Maintenance Support for Microservice-based Systems by Specification and Identification of Microservice Antipatterns

1st Author's Name, Surname
Dept. name
Name of organization
City, Country
Email address or ORCID

2rd Author's Name, Surname
Dept. name
Name of organization
City, Country
Email address or ORCID

3nd Author's Name, Surname
Dept. name
Name of organization
City, Country
Email address or ORCID

4th Author's Name, Surname
Dept. name
Name of organization
City, Country
Email address or ORCID

***Abstract*—The software industry is currently moving from monolithic architectures into microservice-based architectures, which involve independent, reusable, and fine-grained services. However, the lack of understanding of the core concepts of microservice architectures may lead to the introduction of poorly designed solutions, called antipatterns. Microservice antipatterns may affect the quality of services and hinder the maintenance and evolution of systems. The specification and detection of microservice antipatterns could help in evaluating and assessing the design quality of such systems. Several research works studied patterns and antipatterns in microservice-based systems. However, the automatic identification of such antipatterns is still in its infancy. We propose MARS (Microservice Antipatterns Research Software), a fully-automated approach supported by a framework for specifying and identifying microservice antipatterns. Using MARS, we specify and identify 16 microservice antipatterns in 24 microservice-based systems. Results show that MARS can detect microservice antipatterns with an average precision greater than 68% and a recall greater than 78%.**

***Index Terms*—SOA; Microservices; Antipatterns; Detection; Maintenance**

I. Introduction

Microservices have already become the prevailing architectural style used in industry. Indeed, several major actors of the software industry, such as Netflix and Amazon, have already adopted this architectural style. A microservice is defined as a small service, with a single responsibility or business function, running on its own process, and communicating through lightweight mechanisms, such as Representational State Transfer Application Programming Interfaces (REST APIs) and message brokers [1], and managed by a single team.

The popularity of this architectural style still grows, mainly due to the dynamic and distributed nature of microservices, which (1) offers greater agility and operational efficiency and (2) reduces the complexity of handling applications scalability and deployment cycles wrt. monolithic applications [2].

However, the lack of understanding of the core concepts of microservice architecture and of consensual definitions of its principles may lead to the introduction of "poor" solutions to recurring design and implementation problems, called antipatterns [3]. These antipatterns may impact the quality of the microservices and of the related systems as wholes [4].

The dynamic nature of microservices makes the identification of related antipatterns difficult. Thus, despite the importance and extensive usage of microservices, no automated approach for the detection of microservice antipatterns has been proposed so far. Consequently, we propose MARS, a tool-based approach to specify and detect microservice antipatterns. We rely on a meta-model that includes the information needed to specify and apply detection rules. Using MARS, we specify 16 antipatterns and detect their occurrences within 24 microservice-based systems. We perform a manual validation of the detected occurrences in terms of precision and recall. Our results show that MARS allows to specify and detect microservice antipatterns with an average precision of 68.5% and a recall of 78.22%.

This paper reads as follows: Section II describes previous work and a catalog of microservice antipatterns. Section III presents our methodology for antipatterns detection, and describes the used detection rules. Section IV details our study design. Section V describes our results while Section VI discusses them. Section VII concludes with future work.

II. Background

A. Related Work

Several research works exist in the literature on microservice antipatterns. However, only a few concern their detection.

Pahl and Jamdi [5] conducted a systematic literature review of 21 works on microservice design published between 2014 and 2016. They proposed a characterisation framework and

used it to study and classify the works. They showed a lack of research tools supporting microservice-based systems and concluded that research on microservice architecture is novel.

Zimmerman [6] studied the literature and identified seven microservices tenets. He then compared two microservices definitions and contrasted them with SOA principles and patterns. He compiled practitioners questions and derived several research topics related to the differences between SOA and microservices architectural styles. He concluded that microservices are not entirely new but qualify as a one special implementation of the SOA paradigm.

Soldani et al. [7] identified and compared the benefits and limitations of microservices by studying the industrial *grey* literature and the design and development practices of microservices. They wanted to bridge the gap between academia and industry in terms of research focus.

Marquez and Astudillo [8] extended their previous work [9] to propose a catalog of microservice architectural patterns. They provided a list of technologies to build microservice-based systems with these patterns. They also performed a comparative analysis of SOA and microservice architectural patterns to determine whether architectural patterns are used in the development of microservice-based systems.

Taibi et al. [10] introduced a catalog and a taxonomy of microservice antipatterns resulting from the migration of monolithic systems to microservices. They built on the industrial experience of 27 interviewed practitioners. They identified 20 organisational and technical antipatterns. They studied the harmfulness level of each antipattern. They concluded that splitting a monolithic system into microservices is a critical and challenging problem.

Borges and Khan [11] selected five well-known antipatterns in microservice-based systems and proposed an algorithm to automatically detect them. They showed that their algorithm can help practitioners developing microservice-based systems by avoiding mistakes common in microservices-based projects. They applied their algorithm to only one open-source microservice-based system and discussed improvements.

Microservice antipatterns have been discussed in the literature, but very little exists on their automatic detection. To the best of our knowledge, only Borges and Khan [11] proposed an algorithm to automatically detect antipatterns in microservice-based systems. However, it only detects five antipatterns. Our approach encompass more antipatterns. Also, we propose a general approach and systematically validate its detection results on a larger number of systems.

B. Catalog

We now present a catalog of several microservice antipatterns built through a systematic literature review[1] (SLR) [12]. We applied several inclusion and exclusion criteria (e.g., exclude papers not written in English and papers not related to microservice antipatterns) to obtain a total of 27 papers that studied microservice antipatterns.

We also manually analysed 67 open-source systems [13] to assess the concrete presence of the antipatterns identified in the SLR. This study informs our understanding of the antipatterns and how they could be detected in practice.

In this paper, we consider the following 16 antipatterns, which we choose because they can be detected in the source code of microservice-based systems.

1) **Wrong Cuts (WC):** This antipattern consists of microservices organised around technical layers (business, presentation, and data) instead of functional capabilities, which causes strong coupling of the microservices and impedes the delivery of new business functions.
2) **Cyclic Dependencies (CD):** This antipattern occurs when multiple services are co-dependent circularly and, thus, no longer independent, which goes against the very definition of microservices.
3) **Mega Service (MS):** This antipattern appears when a microservice serves multiple business functions. A microservice should be manageable by a single team and bounded to a single business function.
4) **Nano Service (NS):** This antipattern results from a too fine-grained decomposition of a system, i.e., when one business function requires many microservices together.
5) **Shared Libraries (SL):** This antipattern consists of libraries and files (e.g., binaries) used by multiple microservices, which breaks their independence as they rely on a single source to fulfill their business function.
6) **Hard-coded Endpoints (HE):** This antipattern relates to URLs, IP addresses, ports, and other endpoints being hard-coded in the microservice source code and–or configuration files, which interferes with load balancing and deployment.
7) **Manual Configuration (MC):** This antipattern happens with configurations that must be manually pushed to each microservice of a system. Microservice systems evolve rapidly and their management should be automated, including their configuration.
8) **No Continuous Integration (CI) / Continuous Delivery (CD) (NCI):** Continuous integration and delivery are important for microservices to automate repetitive steps during testing and deployment. Not using CI/CD undermines the microservice architectural style, which encourages automation wherever possible.
9) **No API Gateway (NAG):** This antipattern occurs when consumer applications (mobile applications, etc.) communicate directly with microservices and must know how the whole system is decomposed and then manage endpoints and URLs for each microservice.
10) **Timeouts (TO):** This antipattern happens when timeout values are set and hard-coded in HTTP requests, which leads to spurious timeouts or unnecessary delays.
11) **Multiple Service Instances per Host (MSIH):** This

[1]The detailed process of the creation of this catalog is described in details in an accepted but not-yet-published work that we cannot cite to respect the double-blind review process.

antipattern happens when multiple microservices are deployed on a single host, which prevents their independent scaling and may cause technological conflicts inside the host.

12) **Shared Persistence (SP):** This antipattern happens when multiple microservices share a single database: they no longer own their data and cannot use the most suitable database technology for it.
13) **No API Versioning (NAV):** This antipattern happens when no information is available about a microservice version, which can break changes and force backward compatibility when deploying updates.
14) **No Health Check (NHC):** This antipattern occurs when microservices are not periodically health checked. Unavailable microservices may not be noticed and cause timeouts and errors.
15) **Local Logging (LL):** This antipattern results from microservices having their own logging mechanism, which prevents the aggregation and analyses of their logs and the monitoring and recovery of systems.
16) **Insufficient Monitoring (IM):** This antipattern relates to microservice systems performances/failures that are not tracked and cannot help maintain the systems.

III. Approach

We present MARS, a fully automatic approach and tool to detect the antipatterns described in Section II. Figure 1 shows that it takes as input a microservice-based system or a list of microservices (both either as Git repositories or local source-code folders). Then, it extracts from each microservice relevant information to perform the detection of the antipatterns, which it reifies using a dedicated meta-model. Then, we can apply MARS algorithms to detect occurrences of the antipatterns.

A. Meta-model Definition

We created a meta-model to encapsulate the information needed to apply our detection algorithms, which includes information about: the system, its Git repository, individual microservices, dependencies, source code, environment files, configuration files, deployment files, docker images, platforms, HTTP(s) requests, databases, and imports.

This meta-model allows our detection algorithms to access relevant data while being independent of the source of these data. It also allows avoiding parsing the source code of the systems and the evolution of MARS by introducing new constituents in the meta-model and the algorithms to detect new antipatterns. It also allows our algorithms to be as independent as possible of any particular technologies, for example by abstracting dependencies using the `Dependency` constituent, whether they come from Gradle, Maven, etc.

Figure 2 illustrates the meta-model constituents and their relationships. Each constituent of the meta-model is part of the detection process of one or more antipattern(s). For example, the `Configuration` constituent is used to detect the hardcoded endpoints by searching URLs inside configuration files, along with the `Code` and the `Dependency` constituents.

The `System` constituent is the root of a model, it is built either by importing a Git Repository (which is an optional constituent) or by importing a local microservice-based system from the local file system.

A `System` knows about two set of constituents: (1) `Microservices` and (2) `Dependencys`. The `Microservice` constituent represents an actual microservice in the microservice-based system. It contains data about this microservice, e.g., number of files and LOCs (e.g., used to detect Mega Service and Nano Service).

The `Dependency` constituent is referenced by both `System` and `Microservice` because it is common to have dependency files (e.g., Gradle, Maven) at both levels of a system's hierarchy. It contains data about the dependencies of the system or the particular microservice.

The `Configuration` constituent stores data gathered from the various configuration files of a microservice. It allows to search for data only in configuration files, such as framework-related variables or enabled/disabled features.

The `Environment` constituent stores data about environment variables, typically their names and values, which are a common used to dynamically inject variables into a system.

The `Deployment` constituent holds data about deployment configurations and mechanisms, it abstracts Docker files, docker-compose files, and custom deployment files added by the user. It points to `Images` (e.g., for Docker) and–or `Server` (e.g., for Amazon ECS) containing data about these particular deployment targets. For example, they help identifying microservices by extracting the `Images` from which they are derived.

The `Code` constituent is the most used constituent in MARS as it contains data about the source code of a microservice to allow MARS to retrieve source code parts. It includes:

1) Source-files: a dictionary of source-file names and paths; useful to filter test files for example.
2) Filtered files: a dictionary of filtered source files (i.e., with comments removed).
3) Imports: list of all the import statements, by source file.
4) Methods: list of method names in the source code.
5) Annotations: List of all the annotations in the code.
6) HTTP: list of all HTTP requests in the source code
7) DB: list of database queries and "create" statements as well as data-source paths.
8) Call-graph: the call graph of the source code generated by the Modisco KDM framework.

In addition to this, this constituent allows to define some important information such as the language used in the microservice.

B. Detection Rules

For each antipattern, we defined a set of detection rules[2] to detect their occurrences in a given system. A textual description of these detection rules follows.

[2]The complete set of detection rules is available online (URL hidden for double-blind review)

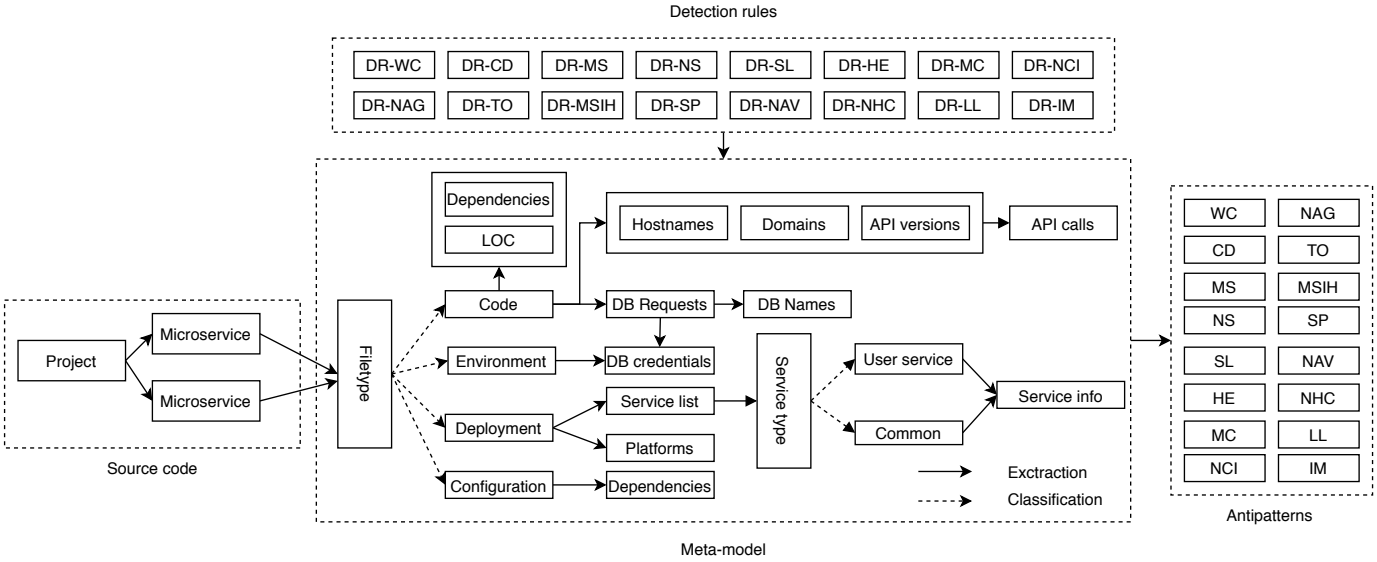

Fig. 1. Microservice Antipatterns Research Software (MARS)

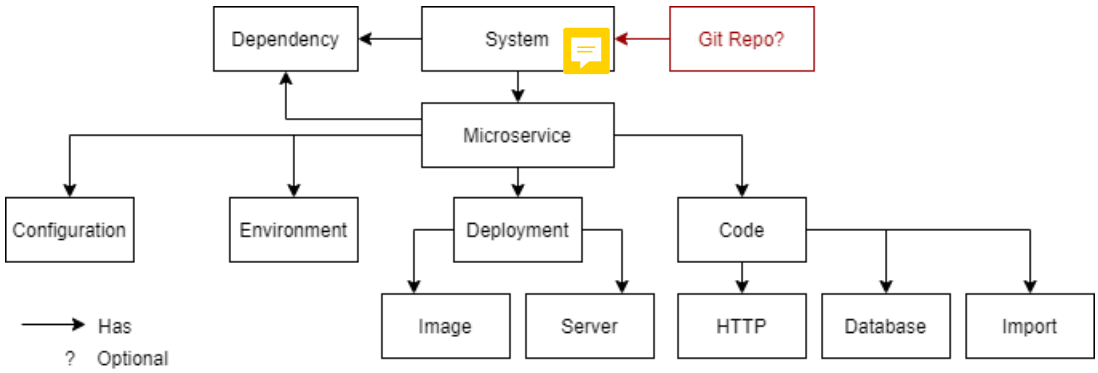

Fig. 2. MARS metamodel constituents

1) **Wrong Cuts (WC)** Microservices have one file type in the source code and connect to multiple microservices having also one file type. An example would be a microservice containing only presentation code connecting to a microservice containing only business logic code. We rely on files extensions, contents, and languages to identify this antipattern.
2) **Cyclic Dependencies (CD)** We use the call graph of the microservice-based system, which we analyse to detect circular dependencies among microservices.
3) **Mega Service (MS)** Such a microservice has more lines of code, files, and dependencies than other microservices relatively. We established the thresholds using box plots of the overall numbers per system.
4) **Nano Service (NS)** Such a microservice has less lines of code, files, and dependencies than other microservices. We computed the thresholds using again box plots.
5) **Shared Libraries (SL)** Some source files, libraries, or other artefacts of one microservice are used by others.
6) **Hardcoded Endpoints (HE)** REST API calls inside microservices source code, deployment files, configuration files, or environment files contain hard-coded IP addresses, port numbers, and–or URLs. No discovery service is used.
7) **Manual Configuration (MC)** Microservices have their own configuration files. In addition, no configuration management tools are present in the dependencies of the system and no microservice is responsible of configuration management.
8) **No CI/CD (NCI)** Configuration files and version control repositories do not contain continuous integration/delivery-related information. We rely on an extensible list of CI/CD tools to perform our analysis.
9) **No API Gateway (NAG)** Microservice source code does not contain signatures of common API gateway implementations (e.g., Netflix Zuul). No frameworks/tools are present in the dependencies of the microservices.
10) **Timeouts (TO)** Timeout values are present in REST API calls. No signatures of common circuit breaker implementations (e.g., Hystrix) are present in the source

code. No circuit breaker is present in the dependencies of the microservices.

11) **Multiple Service Instances per Host (MSIH)** The system does not use deployment technologies, such as Docker Compose. A single deployment file exists in the source code and deploys the whole system.
12) **Shared persistence (SP)** Microservices share data-source URLs. A single database is created by the system and multiple microservices use this database.
13) **No API Versioning (NAV)** Endpoints and URLs do not contain version numbers. No version information is present in the configuration files.
14) **No health check (NHC)** No "health check" or "health" endpoint exists. No common implementation of health checks is used (e.g., Springboot Actuator).
15) **Local Logging (LL)** We detect this antipattern by checking if there is (1) no distributed logging in the dependencies or (2) no common logging microservice. Each microservice has its own log file paths.
16) **Insufficient Monitoring (IM)** We detect this antipattern by looking for monitoring framework or library in the microservices dependencies (e.g., Prometheus).

C. Implementation

We implement MARS using a variety of frameworks and libraries best suited for the tasks at hand. We use Eclipse Modisco KDM[3] to parse the source code of each microservice and create a first model. We identify dependencies among services using Bibliothecary[4]. For Gradle dependencies, we use the NodeJS service Gradle Parser[5]. Finally, we combine this data to create a model of a microservice-based system conform to our meta-model, using Python. We also use Python to implement the detection rules applied to this model. We release MARS as an open-source project, whose source code and other artefacts are available online[6].

MARS was built to be extensible. We built it around a technology-agnostic meta-model to support multiple programming languages, technologies, and tools. For example, the search for libraries in MARS is supported by a configuration file in which developers can specify the libraries that they want to consider. We applied a similar approach for programming languages: to support a new programming language, say Go or JavaScript, we "only" must add the name of the desired language into a configuration file and implement few required, supporting methods.

IV. Study Design

We now discuss the design of our study.

A. Dataset

To validate our approach, rules, and tool, we apply MARS on 24 microservice-based systems written in Java. These systems are taken from a dataset of microservice-based systems available online [14]. Our dataset is described in Table I. Figure 3 and Figure 4 show microservice-based systems number of files and lines of codes respectively, plotted on a logarithmic scale for sake of clarity.

The source code of any microservice-based system contains developers' written code, artefacts, and third-party dependencies and libraries. Including third-party code would produce misleading results. Therefore, we pre-processed our dataset and excluded such code from our analysis, by filtering dependency folders, e.g., node modules, maven folders, composer vendor directories, etc.

We then applied MARS on the filtered systems and compared the detected occurrences of the antipatterns against a ground truth, i.e., instances found manually in the systems.

Fig. 3. Distribution of dataset number of files on a logarithmic scale

Fig. 4. Distribution of dataset number of lines of code on a logarithmic scale

B. Ground Truth

To build a ground truth containing instances of the antipatterns, three authors independently analysed the microservice-

[3]https://wiki.eclipse.org/MoDisco/Components/KDM
[4]https://github.com/librariesio/bibliothecary
[5]https://github.com/librariesio/gradle-parser
[6] URL hidden for double-blind review: XXX

System	# Microservices	# Files	LOCs
Spring Netflix OSS	3	17	443
FTGO	9	257	8239
LakeSide Mutual	13	55	2170
Spring Petclininc	3	25	795
Freddy's BBQ	6	35	1752
Spring cloud Movie	4	33	885
Piggymetrics	4	88	3176
Tap And Eat	6	31	576
E-commerce	3	24	756
Consul	3	38	1750
Microservice Demo	3	38	1766
Qbike	5	77	2057
Spring cloud Microservice	9	21	673
CQRS Microservice Sampler	3	26	1028
Spring boot Microservices	2	4	116
CAS microservice architecture	5	29	871
Cloud Strangler example	3	30	932
Micro company	13	55	2170
MicroService	13	42	1052
MicroService Kubernetes	3	38	1640
TeaStore	3	62	5073
Warehouse Microservice	6	222	4623
Apollo	9	68	29510
Delivery System	2	14	537

TABLE I
NUMBER OF MICROSERVICES, FILES AND LOCS PER SYSTEM

based systems of the dataset. Each author had a specific, overlapping list of antipatterns to identify in the systems.

A fourth author analysed all the systems but tried to find all instances of all antipatterns. Hence, we made sure that we have at least two opinions on each antipattern.

After independently collecting instances of the antipatterns, the four authors discussed their findings and reconciled any discrepancy. Thus, the four authors were involved in the manual validation of the antipattern instances.

Antipattern	Agreement value
Wrong Cuts	0.91
Circular dependencies	1
Mega Service	0.96
Nano Service	0.67
Shared Libraries	1
Hardcoded Endpoints	0.79
Manual Configuration	0.92
No CI / CD	1
No API Gateway	0.88
Timeouts	0.88
Multiple Service Instances per Host	0.79
Shared Persistence	0.83
No API versioning	0.88
No Health Check	0.75
Local Logging	0.83
Insufficient Monitoring	0.83
Average	**0.87**

TABLE II
OVERVIEW OF THE INTER-RATER AGREEMENT VALUES

We computed the inter-rater agreement values per antipattern, shown in Table II, and obtained an average inter-rater agreement of 0.87, which is "Excellent" or "Almost perfect" depending on authors [15], [16].

V. STUDY RESULTS

This section presents the results and observations obtained after running MARS on the 24 microservice-based systems of the dataset.

We calculated precision and recall values for each of the detected antipatterns as follow, with AP meaning "antipattern":

$$Precision = \frac{|\{existing\ APs\} \cap \{Detected\ APs\}|}{|\{Detected\ APs\}|} \quad (1)$$

$$Recall = \frac{|\{Existing\ APs\} \cap \{Detected\ APs\}|}{|\{Existing\ APs\}|} \quad (2)$$

The precision of the detection of our tool is satisfying as it varies between 35% and 100% with an average of 68.5%. The recall value is also satisfying as it varies from 33,33% to 100% with an average of 78.22%.

The achieved precision and recall values confirm the effectiveness of MARS to detect the identified antipatterns.

The detailed results of our analysis are described in Table III and Figure 5.

1. Wrong Cuts (WC)

Decomposing microservices must consider business capabilities, not technical layers. Thus, a microservice should contain all the artefacts needed to fulfil a business function (presentation code, application code, database queries, etc.).

When running MARS on the microservice-based systems in our dataset, we could not detect any system containing this antipattern. However, we found 4 instances when performing the manual analysis. This discrepancy is due to the difficulty to obtain the dependencies among microservices via static analysis [17].

Observation 1: We observe that a few systems (4/24) contain occurrences of wrong cuts. The majority of the analysed systems decompose their microservices in an appropriate way.

2. Circular Dependencies (CD)

Microservices should be independent, without strong coupling. Communication between microservices should be lightweight, e.g., using REST APIs to avoid creating a "distributed monolith" [18].

MARS found 3 occurrences of this antipattern. However, during our manual analysis, before running the tool, we found no instance. This discrepancy is due to the manual analysis of call graphs, which is very tedious. We validated the findings of MARS and confirmed the 3 occurrences. As the manual validation returned no results, we were unable to compute precision and recall for this antipattern.

System	Antipattern															
	WC	CD	MS	NS	SL	HE	MC	NCI	NAG	TO	MSIPH	SP	NAV	NHC	LL	IM
Spring Netflix OSS	-	-	-	-	-	-	-	✓	-	-	-	-	✓	(✓)	(✓)	(✓)
FTGO	-	-	✓	(-)	-	✓	✓	-	-	✓	-	-	(-)	-	-	-
LakesideMutual	(-)	-	(✓)	(✓)	-	(-)	✓	✓	✓	-	(-)	-	-	-	(✓)	(✓)
Spring Petclinic	-	-	-	-	-	-	-	-	-	-	-	✓	-	(✓)	-	-
Freddys BBQ	-	-	-	-	-	-	-	✓	✓	-	✓	(-)	✓	✓	(✓)	(✓)
Spring cloud movie	-	(✓)	-	-	-	-	-	✓	-	-	✓	-	✓	✓	-	(✓)
Piggymetrics	-	-	(✓)	-	-	-	-	-	-	-	-	(-)	✓	-	-	(✓)
Tap And Eat	-	-	-	-	-	-	-	✓	✓	-	-	-	✓	✓	(✓)	(✓)
E-commerce	-	-	-	-	-	-	-	✓	(✓)	(✓)	-	-	✓	✓	✓	✓
Consul	-	-	-	-	-	(✓)	✓	✓	(✓)	✓	-	-	✓	-	(-)	-
Microservice demo	-	-	-	-	-	-	✓	✓	(-)	-	-	-	✓	-	(✓)	(✓)
Qbike	-	-	(✓)	(-)	-	-	(-)	✓	-	-	-	✓	✓	✓	-	(✓)
Spring cloud microservice	-	-	-	✓	-	-	-	✓	-	-	(✓)	(-)	✓	✓	-	(✓)
Cqrs microservice sampler	(-)	-	-	-	-	-	-	✓	-	-	-	-	✓	✓	✓	✓
Spring boot microservices	-	-	-	✓	-	-	✓	✓	-	-	✓	-	✓	✓	✓	✓
Cas microservice architecture	-	-	-	-	-	✓	-	✓	✓	✓	✓	-	✓	✓	✓	✓
Cloud Strangler example	-	-	-	(✓)	-	-	-	✓	-	-	✓	(-)	✓	(✓)	-	(✓)
Micro company	(-)	-	-	✓	-	-	-	(✓)	-	-	-	(-)	✓	(✓)	(✓)	(✓)
MicroService	-	-	-	(✓)	-	-	-	✓	✓	-	✓	-	✓	(✓)	-	(✓)
Microservice Kubernetes	-	-	-	-	-	✓	✓	✓	✓	(✓)	(✓)	-	(-)	-	✓	✓
TeaStore	(-)	(✓)	✓	(-)	-	(✓)	✓	✓	✓	✓	(✓)	✓	(-)	✓	-	✓
Warehouse microservice	-	-	(✓)	-	-	-	-	✓	-	(-)	(✓)	-	✓	(✓)	-	(✓)
Apollo	-	(✓)	✓	(✓)	-	-	(✓)	-	✓	✓	-	✓	(-)	✓	(-)	(-)
Delivery system	-	-	-	-	-	-	-	✓	-	-	✓	-	✓	✓	(-)	✓
Precision	**(NA)**	**(NA)**	**(3/7)**	**(3/7)**	**(NA)**	**(3/5)**	**(7/8)**	**(19/20)**	**(8/10)**	**(5/7)**	**(7/11)**	**(4/4)**	**(18/18)**	**(12/18)**	**(5/11)**	**(7/20)**
	(NA)	**(NA)**	**42.86%**	**42.86%**	**(NA)**	**60%**	**87.5%**	**95%**	**80%**	**71.4%**	**63.64%**	**100%**	**100%**	**66.67%**	**45.45%**	**35%**
Recall	**(NA)**	**(NA)**	**(3/3)**	**(3/8)**	**(NA)**	**(3/5)**	**(7/9)**	**(19/19)**	**(8/9)**	**(5/6)**	**(7/8)**	**(4/9)**	**(18/24)**	**(12/12)**	**(5/8)**	**(7/7)**
	(NA)	**(NA)**	**100%**	**37.5%**	**(NA)**	**60%**	**77.78%**	**100%**	**88.89%**	**83.3%**	**87.5%**	**44.44%**	**75%**	**100%**	**62.5%**	**100%**

TABLE III
DETECTION RESULTS OF MARS : ✓ STANDS FOR TRUE POSITIVE, (✓) STANDS FOR FALSE POSITIVE, - STANDS FOR TRUE NEGATIVE, (-) STANDS FOR FALSE NEGATIVE

Observation 2: The majority of microservice-based systems do not contain circular dependencies among their microservices, which is a good practice and one of the principles of this architectural style.

3. Mega Service (MS)

Microservices should be small and independent pieces of software that provide a single business function. A mega service in a microservice-based system can cause maintenance issues, increase testing complexity, and reduce performance. MARS detected mega services with a precision of 42.85% and a recall of 100%.

Observation 3: We observe that only 12.5% of the analysed systems contain mega services (c.f. Figure 5). Avoiding mega services is a good practice and is widely adopted.

4. Nano Service (NS)

The granularity of microservices is subject to interpretation and is tied to business requirements. However, developers should avoid having multiple microservices for one business function, which lead to coupling and deployment issues.

MARS detected nano services with a precision of 42.86% and a recall of 37.5% because nano services are subjective: they often require a domain expert to determine whether a microservice qualifies as a nano service or not.

Observation 4: We observe that 33.33% of systems contain nano services. This antipattern is more spread than the mega service antipattern. The nature of the systems in the dataset (i.e., non commercial/industrial systems) could also explain this observation.

5. Shared Libraries (SL)

The microservice architectural style is called "share nothing" because it discourages sharing libraries and other artefacts among microservices. We did not find any occurrence of this antipattern between microservices in our dataset, neither with MARS nor through our manual analysis. Thus, we could not calculate precision and recall values for this antipattern.

Our (lack of) finding shows that developers are aware of the importance of separating the artefacts used in microservices.

Observation 5: Developers use separate libraries and artefacts for each microservice in their systems, which is a good practice for the independent management and deployment of individual microservices.

6. Hardcoded Endpoints (HE)

Microservices should communicate through lightweight mechanisms without hardcoded endpoints/URLs. Service discovery must be used to avoid hardcoding endpoints so that they communicate without assuming the location of others.

Our tool found 5 occurrences of hardcoded endpoints in the dataset, with a precision of 60% and a recall of 60%. MARS missed some instances because URLs formats vary and can be dynamically built. It is challenging for a tool to detect automatically hardcoded endpoints via static analysis.

Observation 6: Developers generally use service discovery when developing microservice-based systems. Only 20.83% of the analysed systems contained occurrences of this antipattern.

7. Manual Configuration (MC)

Automated configuration is a basic principle of microservice-based systems, i.e., dynamically load configurations to speed up the development process. We found that MARS effectively detects the occurrence of this antipattern with a precision of 87.5% and a recall of 77.78%. We observed that less than a half of the systems (9/20) use automated configurations.

Observation 7: Although frameworks exist to simplify configuration management, 55% of the systems rely on manual configuration of their microservices.

8. No CI/CD (NCI)

Continuous delivery and continuous integration are fundamental for microservice-based systems for their rapid deployment and shortened development cycles. We found that MARS accurately detects occurrences of this antipattern. Based on the detection results, we observed that only 20% of the analysed systems use CI/CD in their development pipelines.

Observation 8: Although CI/CD is important in microservice architectures, it is not widely used by developers when implementing microservice-based systems.

9. No API Gateway (NAG)

API gateways allow multiple clients to connect a system through a single entry point, which increases security and scalability and simplifies authentication and authorisation. We found that MARS effectively detects the occurrences of this antipattern with a precision of 80% and a recall of 88.89%. We observed that only 9 out of 24 microservice-based systems use API gateways.

Observation 9: Despite the advantages of API gateways, some developers do not use them in their microservice-based systems, possibly due to the complexity added by their use. The nature of the systems in the dataset (non commercial/industrial systems) could also explain this observation.

10. Timeouts (TO)

Treating every request as failed where no response was received within the timeout may increase the system's latency and decrease the performance. Specifying a timeout for microservices requests instead of using circuit breakers is not recommended for microservice-based systems.

We found that MARS wrongly detects two occurrences of this antipattern because the systems use implementations currently not supported in our tool. Thus, we have 71.4% of precision and 83.3% of recall. We observed that this antipattern is present in 25% of the systems. Developers tend to use instead circuit breakers to handle the possible failure of the invoked microservices.

Observation 10: Developers rarely specify timeout values when invoking microservices, which is a good practice for fault tolerance and resilience. They highly use circuit breakers to protect the system against transient failures.

11. Multiple Service Instances per Host (MSIH)

Microservices should be deployed on their own hosts to ease scalability and avoid technological conflicts inside a host. Detection results were satisfactory although MARS wrongly detects some occurrences. We observed that 11 systems in our dataset (45.83%) are deployed on a unique host.

Observation 11: Almost half of the analysed systems use a unique host to deploy their microservices, which should be avoided to improve scalability and to enable running multiple versions of a same microservice.

12. Shared Persistence (SP)

Each microservice should be responsible for its own data and not share databases to store/load data. We obtained a precision of 100% for this antipattern. However, the recall was 44% because some systems use databases and–or ORM APIs not yet included in MARS. We observed that this antipattern appears in 37.5% of the systems.

Observation 12: Developers are aware of the importance of avoiding sharing access to databases in a microservice-based system.

13. No API Versioning (NAV)

API versioning allows the deployment of different versions of microservices to maintain compatibility with existing consumers. Yet, implementing API versioning is relatively hard and adds complexity. We obtained a precision of 100% and a recall of 75%. We observed that none of the 24 microservice-based systems version their APIs.

Observation 13: Developers tend to ignore APIs versioning. This could possibly due to its complexity and–or the nature of the systems that do not require such a feature.

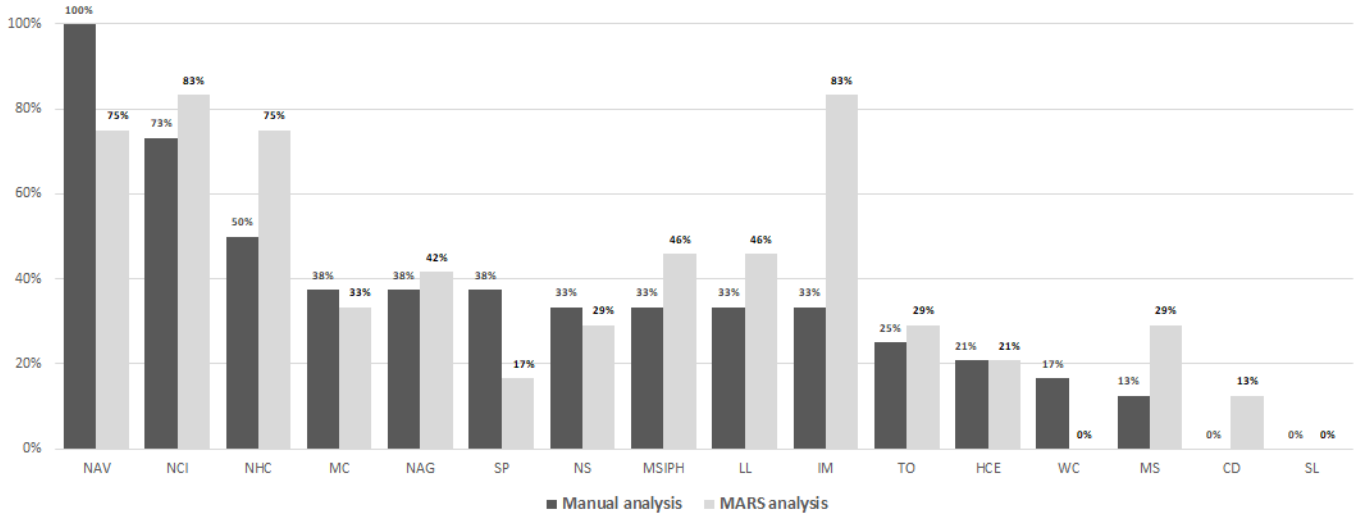

Fig. 5. Comparison between MARS results and manual analysis results

14. No Health Check (NHC)

Verifying microservices health and availability is essential to insure fault tolerance in a microservice-based system. We found that MARS effectively detects the occurrence of such antipattern with a precision of 66.67% and a recall of 100%. Based on our analysis, we report that half of the systems do not perform periodic health checks of their microservices.

> **Observation 14:** The No Health Check antipattern is common in microservice-based systems, which can lead to failures and unavailability.

15. Local Logging (LL)

Distributed logging eases log management and aggregation, and thus, help rapidly identify issues in systems. We did not get satisfactory results with a precision of 45.45% and a recall of 62.5%. Based on our manual analysis, we observed that this practice is not spread across systems: only 8 out of the 24 systems use distributed logging.

> **Observation 15:** 66.66% of the analysed microservice-based systems use local logging mechanisms, which should be avoided because it prevents tracing failures among microservices.

16. Insufficient Monitoring (IM)

Monitoring is important for microservice architectures to trace issues as soon as possible. MARS generated many false positives but did not miss any occurrence of this antipattern, with a precision of 35% and a recall of 100%. We saw that developers are aware of the importance of microservice monitoring: 70.83% of the systems use monitoring tools.

> **Observation 16:** Monitoring microservices is a well established practice, 17 out of 24 systems include monitoring tools.

VI. Discussions

A. Other Observations

Regarding Hardcoded Endpoints and Timeouts, some microservice-based systems use particular communication protocols that intrinsically lead to the occurrence of these antipatterns. In particular, the use of the MQTT communication protocol or CQRS architectural pattern, typically using RabbitMQ as in CQRS Microservice Sampler[7], implies that endpoints must be hardcoded towards the MQTT broker and that timeouts are handled by the broker. Hence, MARS detects occurrences of these two antipatterns although they are not really due to the microservices themselves but the chosen communication protocols.

Regarding No Health Check (NHC), it is possible that some microservice-based systems may use some frameworks that, by default, enable this check. In such a case, MARS may *have* detected an occurrence of this antipattern although, by default into the framework, health check is enabled. Future work includes improving MARS to consider the idiosyncrasies and default behaviour of different frameworks.

In some microservice-based systems, we observed that an antipattern may be both present *and* not present, which we call a "Schrödinger occurrence". A Schrödinger occurrence occurs, in particular, when repositories include both local and multi-host deployment or monolithic and service-based version of a system, e.g., the repositories of LakeSide Mutual[8] (deployment) and of Micro Company[9] (monolithic and microservice-

[7]https://github.com/benwilcock/cqrs-microservice-sampler
[8]https://github.com/Microservice-API-Patterns/LakesideMutual
[9]https://github.com/idugalic/micro-company

based versions). For repositories containing both monolithic and microservice-based versions, we excluded from these repositories the folders containing the monolithic versions because they are irrelevant for MARS.

Finally, we observed some implementations of microservice-based systems that were "erroneous". For example, LakeSide Mutual [10] contains a discovery service but some of its service uses hardcoded endpoints. In such a case, MARS reports occurrences of the Hardcoded Endpoint antipattern while they are due to developers' overlooking some configurations rather than to a poor design.

Based on the analysis of the systems in our dataset, we could draw a first overview of the recommended practices that developers tend to ignore in microservice-based systems. Indeed, we found that developers tend to ignore API versioning, CI/CD, and health check. On the other hand, we found that developers avoid circular dependencies among microservices. They do not share libraries among microservices and properly decompose their microservices. Although preliminary, these observations provide interesting research directions to understand why developers favour certain practices over others. They also warn practitioners of the practices that typically tend to be foregone. Further analyses on a larger dataset should be performed to confirm/infirm our observations.

B. Threats to Validity

We now discuss threats to validity of our results.

a) Construct Validity: The detection rules that we used to detect antipatterns are our interpretation of the antipatterns. Although they are based on the community and our experience with the development and analyses of microservices and while we tried to minimise any bias, we accept that other, better rules may exist. We mitigated this bias by carefully considering previous work, describing the rules in this paper, and providing an open-source implementations for all to study[6].

Although we extensively reviewed the literature to identify the most common antipatterns in microservice-based systems, other antipatterns may exist, which we did not include in our study. MARS should allow specifying other antipatterns, which we consider as future work. Others can also use MARS to specify the same and–or other antipatterns. We provide MARS, its implementation and our rules, as open-source[6].

b) Internal validity: All four authors identified manually and independently potential instances of the microservices in the 24 systems before reconciling them to obtain a ground truth for the validation. We thus tried to remove any bias towards our rules. Also, we release the ground truth online[6] so that other researchers can vet and use it.

c) External Validity: Microservice-based systems are volatile. They can be built using multiple technologies, deployed on multiple hosts, and changed easily. Although we tried to identify and consider the most common technologies for microservices in MARS, we considered only Java microservices. We also may have omitted some other technologies. We minimised this threat by building and providing a tool that can be extended with more parsers.

We relied on lists of dependencies and frameworks to identify some antipatterns. We chose these lists by considering the most widely-used technologies to develop microservices. We do not pretend to have exhaustive lists. To mitigate this threat, MARS can include and use other, additional lists to cover more tools and frameworks.

Even though configuration files are generally written in JSON, XML, or YAML file formats, they can also be written in programming languages, such as Java, which may lead MARS to not consider a file as a configuration file and exclude it. We reduced this threat by relying on file extensions and also on the file names and its content to do the classification.

VII. Conclusion

Microservices are becoming a major architectural style in industry. Several major software-intensive companies, like Netflix and Amazon, adopted this architectural style with great successes. A microservice is a "small" service, providing a single business function, running independently of others, and communicating through lightweight mechanisms. It is also managed by a single team.

The use of microservices is growing every day thanks to their dynamic and distributed nature, which (1) offers greater agility and operational efficiency and (2) reduces the complexity of scalability and deployment.

However, a lack of understanding of the core concepts and principles of this architectural style leads to the introduction of "poor" solutions to recurring design and implementation problems of microservices and microservice-based systems, which are antipatterns that impact the quality of the system. Contrary to object-oriented systems, the dynamic and distributed nature of microservices makes the detection of antipatterns difficult. Consequently, we proposed MARS, a tool-based approach to specify and detect microservice antipatterns. We proposed a meta-model to reify information about microservices, their antipatterns, and their detection rules. We specified 16 antipatterns whose occurrences we detected on and validate with 24 microservice-based systems. A manual validation of the detected occurrences showed that MARS allowed us to specify and detect microservice antipatterns with an average precision of 68.5% and a recall of 78.22%. Our work is useful to both practitioners and researchers. It provides a complete approach for specifying and detecting microservice antipatterns.

Future work includes specifying more antipatterns and analysing all their prevalence in existing microservice-based systems. Thus, we could recommend to developers and researchers good and bad practices to consider when developing microservice-based systems. We also want to empirically and quantitatively study the presence of microservice antipatterns in a larger dataset and study their impact on maintenance.

[10]https://github.com/Microservice-API-Patterns/LakesideMutual

REFERENCES

[1] "Microservices: a definition of this new architectural term," 2019, URL: https://martinfowler.com/articles/microservices.html [retrieved: August, 2020].

[2] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Inc., Feb. 2015, ISBN: 978-1491950357.

[3] D. Taibi and V. Lenarduzzi, "On the Definition of Microservice Bad Smells," *IEEE Software*, vol. 35, pp. 56–62, May 2018.

[4] F. Palma, "Detection of SOA Antipatterns," in *Service-Oriented Computing - ICSOC 2012 Workshops*. Springer Berlin Heidelberg, Jan. 2013, pp. 412–418.

[5] C. Pahl and P. Jamshidi, "Microservices: A Systematic Mapping Study," in *Proceedings of the 6th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and and Technology Publications, Apr. 2016, pp. 137–146.

[6] O. Zimmermann, "Microservices tenets: : Agile approach to service development and deployment," *Computer Science - Research and Development*, vol. 21, pp. 301–310, Nov. 2016.

[7] J. Soldani, D. A. Tamburri, and W.-J. V. D. Heuvel, "The pains and gains of microservices: A Systematic grey literature review," *Journal of Systems and Software*, vol. 146, pp. 215–232, Dec. 2018.

[8] G. Marquez and H. Astudillo, "Actual Use of Architectural Patterns in Microservices-Based Open Source Projects," in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, Dec. 2018, pp. 31–40.

[9] F. Osses, G. Marquez, and H. Astudillo, "Exploration of academic and industrial evidence about architectural tactics and patterns in microservices," in *Proceedings of the 40th International Conference on Software Engineering Companion Proceeedings - ICSE*. ACM Press, May 2018, pp. 256–257.

[10] D. Taibi, V. Lenarduzzi, and C. Pahl, *Microservices Anti-patterns: A Taxonomy*. Springer International Publishing, Jan. 2020, chapter 5, pp. 111–128, in Microservices: Science and Engineering, ISBN: 978-3-030-31646-4.

[11] R. Borges and T. Khan, "Algorithm for detecting antipatterns in microservices projects," in *Joint Proceedings of the Inforte Summer School on Software Maintenance and Evolution*. CEUR-WS, Sep. 2019, pp. 21–29.

[12] B. Kitchenham, "Procedures for performing systematic reviews," *Keele, UK, Keele University*, vol. 33, pp. 1–26, Jul. 2004.

[13] M. I. Rahman, S. Panichella, and D. Taibi, "A curated Dataset of Microservices-Based Systems," in *Joint Proceedings of the Inforte Summer School on Software Maintenance and Evolution*. CEUR-WS, Sep. 2019, pp. 1–9.

[14] ——, "A curated dataset of microservices-based systems," 2019.

[15] D. Cicchetti and S. Sparrow, "Developing criteria for establishing interrater reliability of specific items: Applications to assessment of adaptive behavior," *American journal of mental deficiency*, vol. 86, pp. 127–37, 10 1981.

[16] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, vol. 33, no. 1, pp. 159–174, 1977. [Online]. Available: http://www.jstor.org/stable/2529310

[17] S. Kitajima and N. Matsuoka, "Inferring calling relationship based on external observation for microservice architecture," in *Service-Oriented Computing*, M. Maximilien, A. Vallecillo, J. Wang, and M. Oriol, Eds. Cham: Springer International Publishing, 2017, pp. 229–237.

[18] D. Taibi and V. Lenarduzzi, "On the definition of microservice bad smells," *IEEE Software*, vol. 35, no. 3, pp. 56–62, May 2018.