah, A., Bushong, V., Maruf, A. and Taibi, D. Microvision: Static analysis-based approach to visualizing microservices in augmented reality. Aug. 2022, pp. 49–58. DOI: 10.1109/SOSE55356.2022.00012.

[45] Schiewe, M., Curtis, J., Bushong, V. and Černý, T. Advancing Static Code Analysis With Language-Agnostic Component Identification. *IEEE Access* 10 (Jan. 2022), pp. 1–1. DOI: 10.1109/ACCESS.2022.3160485.

[46] Bushong, V., Das, D., Maruf, A. and Černý, T. Using Static Analysis to Address Microservice Architecture Reconstruction. Nov. 2021, pp. 1199–1201. DOI: 10.1109/ASE51524.2021.9678749.

[47] *Kubernetes*. URL: https://kubernetes.io/ (visited on 11/11/2022).

[48] *Istio: Simplify observability, traffic management, security, and policy with the leading service mesh*. URL: https://istio.io/ (visited on 11/11/2022).

[49] *kind: Kubernetes-in-Docker*. URL: https://kind.sigs.k8s.io/ (visited on 11/11/2022).

[50] Avritzer, A., Menasché, D. S., Rufino, V. Q., Russo, B., Janes, A., Ferme, V., Hoorn, A. van and Schulz, H. PPTAM: Production and Performance Testing Based Application Monitoring. *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering* (2019).

[51] *Locust - A modern load testing framework*. URL: https://locust.io/ (visited on 11/11/2022).

[52] Burns, B. and Oppenheimer, D. Design Patterns for Container-based Distributed Systems. *HotCloud*. 2016.

[53] *The world's lightest, fastest service mesh*. URL: https://linkerd.io (visited on 11/11/2022).

[54] *Service mesh on Consul*. URL: https://developer.hashicorp.com/consul (visited on 11/11/2022).

[55] Marmol, V., Jnagal, R. and Hockin, T. Networking in Containers and Container Clusters. 2015.

[56] *Postman API Platform*. 2022. URL: https://www.postman.com/.

[57] Lenarduzzi, V., Stan, A. C., Taibi, D., Tosi, D. and Venters, G. A Dynamical Quality Model to Continuously Monitor Software Maintenance. *11th European Conference on Information Systems Management (ECISM2017)*. 2017.

[58] Taibi, D., Lenarduzzi, V. and Pahl, C. Continuous Architecting with Microservices and DevOps: A Systematic Mapping Study. *Cloud Computing and Services Science (CLOSER)*. 2019, pp. 126–151. ISBN: 978-3-030-29193-8. DOI: 10.1007/978-3-030-29193-8_7.