The Journal of Systems & Software 204 (2023) 111755

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

On the maintenance support for microservice-based systems through the specification and the detection of microservice antipatterns☆

Rafik Tighilt [a], Manel Abdellatif [b,*], Imen Trabelsi [b], Loïc Madern [c], Naouel Moha [b], Yann-Gaël Guéhéneuc [d]

[a] University of Quebec in Montreal, Montreal, Quebec, Canada
[b] École de Technologie Supérieure, Montreal, Quebec, Canada
[c] Polytech Nice Sophia, Biot, Provence-Alpes-Côte d'Azur, France
[d] Concordia University, Montreal, Quebec, Canada

ARTICLE INFO

Article history:
Received 15 August 2022
Received in revised form 18 May 2023
Accepted 22 May 2023
Available online 29 May 2023

Dataset link: https://github.com/LoicMadern/MARS

Keywords:
Microservices
Antipatterns
Detection
Maintenance

ABSTRACT

The software industry is currently moving from monolithic to microservice architectures, which are made up of independent, reusable, and fine-grained services. A lack of understanding of the core concepts of microservice architectures can lead to poorly designed systems that include microservice antipatterns. These microservice antipatterns may affect the quality of services and hinder the maintenance and evolution of software systems. The specification and detection of microservice antipatterns could help in evaluating and assessing the design quality of systems. Several research works have studied patterns and antipatterns in microservice-based systems, but the automatic detection of these 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. The results show that MARS can effectively detect microservice antipatterns with an average precision of 82% and a recall of 89%. Thus, our approach can help developers assert and improve the quality of their microservices and development practices.

1. Introduction

Microservices have already become the prevailing architectural style used in the software industry, and have been adopted by several major actors, such as Netflix and Amazon. A microservice is defined as a service with a single responsibility or business function, running in its own process, communicating through lightweight mechanisms, and managed by a single team (Anon, 2020). For communication, microservices commonly employ Representational State Transfer (REST) Application Programming Interfaces (APIs) and message brokers. The popularity of this architectural style continues to grow thanks to the dynamic and distributed nature of microservices, which (1) offers greater agility and operational efficiency and (2) reduces the complexity of deploying and scaling systems wrt. monolithic applications (Newman, 2015).

However, the lack of understanding of the core concepts of microservice architectural style and consensual definitions of its founding principles may lead to the introduction of "poor" solutions to recurring problems in the design and implementation of microservices, called antipatterns (Taibi and Lenarduzzi, 2018). These antipatterns may impact the quality of the microservices and the systems using them (Palma, 2013). Indeed, Pulnil and Senivongse (2022) showed that the presence of microservice antipatterns negatively impacts the quality of microservice-based systems and that refactoring such antipatterns improves many software quality attributes, such as understandability, modularity, and fault tolerance. An example of a microservice antipattern is the presence of cyclic dependencies between microservices, which can lead to maintenance issues, because a failure in one of the cyclically dependent microservices will lead to a failure in the other microservices involved in this cyclic dependency (Taibi et al., 2020). In addition, the presence of cyclic dependencies between microservices may hinder their scaling and independent deployment (Taibi et al., 2020). As another example, shared persistence (i.e., sharing the same database between microservices) increases coupling between microservices through the same data, which consequently reduces the independence of microservices and impedes their deployment (Taibi et al., 2020). Despite the

☆ Editor: Xin Peng.
* Corresponding author.
E-mail address: manel.abdellatif@etsmtl.ca (M. Abdellatif).

https://doi.org/10.1016/j.jss.2023.111755
0164-1212/

importance and extensive usage of microservices, to our knowledge, no automated approach for the detection of microservice antipatterns has been proposed so far.

We propose MARS, a tool-based approach to specify and detect microservice antipatterns. We rely on a metamodel that includes the data needed to specify and apply detection rules on the source code of microservice-based systems. Using MARS, we specify 16 antipatterns and detect their occurrences within 24 microservice-based systems. We perform a manual validation of the detected occurrences to compute precision and recall. Our results show that MARS allows us to specify and detect microservice antipatterns with an average precision of 82% and a recall of 89%. Thus, we propose a highly automated approach and a large-scale study to specify and detect microservice antipatterns, which should pave the way for future practical and research applications, with the ultimate aim of improving the design and implementation of microservices.

This paper is organised as follows: Section 2 describes previous work and presents a catalogue of microservice antipatterns. Section 3 outlines our approach for antipatterns detection and the detection rules. Section 4 details our study, Section 5 describes our results and Section 6 discusses them. Section 7 describes the threats to the validity of our work. Finally, Section 8 concludes with future work.

2. Background

2.1. Catalogue

We now present a catalogue of several microservice antipatterns constructed through a systematic literature review (Kitchenham, 2004). The detailed process of the creation of this catalogue is described in detail in our prior work (Tighilt et al., 2020). Essentially, we identified 1195 papers through a query in major scientific databases. We then applied several inclusion and exclusion criteria (e.g., we excluded papers not written in English and papers not related to microservice antipatterns) to obtain a total of 27 papers that pertained to microservice antipatterns. We also manually analysed 67 open-source systems (Rahman et al., 2019) to assess the concrete presence of the 16 antipatterns identified in the systematic literature review. This analysis 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 among microservices and impedes the delivery of new business functions.
(2) **Cyclic Dependencies (CD).** This antipattern occurs when multiple microservices are circularly co-dependent and thus no longer independent, which goes against the very definition of microservices.
(3) **Mega Service (MS).** This antipattern appears when a microservice provides multiple business functions. A microservice should be manageable by a single team and should pertain 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 to work together.
(5) **Shared Libraries (SL).** This antipattern relates to the sharing of libraries and files (e.g., binaries) by multiple microservices, which breaks their independence as they rely on a single source to fulfil their business function.
(6) **Hard-Coded Endpoints (HE).** This antipattern relates to URLs, IP addresses, ports, and other endpoints being hard-coded in the source code of microservices and/or configuration files, which interferes with load balancing and deployment.
(7) **Manual Configuration (MC).** This refers to configurations that must be manually pushed in some microservices. Since microservice-based systems evolve rapidly, 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, managing 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 unnecessary disconnections or delays.
(11) **Multiple Service Instances Per Host (MSIPH).** This antipattern happens when multiple microservices are deployed on a single host (e.g., container, physical machine, virtual machine), 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 their business function.
(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 describes microservices that are not periodically health-checked. Unavailable microservices may not be noticed, leading to timeouts and other errors.
(15) **Local Logging (LL).** This antipattern results from microservices having their own logging mechanism, which prevents aggregation and analyses of their logs and the monitoring and recovery of systems.
(16) **Insufficient Monitoring (IM).** This antipattern describes neglecting to record data on performance levels and failures of microservice-based systems that would be useful for maintenance purposes.

2.2. Related work

The work presented in this paper relates to maintenance support for microservice-based systems through the specification and the detection of microservice antipatterns. Therefore, we provide in this section an overview of existing approaches on (1) the detection of antipatterns in software engineering, (2) the specification of microservice (anti)patterns, and (3) the detection of microservice antipatterns.

2.2.0.1. Antipatterns detection in software engineering. Antipatterns in multiple fields and programming paradigms have been studied in the literature. DECOR (Moha et al., 2009), for example, allows the automatic detection of object-oriented code smells and antipatterns in object-oriented code sources, like Java. Paprika (Hecht et al., 2015) and aDoctor (Palomba et al., 2017) allow the detection of antipatterns in Android mobile

applications. In the following, we focus on the literature related to microservice antipatterns. Several research works exist on microservice antipatterns, of which only a few works propose approaches for their detection.

2.2.0.2. Specification of microservice patterns and antipatterns. Pahl and Jamshidi (2016) 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 the design of microservice-based systems and concluded that research on microservice architecture is limited. We concur with their conclusion and confirm that there is a lack of specification and detection of microservice antipatterns in the literature.

Zimmermann (2016) studied the literature and identified seven microservice principles. He compiled some practitioners' questions and derived several research topics related to the differences between Service-Oriented Architecture (SOA) and the microservice architectural style. He concluded that microservices are not entirely new but qualify as a special implementation of the SOA paradigm. Still, we argue that the microservice architectural style is subject to particular antipatterns and requires dedicated detection approaches.

Marquez and Astudillo (2018) extended their previous work Osses et al. (2018) to propose a catalogue of microservice architectural patterns. They provided a list of technologies that can be used to build microservice-based systems with these patterns. They also studied the distribution of these patterns in 30 open-source projects relying on a manual analysis of their source code to assess the state of usage of microservice patterns. They found that developers use only a few architectural patterns broadly and that most of the analysed systems rely on SOA and not microservice-specific patterns. What differentiates our work from this study is that we focus on the specification and identification of microservice antipatterns by providing a tool, MARS, that automatically detects their presence in microservice-based systems.

Taibi et al. (2020) introduced a catalogue and a taxonomy of microservice antipatterns based on a literature review and evidence from 27 practitioners on bad practices experienced while developing microservice-based systems. They identified 20 organisational and technical antipatterns. They studied and reported the harmfulness level of each antipattern. They concluded that splitting a monolithic system into microservices is a critical and challenging problem. They also concluded that antipatterns specific to microservices can hinder the maintenance and evolution of microservice-based systems. Finally, unlike our work, this study does not provide any automated approach to detect occurrences of microservice antipatterns in existing systems. They only provide a taxonomy of microservice antipatterns and discuss their harmfulness according to practitioners' experiences.

2.2.0.3. Detection of microservice antipatterns. Microservice antipatterns have been discussed in the literature but few works exist on their automatic detection. To the best of our knowledge, only Borges and Khan (2019), Walker et al. (2020), and Pigazzini et al. (2020) have proposed algorithms for automatic detection of antipatterns in microservice-based systems.

Pigazzini et al. (2020) extended the existing tool Arcan, developed for architectural smells detection, to explore microservice architectural antipatterns. They manually validated their tool using five open-source microservice-based systems by computing the accuracy of the detection results. They detected three antipatterns: cyclic dependencies, shared persistence, and hard-coded endpoints (Pigazzini et al., 2020). For instance, they detected circular dependencies between microservices by performing a depth-first search in microservices call graph. In contrast, MARS only detects direct circular dependencies between pairs of microservices. Although in our work we rely on similar detection rules for detecting shared persistence and hard-coded endpoints, we cover additional antipatterns and validate our tool on a larger number microservice-based systems.

Borges and Khan (2019) proposed an algorithm for detecting five microservice antipatterns using static analysis. We cover in common only two antipatterns: API versioning and hard-coded endpoints. While they applied their algorithm on one open-source microservice-based system and discussed some improvements, we built our approach for 16 microservice antipatterns and studied their prevalence in 24 microservice-based systems.

Walker et al. (2020) proposed *MSANose* to detect antipatterns in microservice-based systems. Their approach detects 11 antipatterns, using a software architecture recovery approach (Rademacher et al., 2020; Alshuqayran et al., 2018; Granchelli et al., 2017). This approach first analyses microservices individually, then groups them to build a graph on which it performs the detection. The authors validated their approach on two microservice-based systems. While they share eight microservice antipatterns in common with our approach, our detection methods differ for some of them. For example, although they defined the no API gateway antipattern in their work, they reported that their tool only generates a warning message recommending the usage of an API gateway when the number of microservices exceeds 50, without explicitly detecting the antipattern itself. In contrast, our approach detects the presence of this antipattern regardless of the number of microservices in the system being analysed.

Furthermore, similar to Pigazzini et al. (2020), they detected circular dependencies between microservices by applying a depth-first search on the call graph of the system being analysed. However, we detect cyclic dependencies between pairs of microservices as mentioned above. They relied on microservice bytecode and analysed the parameters of the methods used to communicate with other microservices. In contrast, MARS analyses the source code, deployment files, configuration files, and environment files of microservice-based systems to identify hard-coded IP addresses, port numbers, and/or URLs, providing a more holistic approach. In the case of the shared libraries antipattern, they only compared the names of libraries used by different microservices and did not exclude local libraries, potentially leading to less precise detection. In contrast, MARS detects shared libraries between microservices and excludes local libraries, allowing for more accurate detection. We compare the results of *MSANose* with MARS systematically, and we provide detailed results in Section 5.2.

Our paper makes significant novel contributions to the literature on microservice antipatterns detection (Borges and Khan, 2019; Walker et al., 2020; Pigazzini et al., 2020). First, we introduce MARS, a highly automated tool that uses a novel metamodel to detect 16 microservice antipatterns. We assembled this list of antipatterns in our prior work (Tighilt et al., 2020) based on two methods: (1) a comprehensive and diverse literature review, and (2) a manual analysis of 64 microservice-based systems. Additionally, the MARS metamodel is generalisable, language- and technology-agnostic, and applicable to any type of microservice-based system. Second, we conducted the largest empirical analysis to date on microservice antipatterns by automatically and accurately detecting 16 microservice antipatterns with an average precision of 82% and a recall of 89%. We validated the detection results of MARS on a dataset of 24 microservice-based systems, the largest validation dataset when compared to existing works (Borges and Khan, 2019; Walker et al., 2020; Pigazzini et al., 2020). Finally, our validation is

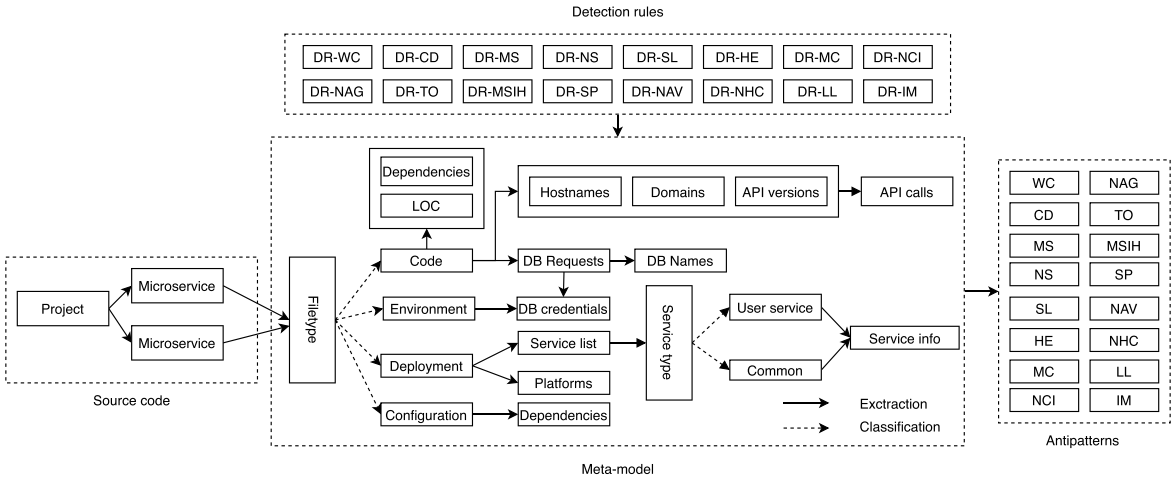

Fig. 1. Microservice Antipatterns Research Software (MARS).

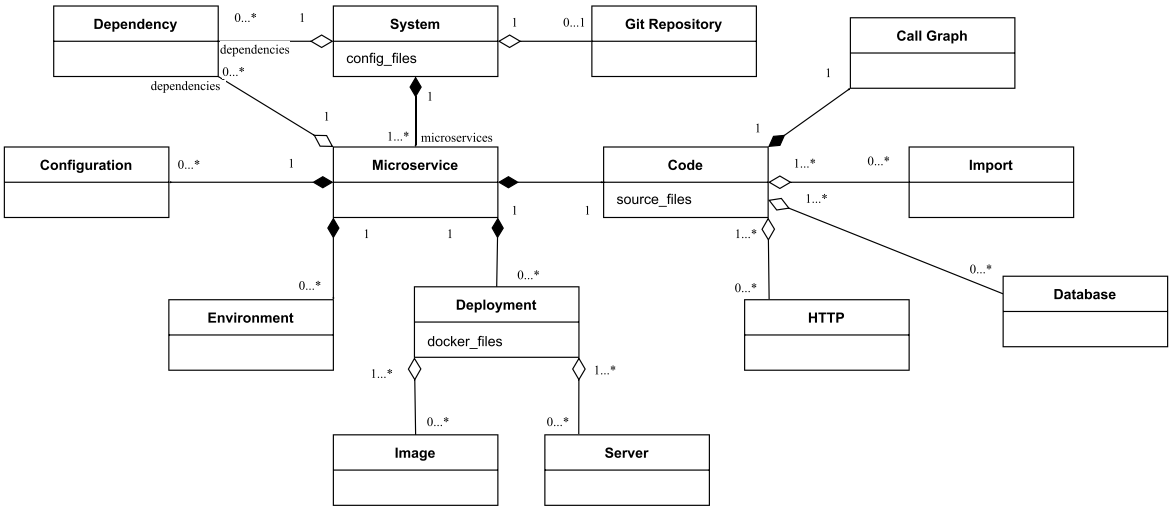

Fig. 2. MARS metamodel constituents.

reproducible and publicly available to enable new research to build upon our work. We are sharing our detection tool as well as our ground truth, which includes 172 instances of microservice antipatterns manually detected by two of the authors, as we will detail in Section 4.1.

3. Approach

We now present MARS, a software tool that uses a fully automatic approach to detect the antipatterns described in Section 2. Fig. 1 summarises our method of detecting microservice antipatterns. It shows that MARS takes as input a microservice-based system or a list of microservices (either as Git repositories or local folders). Then, it extracts from each microservice the data necessary to detect antipatterns, which is reified using a dedicated metamodel. Finally, it applies detection algorithms on a model of the system that conforms to the metamodel to detect occurrences of each specified antipattern.

3.1. Metamodel definition

We created a metamodel to describe the data needed to apply our detection algorithms, which includes general information about the system, its Git repository, its individual microservices, and their dependencies, as well as the source code, environment files, configuration files, deployment files, docker images, databases, HTTP requests, and imports.

This metamodel allows our detection algorithms to have access to relevant data while being independent of its sources. It also allows for avoiding parsing the source code of the systems and eases the evolution of MARS by introducing new constituents in the metamodel 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 or another build tool.

Fig. 2 illustrates the metamodel constituents and their relationships. Each constituent of the metamodel is necessary for the detection of one or more antipatterns. For example, the `Configuration` constituent is used to detect hard-coded endpoints by searching URLs inside configuration files, along with the `Code` and `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 analysing a microservice-based system on the local file system.

A `System` knows about two sets of constituents: (1) `Microservice` and (2) `Dependency`. The `Microservice` constituent represents an actual microservice in the microservice-based system. It contains data about this microservice, such as the

number of files and LOCs (which might be used to detect mega services and nano services).

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

The `Configuration` constituent stores data gathered from the various configuration files of a microservice. It allows searching 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 commonly 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 developers. It points to an `Image` (e.g., for Docker) and/or `Server` (e.g., for Amazon ECS), containing data about these particular deployment targets. They allow microservices to be identified by extracting the `Images` from which they are derived.

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

(1) Source files: a dictionary of source-file names and paths used to filter test files, for example, which are not relevant for detection.
(2) Imports: a list of all the import statements in source files.
(3) HTTP: a list of all HTTP requests in the source code.
(4) Database: a list of database queries and "create" statements as well as data source paths.
(5) Call-graph: a call graph of the source code generated with a static code analysis tool, *Understand*.[1]

3.2. Detection rules

For each antipattern, we defined a set of detection rules to detect their occurrences in a given microservice-based system. Below, we provide textual and pseudo-code descriptions of these rules and list the metamodel constituents used by each rule. For simplicity, not all of these constituents are visible in the provided pseudo-code.

(1) **Wrong Cuts (WC).** Microservices have one file type in the source code. An example would be a microservice containing only presentation code connecting to a microservice containing only business logic code. We rely on file extensions, contents, and languages to identify this antipattern.
Required constituents: *Microservice, Code.*

```
frontend_languages: a list that contains frontend
extension languages
threshold_frontend_files: a threshold for the
allowed percentage of frontend_languages
in a microservice (80% in our study)
def WrongCut(Microservice MicroS):
  exist = false
  cpt = 0
  for extension in frontend_languages :
      for file in MicroS.Code.source_files:
          if extension in file:
              cpt += 1
  if cpt > threshold_files *
    MicroS.Code.source_files.size:
      exist = true
  return exist
```

(2) **Cyclic Dependencies (CD).** We use the call graph of the microservice-based system, which we analyse to detect circular dependencies among microservices.
Required constituents: *Microservice, Code, Dependency, Import, Call Graph.*

```
isConnectedTo(): a function that verifies the
existence of a direct dependency between
two microservices.

def CyclicDependencies(Microservice MicroSA,
                       Microservice MicroSB):
  return     isConnectedTo(MicroSA, MicroSB)
         AND isConnectedTo(MicroSB, MicroSA)
```

(3) **Mega Service (MS).** A mega service supports multiple business functionalities and thus is potentially large compared to microservices that do not have this antipattern. MARS detects the presence of mega services by counting the lines of code and the number of files within a microservice. These numbers should be greater than certain thresholds specified by an expert.
Required constituents: *Microservice, Code.*

```
def MegaService(Microservice microS):
  exist = false
  if LOCs(microS.Code) > threshold_LOCs
      exist = true
  return exist
```

(4) **Nano Service (NS).** A nano service is a fine-grained microservice that provides only a part of a business functiona in a microservice-based system. This antipattern generally results from a too fine-grained decomposition of the system. Nano services are small by definition, and MARS detects their presence by analysing the number of lines of code and the number of files within a microservice. These should not exceed specific thresholds that must be specified by an expert.
Required constituents: *Microservice, Code.*

```
def NanoService(Microservice MicroS):
  exist = false
  if LOCs(MicroS.Code) < threshold_LOCs AND
      NumFiles(MicroS.Code) < threshold_files:
      exist = true
  return exist
```

(5) **Shared Libraries (SL).** Some source files, libraries, or other artefacts of one microservice are used by other microservices.
Required constituents: *Microservice, Code, Dependency, Import.*

```
def SharedLibs(Microservice[] microservices):
  shared_libs = []
  libs = []
  for each ms in microservices:
    for each dep in ms.dependencies:
      if libs.contain(dep) AND
         NOT shared_libs.contain(dep):
        shared_libs.add(dep)
      else:
        libs.add(dep)
  return shared_libs.length > 1
```

(6) **Hard-Coded Endpoints (HE).** REST API calls inside some source code, deployment files, configuration files, or environment files contain hard-coded IP addresses, port numbers, and/or URLs. Hard-coded endpoints may also be present when no discovery service is used.
Required constituents: *Microservice, Code, HTTP, Configuration, Environment, Deployment, Dependency.*

[1] https://www.scitools.com

```
service_discovery_tools: a list that contains
names of existing service discovery tools

1 def Hard–codedEndpoints(System aSystem):
2   if !intersect(aSystem.dependencies,
                  service_discovery_tools):
4     potential_hard–coded = true
5   if potential_hard–coded:
      for each ms in aSystem.microservices:
6       if has_urls(ms.Configuration)
7          OR has_urls(ms.Code)
8          OR has_urls(ms.Environment)
9          OR has_urls(ms.Deployment):
10        list_urls.append(ms.Code.HTTP)
11  return list_urls
```

(7) **Manual Configuration (MC).** Microservices have their own configuration files. No microservice is responsible for configuration management. No configuration management tools are present in the dependencies of the system. The detection algorithm for this antipattern works as follows:
Required constituents: *Microservice, Code, Configuration, Environment, Dependency.*

```
defined_config_libs: a list that contains names
of existing service configuration libraries

1 def ManualConfiguration(System aSystem):
2   exist = false
3   for each cl in defined_config_libs:
4     if NOT aSystem.dependencies.contain(cl)
         AND length(aSystem.Configuration) > 0:
5       exist = true
6     for each ms in aSystem.microServices:
7           if NOT ms.dependencies.contain(cl)
           AND length(ms.Configuration) > 0:
8         exist = true
9   return exist
```

(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.
Required constituents: *Microservice, Code, Dependency, GitRepository.*

```
defined_ci_libs: a list of names of
existing CI/CD tools
defined_ci_folders: a list of names of
existing CI/CD configuration folders
(e.g., .circleci, .travisci)

1 def NoCICD(System aSystem):
2   exist = true
3   for each ms in aSystem.microservices:
4     if intersect(ms.dependencies,
                   defined_ci_libs):
5       exist = false
6   if exist:
7     if Regex_match(defined_ci_folders,
                     aSystem.GitRepository:
8       exist = false
9   return exist
```

(9) **No API Gateway (NAG).** Source code does not contain signatures of common API gateway implementations (e.g., Netflix Zuul). No frameworks or tools related to API gateways are present in the dependencies of the microservices.
Required constituents: *Microservice, Dependency.*

```
api_gateway_libs: a list of names of API gateways

1 def NoApiGeteway(System aSystem):
2   exist = true
3   for each agl in api_gateway_libs:
4     if aSystem.dependencies.contain(agl):
5       exist = false
6   for each ms in aSystem.microServices:
7         if ms.dependencies.contain(agl):
8       exist = false
9   return exist
```

(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.
Required constituents: *Microservice, Code, Dependency.*

```
list_circuit_breakers: a list that contains circuit
breakers libraries

1 def Timeouts (Microservice MicroS):
2   return (NOT MicroS.dependencies
                .contain(list_circuit_breakers)
            AND intersect(Fallback, MicroS.Code))
            OR intersect(Timeout, MicroS.Code)
```

(11) **Multiple Service Instances Per Host (MSIPH).** 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.
Required constituents: *Microservice, Deployment, Image, Server, Environment, Configuration.*

```
1 def MultipleServiceInstancePerHost(System aSystem):
2   no_docker_file = 0
3   system_has_docker = false
4   if (conf_file in aSystem.config_files)
        .contain(docker–compose.yml):
5     system_has_docker=true
6   for each ms in aSystem.microservices:
7     if length(ms.Deployment.docker_files)<1:
8       no_docker_file+=1
9   return NOT system_has_docker AND
          no_docker_file >=
          length(aSystem.microservices)
```

(12) **Shared Persistence (SP).** Microservices share data-source URLs. A single database is created by the system and multiple microservices use this database.
Required constituents: *Microservice, Code, Database, Image.*

```
1 def SharedPersistence(Microservice[] microservices):
2   shared_databases = []
3   databases = []
4   for each ms in microservices:
5     for each db in ms.Code.DataBase:
6       if data_bases.contain(db) AND
           NOT shared_data_bases.contain(db):
7         shared_data_bases.add(db)
8       else:
9         data_bases.add(db)
10 return length(shared_data_bases) > 1
```

(13) **No API Versioning (NAV).** Endpoints and URLs do not contain version numbers. No version information is present in the configuration files.
Required constituents: *Microservice, Code, Configuration.*

```
1 def HasNoApiVersioning(System aSystem):
2   no_api_versioning = 0
3   has_api_versioning = false
4   for each ms in aSystem.microservices:
5     if NOT ms.Configuration
         .contain('apiVersion'):
6       no_api_versioning +=1
7   for each file in aSystem.config_files:
      if file.contain('apiVersion'):
8       has_api_versioning = true
9   return NOT has_api_versioning AND
          no_api_versioning
          >= length(aSystem.microservices)
```

(14) **No Health Check (NHC).** No "health check" or "health" endpoint exists. No common implementation of health checks is used (e.g., Springboot Actuator).
Required constituents: *Microservice, Code, Configuration, Image, Dependency.*

```
health_check_tools: a list of health—check libraries

1 def HasNoHealthCheck(System aSystem):
2   no_health_check=true
3   number_ms_without_hc = 0
4   for each dp in aSystem.dependencies:
5     if health_check_tools
        .contain(dp):
6       no_health_check=False
7   for each ms in aSystem.microservices:
8     for each dp in ms.dependencies:
9       if NOT health_check_tools
          .contain(dp):
10        number_ms_without_hc += 1
11 return no_health_check AND
          length(number_ms_without_hc) > 0
```

(15) **Local Logging (LL).** We detect this antipattern by checking if there is (1) no distributed logging in the dependencies and–or (2) no common logging microservice. Each microservice has its own log file paths.
Required constituents: *Microservice, Dependency.*

```
list_logging_libs: list of logging libraries

1 def LocalLogging (System aSystem):
2   exist = true
3   for each ll in list_logging_libs:
4     if aSystem.dependencies.contain(ll):
5       exist = false
6   for each ms in aSystem.microServices:
7     if ms.dependencies.contain(ll):
8       exit = false
9   return exit
```

(16) **Insufficient Monitoring (IM).** We detect this antipattern by looking for a monitoring framework or library in the microservice dependencies (e.g., Prometheus).
Required constituents: *Microservice, Code, Dependency.*

```
list_monitor_libs: list of monitoring libraries

1 def InsufficientMonitoring ( System aSystem ):
2   exist = true
3   for each ml in list_monitor_libs:
4     if aSystem.dependencies.contain(ml):
5       exist = false
6     for each ms in aSystem.microServices:
7       if ms.dependencies.contain(ml):
8         exit = false
9   return exist
```

3.3. Implementation

We implemented MARS using a variety of frameworks and libraries to detect antipatterns in microservice-based systems. We used Python scripts to parse the source code of each microservice and create a model of the system. We relied on several libraries such as glob[2] and javalang[3].

We retrieved each constituent of the metamodel by applying two types of parsers: regex-based regular expressions and the javalang library using an abstract syntax tree. MARS relies on regex-based regular expressions to extract the *HTTP*, *Database*, *Configuration*, and *Environment* constituents. The javalang parser was used to retrieve some *code*-related data, such as lists of methods and imports. We also used the Python library dockerfile-parse[4] to retrieve all the images and docker files used by a system. Finally, we identified dependencies among services using the Bibliothecary library,[5] which parses dependency manifests.

We released MARS as an open-source project, whose source code and other artefacts are available online.[6] MARS was designed to be extensible. We developed MARS on a technology-agnostic metamodel to support multiple programming languages, technologies, and tools. For example, the search for libraries in MARS uses a configuration file in which developers can specify the libraries that they want to consider. To support the analysis of microservice-based systems written in different programming languages (e.g., Go, JavaScript, Perl), we only must add dedicated parsing tools and customise our parsing methods to extract the required data (e.g., methods, dependencies, HTTP requests, database queries) for instantiating MARS metamodel. All the other functionalities in MARS will remain unchanged because the detection of antipatterns relies on analysing models that conform to the MARS metamodel.

3.4. Building a model from source code

To generate models that conform to the MARS metamodel, we develop and use various tools to extract the date needed to instantiate and relate to each constituent of the metamodel. As explained in Section 3.1, the *System* constituent represents the system as a whole and is the root of the model. The microservices defined in the project are extracted from the project root, assuming that each microservice is a folder inside the project root. Each *Microservice* object contains a name, the global number of lines of code (extracted using *cloc*[7]), the main programming language (extracted using *enry*[8]), and sub-constituents (*Configuration*, *Environment*, *Deployment* and *Code*).

We extract configuration data for each microservice by parsing commonly-used configuration files (Spring app.properties, config.xml, *.conf, etc.). The parsing depends on the type of files. Currently, MARS includes a parser for Spring configuration files, even though other types of configuration files are partly supported. The same process of parsing is used to instantiate the *Environment* and *Deployment* constituents.

The source code of a project is parsed to extract various pieces of data: method names, comments, imports, HTTP requests, and database calls. We also search for HTTP URLs and database credentials and statements in configuration files, deployment scripts, and environment files during parsing, using regular expressions.

We describe in the following a running example of the detection of the *timeouts* antipattern in a microservice-based system. MARS takes as input the Git repository of the system to analyse and extract the corresponding files from the project root. We start by manually excluding folders that are not relevant (e.g., monolith version of the system, third-party folders). We then parse the source files and generate the metamodel of the system. We also use a static code analysis tool, i.e., *Understand*, to generate the call graph of the system and instantiate the related constituents and add them to the model. We then apply the detection rule of the *timeouts* antipattern (Section 3.2) to detect its occurrences. This detection rule uses the *Dependency* and *Code* constituents of the metamodel. The *Dependency* constituent is used to search if any circuit breaker tool is used in the system. The *Code* constituent is used to search for keywords and methods, such as "*timeouts*"

2 https://docs.python.org/3/library/glob.html
3 https://pypi.org/project/javalang/
4 https://pypi.org/project/dockerfile-parse/
5 https://github.com/librariesio/bibliothecary
6 https://github.com/LoicMadern/MARS
7 https://github.com/AlDanial/cloc
8 https://github.com/src-d/enry

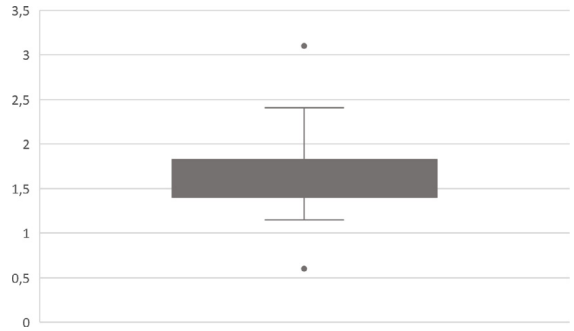

Fig. 3. Boxplot of the number of files per project, on a logarithmic scale.

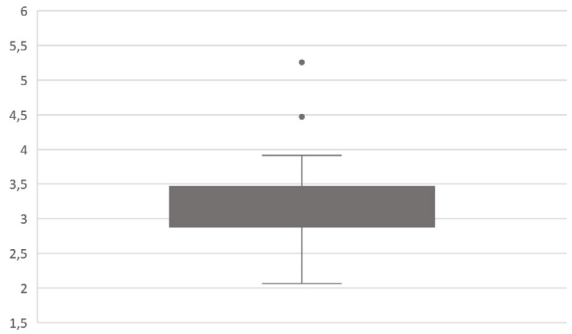

Fig. 4. Boxplot of the number of lines of code per project, on a logarithmic scale.

and "*fallback*" in the source code. We combine the search results for both constituents and check if there is no dependency on any circuit breaker and if a fallback method is used in the system or if a timeout value is specified in the source code or configuration files. Based on the output of the rule, MARS indicates whether the antipattern is present or not in the targeted system.

4. Study design

We now discuss the design of a study to validate our approach. We applied MARS on a set of microservice-based systems and compared the detected antipattern occurrences against ground truth, i.e., the instances of the antipatterns found manually in the systems. We describe in the following our dataset and how we built the ground truth.

4.1. Dataset

We applied MARS on 24 microservice-based systems written in Java. These systems are taken from a dataset of microservice-based systems available online (Rahman et al., 2019). As mentioned in Section 2.1, we manually analysed in our prior work (Tighilt et al., 2020), a dataset of 67 open-source microservice projects—implemented with different programming languages—to build our catalogue of microservice antipatterns and assess how they appear in practice. We relied on this dataset because (1) it is the state-of-the-art dataset of microservice-based systems widely used in the literature (Rahman et al., 2019), (2) it is open-source, and (3) it contains microservice-based systems of different sizes and types (i.e., industrial as well as demo systems). In our work, we considered only microservice-based systems written in Java, we also excluded toy systems (with only one microservice), and thus retained from this dataset 24 Java-based microservices systems that are described in Table 1. Figs. 3 and 4 show boxplots of the numbers of files and lines of codes of the systems, respectively, plotted on a logarithmic scale for clarity.

The source code of any microservice-based system contains developer-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 such as node modules, Maven folders, and composer vendor directories.

4.2. Ground truth

To build a ground truth of instances of the antipatterns, two of the authors independently analysed each microservice-based system in the dataset to find all instances of each antipattern. After independently collecting instances of the antipatterns, the two authors compared their findings. In case of a discrepancy, a third author was responsible for reconciling the two other authors' findings. Thus, three authors were involved in the manual identification and validation of the antipattern instances. The two authors agreed on most instances and the third author only validated five instances, which were all related to the nano and mega services because their detection relies on thresholds that can be difficult to assess manually. To make our ground truth easily accessible to the research community, we have made it available online.[9] The ground truth comprises all instances of microservice antipatterns that we identified through our manual analyses.

5. Study results

This section presents the results and observations after performing our study. For each microservice antipattern, we report the precision and recall scores achieved and provide a concrete example of one of its occurrences within one of the microservice-based systems. We calculated the precision and recall values for each of the detected antipatterns as follows, 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 detection precision of our tool is satisfactory, varying between 60% and 100% with an average of 82%. The recall value is also satisfactory and varies from 64% to 100% with an average of 89%. These precision and recall values confirm the effectiveness of MARS at detecting the selected antipatterns. Detailed results are presented in Table 2 and Fig. 5.

5.1. Detection results of MARS

After applying MARS to the 24 microservice-based systems we analysed, we obtained promising results. Specifically, MARS accurately identified all instances of the shared libraries, multiple service instances per host, and circular dependencies antipatterns, demonstrating its effectiveness in detecting some of the most common antipatterns in microservice architectures. Furthermore, the tool achieved high precision and recall scores in identifying the wrong cuts, manual configurations, no CI/CD, no API gateway, timeouts, and shared persistence antipatterns. While these results are encouraging, we observed that MARS generated a higher number of false positives when detecting the seven remaining

[9] https://github.com/LoicMadern/MARS/blob/main/groundtruth.xlsx

Table 1
Number of microservices, files and LOCs per system.

System	# Microservices	# Files	# LOC	Version Date
Spring Netflix OSS	3	17	443	07-18-2017
FTGO	9	257	8239	06-02-2021
LakeSide Mutual	9	424	89477	03-14-2022
Spring Petclininc	3	25	795	04-15-2022
Freddy's BBQ	6	35	1752	06-04-2017
Spring Cloud Movie	4	33	885	04-11-2017
Piggymetrics	4	88	3176	11-15-2022
Tap And Eat	6	31	576	01-04-2017
E-commerce	3	24	756	06-07-2017
Consul	3	38	1750	09-28-2020
Microservice Demo	3	38	1766	09-17-2020
Qbike	5	77	2057	06-03-2019
Spring Cloud Microservice	9	21	673	03-23-2017
CQRS Microservice Sampler	3	26	1028	10-30-2016
Spring Boot Microservices	2	4	116	10-11-2018
Cloud Strangler Example	3	30	932	03-07-2019
Micro Company	17	244	90315	07-10-2020
MicroService	13	42	1052	02-09-2018
MicroService Kubernetes	3	38	1640	12-04-2020
TeaStore	3	62	5073	04-20-2022
Warehouse Microservice	6	222	4623	03-03-2018
Apollo	9	68	29510	06-21-2022
Delivery System	2	14	537	11-08-2017
Ticket-Train	45	1258	180338	09-02-2021

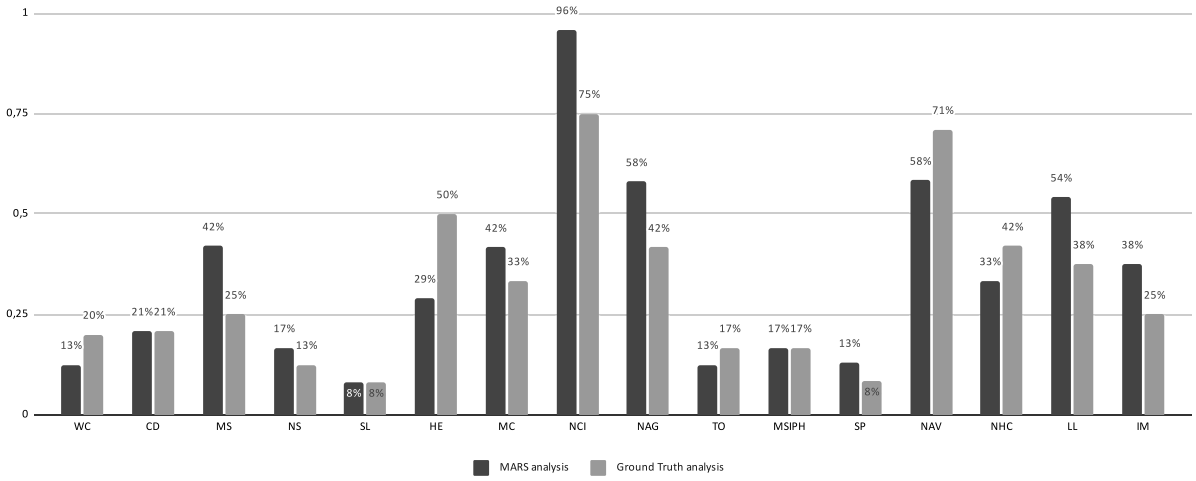

Fig. 5. Comparison between MARS results and ground truth analysis results.

microservice antipatterns in our catalogue. In the following, we present a comprehensive analysis of MARS detection results for each antipattern, along with corresponding examples and the prevalence rates of antipatterns observed in our dataset.

5.1.1. Wrong Cuts (WC)

Our evaluation of MARS showed that it achieved high precision (100%) and recall (71.42%) rates in detecting wrong cuts in microservices. We missed detecting two occurrences of wrong cuts due to the subjectivity in defining this antipattern and our choice of thresholds. As shown in Table 2 and Fig. 5, we found that only a few systems (5/24) in our dataset contained occurrences of wrong cuts. Thus, we can conclude that the majority of the microservice-based systems analysed in our study decompose their microservices in an appropriate manner.

Example. In *LakeSide Mutual*, microservices are organised according to their technical layers (i.e., presentation or business) instead of their business capabilities. The system includes microservices such as *customer-self-service-frontend* and *customer-self-service-backend*. Our detection rule identified frontend microservices by identifying a large number of web interface-related files *(e.g., .js, .vue, .html, .json, and .css)*.

5.1.2. Circular Dependencies (CD)

MARS accurately detected all microservice circular dependencies in our dataset, achieving 100% precision and recall. However, it should be noted that MARS only detects direct circular dependencies between pairs of microservices due to the NP-hard nature of detecting cycles in call graphs. This antipattern was observed in a small fraction of systems (5/25) in our dataset.

Table 2
Detection results of MARS, - stands for no occurrences detected by Mars and reported in the Ground Truth.

System		Antipattern															
	Precision (P) & Recall (R)	WC	CD	MS	NS	SL	HE	MC	NCI	NAG	TO	MSIPH	SP	NAV	NHC	LL	IM
Spring Netflix OSS	P	–	–	–	–	–	–	–	1/1	–	–	–	–	1/1	–	1/1	–
	R	–	–	–	–	–	–	–	1/1	–	–	–	–	1/1	–	1/1	–
FTGO	P	–	1/1	–	–	–	3/3	1/1	–	–	1/1	–	0/1	–	–	–	–
	R	–	1/1	–	–	–	3/3	1/1	–	–	1/1	–	–	–	–	–	–
LakesideMutual	P	3/3	–	–	–	–	–	1/1	1/1	1/1	–	–	–	–	–	0/1	0/1
	R	3/3	–	–	–	–	0/2	1/1	1/1	1/1	–	–	–	–	–	–	–
Spring Petclinic	P	–	–	0/1	–	–	–	–	0/1	–	–	–	–	0/1	–	–	–
	R	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Freddys BBQ	P	–	–	1/1	–	–	3/3	0/1	1/1	1/1	–	1/1	–	1/1	1/1	1/1	–
	R	–	–	1/1	–	–	3/3	–	1/1	1/1	–	1/1	–	1/1	1/1	1/1	–
Spring Cloud Movie	P	–	1/1	–	–	–	–	–	1/1	–	–	1/1	–	1/1	1/1	–	–
	R	–	1/1	–	–	–	0/1	–	1/1	–	–	1/1	–	1/1	–	–	–
Piggymetrics	P	–	–	0/1	–	–	–	–	1/1	–	–	–	–	1/1	–	–	0/1
	R	–	–	–	–	–	0/2	–	1/1	–	–	–	–	1/1	–	–	–
Tap And Eat	P	–	–	–	–	–	–	–	1/1	1/1	–	–	–	0/1	1/1	1/1	–
	R	–	–	–	–	–	–	–	1/1	1/1	–	–	–	–	1/1	1/1	–
E-Commerce	P	–	–	–	–	–	–	–	1/1	1/1	–	–	–	1/1	–	1/1	1/1
	R	–	–	–	–	–	–	–	1/1	1/1	–	–	–	1/1	0/1	1/1	1/1
Consul	P	–	–	–	–	–	1/1	1/1	1/1	1/1	–	–	–	1/1	–	–	–
	R	–	–	–	–	–	1/1	1/1	1/1	1/1	–	–	–	1/1	–	–	–
Microservice Demo	P	–	–	–	–	–	–	1/1	1/1	1/1	–	–	–	1/1	–	1/1	0/1
	R	–	–	–	–	–	0/1	1/1	1/1	1/1	0/1	–	–	1/1	–	1/1	–
Qbike	P	–	–	–	–	–	–	–	1/1	–	–	1/1	2/2	–	1/1	–	–
	R	–	–	–	–	–	0/3	–	1/1	–	–	1/1	2/2	0/1	1/1	–	–
Spring Cloud Microservice	P	–	–	–	–	–	–	–	0/1	–	–	–	–	–	–	–	–
	R	–	–	–	0/2	–	0/3	–	–	–	–	–	–	0/1	0/1	–	–
Cqrs Microservice Sampler	P	–	–	–	–	–	–	–	1/1	0/1	–	–	–	1/1	–	1/1	1/1
	R	–	–	–	–	–	0/1	–	1/1	–	–	–	–	1/1	–	1/1	1/1
Spring Boot Microservices	P	–	–	1/1	–	–	–	1/1	1/1	0/1	–	1/1	–	–	–	1/1	1/1
	R	–	–	1/1	–	–	–	1/1	1/1	–	–	1/1	–	0/1	0/1	1/1	1/1
Cloud Strangler Example	P	1/1	–	0/1	–	–	–	–	1/1	–	–	–	2/2	1/1	1/1	0/1	–
	R	1/1	–	–	–	–	–	–	1/1	–	–	–	2/2	1/1	1/1	–	–
Micro Company	P	–	–	1/1	0/2	–	–	–	1/1	–	–	–	–	–	–	0/1	–
	R	0/1	–	1/1	0/1	–	–	–	1/1	–	–	–	–	0/1	–	–	–
MicroService	P	–	–	–	–	–	0/2	–	1/1	1/1	–	–	–	0/1	–	–	–
	R	–	–	–	–	–	–	–	1/1	1/1	–	–	–	–	–	–	–
Microservice Kubernetes	P	–	2/2	–	–	–	0/2	1/1	1/1	1/1	–	–	–	1/1	–	1/1	1/1
	R	–	2/2	–	–	–	–	1/1	1/1	1/1	–	–	–	1/1	–	1/1	1/1
TeaStore	P	–	1/1	0/1	–	–	0/1	1/1	1/1	1/1	1/1	–	–	–	1/1	0/1	1/1
	R	0/1	1/1	–	–	–	–	1/1	1/1	1/1	1/1	–	–	0/1	1/1	–	1/1
Warehouse Microservice	P	–	2/2	–	0/1	–	–	–	1/1	–	–	–	–	–	–	–	–
	R	–	2/2	–	–	–	–	–	1/1	–	–	–	–	0/1	–	–	–
Apollo	P	–	–	1/1	0/2	2/2	–	0/1	0/1	1/1	1/1	–	–	–	1/1	–	–
	R	–	–	1/1	–	2/2	–	–	–	1/1	1/1	–	–	–	1/1	–	–
Delivery System	P	–	–	1/1	–	–	–	–	0/1	0/1	–	–	–	1/1	1/1	1/1	1/1
	R	–	–	1/1	–	–	0/1	–	–	–	–	–	–	1/1	1/1	1/1	1/1
Ticket-Train	P	1/1	–	1/1	15/19	1/1	20/26	1/1	0/1	0/1	–	–	–	–	–	–	–
	R	1/1	–	1/1	15/15	1/1	20/20	1/1	–	–	–	–	–	–	–	–	–
Precision	**Ratio**	**5/5**	**7/7**	**6/10**	**15/24**	**3/3**	**27/38**	**8/10**	**18/23**	**10/14**	**3/3**	**4/4**	**4/5**	**11/14**	**7/8**	**9/13**	**6/9**
	Percentage	**100.00%**	**100.00%**	**60.00%**	**62.50%**	**100.00%**	**71.05%**	**80.00%**	**78.26%**	**71.43%**	**100.00%**	**100.00%**	**80.00%**	**78.57%**	**87.50%**	**69.23%**	**66.67%**
Recall	**Ratio**	**5/7**	**7/7**	**6/6**	**15/18**	**3/3**	**27/41**	**8/8**	**18/18**	**10/10**	**3/4**	**4/4**	**4/4**	**11/17**	**7/10**	**9/9**	**6/6**
	Percentage	**71.43%**	**100.00%**	**100.00%**	**83.33%**	**100.00%**	**65.85%**	**100.00%**	**100.00%**	**100.00%**	**75.00%**	**100.00%**	**100.00%**	**64.71%**	**70.00%**	**100.00%**	**100.00%**

Example. The *Warehouse microservice* system shows interdependence between *product-catalogue-service* and *account-service*. Class instantiations from both microservices packages contribute to this interdependence.

5.1.3. Mega Service (MS)

MARS performed well in detecting mega services, with 60% precision and 100% recall. However, it occasionally misclassified microservices as mega services due to threshold limitations. As shown in Fig. 5, only 25% of the systems in our dataset had mega services.

Example. In the *Apollo* system, the average lines of code per service is *7315*, while the apollo-portal microservice stands out with *43,700* lines of code. With six times the average lines of code, it is identified as a mega service within the system.

5.1.4. Nano Service (NS)

MARS performed well in detecting nano services, with a precision of 62.5% and a recall of 83.3%. However, false positives occurred due to specialised microservices in the systems analysed. Determining nano service classification is subjective and requires domain expertise. Notably, only 13% of the systems in our dataset contained nano services.

Example. In the *Ticket-Train* system, MARS identified the microservice *ts-train-service* as a nano service because it has a few files (10) with a few lines of code (537).

```
env:
  - name: JAVA_OPTS
    value: "-Dsun.net.inetaddr.ttl=30"
  - name: ORDER_DESTINATIONS_ORDERSERVICEURL
    value: http://ftgo-order-service:8080
  - name: ORDER_DESTINATIONS_ORDERHISTORYSERVICEURL
    value: http://ftgo-order-history-service:8080
  - name: CONSUMER_DESTINATIONS_CONSUMERSERVICEURL
    value: http://ftgo-consumer-service:8080
```

Fig. 6. Hard-coded example in the ftgo-api-gateway.yml file in the FTGO application.

5.1.5. Shared Libraries (SL)

MARS effectively detected shared libraries in microservices, achieving 100% precision and recall. Our evaluation found this antipattern in just 2 out of 24 systems, aligning with the "share nothing" principle of microservice architecture, which discourages sharing libraries and other artefacts among microservices.

Example. In the *Ticket-Train* system, the *org.microservices* dependency was shared among 16 microservices, such as *ts-common*, *ts-admin-order-service*, and *ts-admin-route-service*.

5.1.6. Hard-coded Endpoints (HE)

MARS effectively detected hard-coded endpoints in our dataset, with 71.05% precision and 65.85% recall. It identified 27 occurrences but missed some due to URL format variability and their dynamic construction. Detecting all hard-coded endpoints is challenging with static analysis alone. Furthermore, despite service discovery being commonly used, half of the analysed systems still had instances of this antipattern.

Example. In the *FTGO* system, no discovery tool is used. MARS found three hard-coded endpoints in the *ftgo-api-gateway* service, as shown in Fig. 6.

5.1.7. Manual Configuration (MC)

We found that MARS effectively detects the manual configuration antipattern with 80% precision and 100% recall. However, there were two false positives caused by configuration-related libraries in systems' dependencies that were not used for microservices configuration. We observed that 33% of the analysed systems rely on manual configuration, despite the availability of configuration management frameworks.

Example. In the *Consul* system, the *microservice-consul-demo-catalogue* microservice declares two configuration files without any configuration tool: *application.properties* and *bootstrap.properties*.

5.1.8. No CI/CD (NCI)

MARS effectively detected the No CI/CD (NCI) antipattern, with 78.26% precision and 100% recall. However, it falsely identified this antipattern in only three projects, as CI/CD tool usage is often visible only in the production environment of microservices-based systems. For example, in the *Ticket-Train* system, MARS did not detect CI/CD tool usage initially, but a manual analysis of GitHub releases revealed its presence. Overall, only 35% of the analysed systems use CI/CD in their development pipelines.

Example. In the *Micro Company* system, MARS found no continuous integration tool such as Travis or Jenkins.

```
service: ftgo-application-lambda

provider:
  name: aws
  runtime: java8
  timeout: 35
  region: ${env:AWS_REGION}
```

Fig. 7. Example of a timeouts example in the serverless.yml file in the FTGO application.

5.1.9. No API Gateway (NAG)

We found that MARS effectively detects the occurrence of this antipattern with a precision of 71.43% and a recall of 100%. We observed that 42% of the studied microservice-based systems use API gateways. Despite their advantages, some developers of the systems in our dataset did not use API gateways, possibly due to the additional complexity introduced by their use. The nature of the systems in our dataset (non-commercial or industrial systems) could also explain this observation.

Example. MARS did not found an API gateway in the *Micro Company* system. The architecture of this system has been designed without any API gateway.

5.1.10. Timeouts (TO)

MARS effectively detected the timeout antipattern in microservice-based systems, achieving a precision rate of 100% and a recall rate of 75%. However, one instance of this antipattern was not detected due to the system's use of an implementation unsupported by MARS (timeouts in Java annotations). We also found that the timeout antipattern was only present in 17% of the systems we analysed. Furthermore, we observed that developers tended to prioritise fault tolerance and resilience by using circuit breakers rather than by specifying timeout values when invoking microservices.

Example. In the *FTGO* system, we found a hard-coded timeout in the configuration file of the *ftgo-application-lambda* microservice. We can see this instance on Line 6 of Fig. 7, where a timeout parameter has been set to 35.

5.1.11. Multiple Service Instances Per Host (MSIPH)

We found that MARS performed well in detecting the MSIPH antipattern, with a precision and recall of 100%. We observed that only 17% of the microservice-based systems analysed in our study use a unique host to deploy their microservices. This practice enhances scalability and allows for multiple versions of the same microservice to be run simultaneously.

Example. MARS detected the presence of this antipattern in the *Qbike* system, in which five microservices were deployed on the same host.

5.1.12. Shared Persistence (SP)

For the shared persistence antipattern, MARS achieved a precision of 80% and a recall of 100%. It successfully detected a database that is shared among five microservices in FTGO (*Eventuate*). However, we did not include it in the ground truth because the shared database was used for event sourcing. This antipattern appeared in only 8% of the systems in our dataset, suggesting that developers prioritise microservices' independent data management over shared databases.

Example. In the *Cloud Strangler* system, MARS found that one MySQL database was shared by two microservices: *profile-service* and *user-service*.

5.1.13. No API Versioning (NAV)

MARS achieved a precision of 78.57% and a recall of 64.71% in detecting the no API versioning antipattern. We observed that only 25% of the microservice-based systems in our dataset have implemented API versioning. This could be due to the perceived complexity of implementing versioning or because the systems' requirements do not necessitate such a feature.

Example. We detected this antipattern in the *Piggymetrics* system. No implementation of API versioning was found in this system.

5.1.14. No Health Check (NHC)

MARS effectively detected the no health check antipattern, with 88.89% precision and 66.67% recall. Surprisingly, 42% of systems in our dataset do not perform periodic health checks on their microservices. This omission increases the risk of system failures and unavailability, emphasising the need for incorporating health checks as a critical aspect of microservice design and deployment.

Example. We detected this antipattern in the *Freddys BBQ* system. We did not find any health check in the *microsec-admin-portal*, *microsec-common*, *microsec-custom-registry*, or *microsec-customer-portal* services, or the system itself.

5.1.15. Local Logging (LL)

The detection results of MARS were satisfactory for the local logging antipattern, with a precision of 69.23% and a recall of 100%. We observed that this practice is relatively widespread: 38% of the analysed microservice-based systems use local logging mechanisms, which should be avoided because they prevent the tracing of failures among microservices.

Example. The antipattern was detected neither by MARS nor in the ground truth in the *Spring Netflix OSS* system. The system does not contain any logging tool.

5.1.16. Insufficient Monitoring (IM)

MARS detected the insufficient monitoring antipattern with a precision of 66.67% and a recall of 100%. We observed that the false positives generated by MARS are due to the presence of monitoring tools in some microservice-based systems that are not listed in MARS's configuration files. Finally, we observed that 75% of the systems in our dataset use monitoring tools, which are crucial to trace issues as soon as possible.

Example. No monitoring tool was identified by MARS in the *Microservice Kubernetes* system. The system has no monitoring tool, such as *Grafana*, *Zipkin*, or *Heapster*.

5.2. Comparison with a baseline approach

We compared the detection results of MARS with *MSANose* (Walker et al., 2020), a tool for detecting microservice antipatterns based on static analysis of source code. We selected this tool because it is open-source, supports analysis of microservice-based systems implemented in Java, and covers 11 microservice antipatterns, eight of which are in common with MARS. We considered only the antipatterns that are common to the two tools and tried to replicate *MSANose*'s results on the *Ticket-Train* system since it is already included in our dataset. We found that *MSANose* detected only occurrences of the shared libraries and no API versioning antipatterns. We observed for shared libraries an average detection precision of 1.5% and a recall of 100%. The tool generated a large number of false positives when detecting shared libraries because it only compares the names of the list of libraries used by different microservices, and does not exclude local libraries of which the source code or runtime assets are duplicated in the repositories of microservices. Additionally, we observed for no API versioning an average detection precision of 57% and recall of 47%. For this antipattern, we found that *MSANose* only examines the files of each microservice, rather than the entire system's configuration files, resulting in relatively low precision and recall. We conclude that MARS clearly outperforms *MSANose* in detecting microservice antipatterns. MARS does not only detect more accurately shared libraries and no API versioning than *MSANose* but also covers more microservice antipatterns.

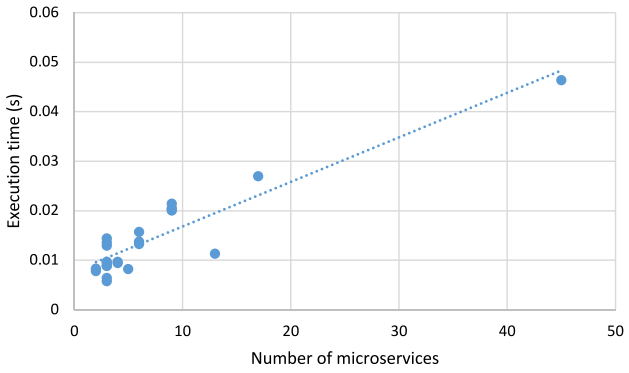

Fig. 8. Execution time per #Microservices.

5.3. Performance analysis

To evaluate the scalability and efficiency of our approach, we performed both a theoretical and an empirical analysis of the time complexity of MARS. The theoretical analysis considers the computational complexity of our approach's key functions, including the model generation and detection rules. This theoretical analysis aims to evaluate the scalability of MARS on large systems, since large and/or industrial Java microservice-based systems are not accessible to allow the testing of the scalability of our tool. The empirical evaluation focuses on the execution time of MARS and its dependence on the number of microservices, lines of code, or files in a system, to evaluate the efficiency of our approach in larger systems.

Time Complexity Analysis. Our analysis shows that the time complexity of MARS is $O(n(k+m))$, where n is the number of microservices in the system, m is the maximum number of dependencies of the microservices, and k is the maximum number of files in each microservice. We note that the time complexity of the antipatterns detection functions dominates the overall time complexity, as they involve the analysis of the dependencies and configuration files of all microservices.

Our analysis demonstrates that the time complexity of our approach grows linearly with the number of microservices in the system being analysed. This growth pattern indicates that our approach is scalable and can be applied to large microservice-based systems. Our approach can efficiently evaluate industrial-scale microservice-based systems, as it scales with the size of the system. Details of our time complexity analysis, including the time complexity of each function, can be found in the replication package provided with our paper.[10]

Execution Time Analysis. To complement our theoretical analysis, we conducted an empirical analysis of the execution time of MARS on the microservice-based systems. We studied the impact

[10] https://github.com/LoicMadern/MARS/blob/main/MarsTimeComplexity.txt

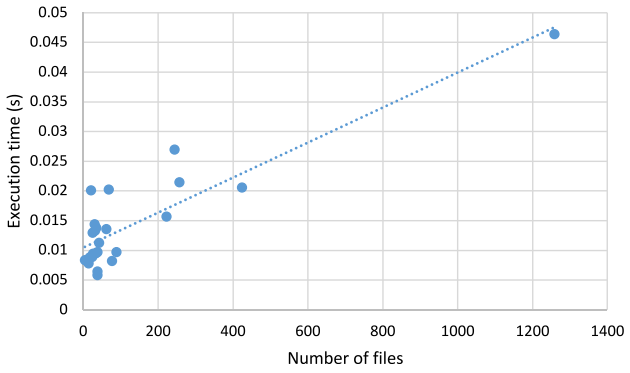

Fig. 9. Execution time per #Files.

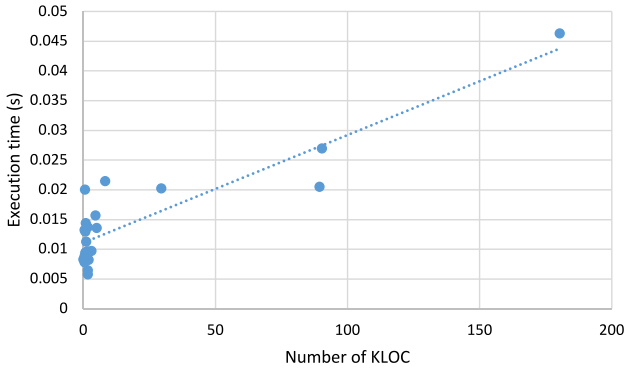

Fig. 10. Execution time per #KLOC.

of factors such as the number of microservices, lines of code, and files on the efficiency of MARS in detecting microservice antipatterns. We executed MARS four times and calculated the average execution time for each system in our dataset. This analysis demonstrated that MARS can detect antipatterns efficiently, with execution times ranging from 0.005 s to 0.05 s on average. The results of our analysis are presented in Figs. 8, 9, and 10. These plots show that MARS scales well with the number of microservices, lines of code, and files in the systems. This empirical analysis also provides evidence of the efficiency and scalability of MARS in detecting microservice antipatterns.

6. Discussions

We will describe in this section other observations that we made in our experiments. We will also discuss the limitation of MARS and the implication of our results for researchers and practitioners.

6.1. Other observations

Regarding hard-coded 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 (Hunkeler et al., 2008) or the *CQRS* architectural pattern[11], typically using *RabbitMQ* as in the *CQRS Microservice Sampler* system,[12] implies that endpoints must be hard-coded 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. Even the two state-of-the-art microservice antipatterns detection approaches (Walker et al., 2020; Pigazzini et al., 2020), have their limitations when it comes to effectively identifying hard-coded endpoints and timeouts. For example, *MSA-NOSE* (Walker et al., 2020) is constrained to specific frameworks and faces challenges in capturing dynamically derived configurations. On the other hand, the regex-based approach employed by Pigazzini et al. (2020) can miss variations and produce false positives. These limitations highlight the need for further improvement in detecting and addressing the presence of hard-coded endpoints and timeouts in microservice-based systems.

Regarding no health check, it is possible that some microservice-based systems may use frameworks (i.e. Kubernetes Anon 2022a and Openliberty Anon 2022b) that, by default, enable this check. In such a case, MARS may have detected an occurrence of this antipattern, even though the health check is enabled by default into the framework. Future work could include improving MARS to consider the idiosyncrasies and default behaviour of different frameworks. However, it is worth mentioning that existing approaches for detecting microservice antipatterns (Borges and Khan, 2019; Walker et al., 2020; Pigazzini et al., 2020) have not addressed the identification of this particular antipattern, which is one of the contributions of our work.

In some microservice-based systems, we observed that an antipattern may be both present *and* not present, which we call a "Schrödinger occurrence", in reference to the "Schrödinger's cat" thought experiment (Monroe et al., 1996). A Schrödinger occurrence occurs, in particular, when repositories include both local and multi-host deployments or both monolithic and service-based versions of a system, e.g., the repositories of *LakeSide Mutual* (local and multi-host deployments) and of *Micro Company* (monolithic and microservice-based versions). For repositories containing both monolithic and microservice-based versions, we excluded 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* contains a discovery service but some of its services use hard-coded endpoints. In such a case, MARS reports occurrences of the hard-coded endpoint antipattern, even though they are due to developers' overlooking some configurations rather than to a poor design.

Similarly to existing studies (Walker et al., 2020; Taibi and Lenarduzzi, 2018), we did not rely on the microservice coupling to detect wrong cuts in microservice-based systems. Although the presence of wrong cuts in microservices potentially increases the coupling between associated microservices, this does not necessarily imply that all highly coupled microservices will exhibit this antipattern. Indeed, we observed in our dataset that highly coupled microservices may occur for other reasons, such as the centrality and importance of a microservice's business functionality, or the presence of mega microservices. Thus, we relied on an analysis of the distribution of file extensions, content, and programming languages in microservices to identify wrong cuts (i.e., mono-layered services).

Given that determining the optimal size for a microservice is a complex endeavour, influenced by factors like functional cohesion, detecting mega and nano services is subjective as it relies on manual specification of thresholds that can vary from one expert and one system to another (Vural et al., 2018). Thus, we recommend creating boxplots of the number of lines of code and files in the system being analysed. An expert can thus visualise their distributions and determine appropriate thresholds based

11 https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs
12 https://github.com/benwilcock/cqrs-microservice-sampler

on the quartiles of the boxplots. We will automate the detection of such antipatterns and automatically set appropriate thresholds in our detection rules in future work.

We validated MARS on 24 open-source systems, including one industrial microservice-based system (Apollo). Practitioners could use MARS for the analysis of their microservice-based systems. Indeed, to detect some antipatterns such as WC, MS, and NS, MARS does not need to analyse the entire microservice-based system to detect their occurrences: it only needs to parse the list of URLs to the related distributed microservices repositories. It will then extract all the metadata required to detect antipatterns. In future work, we plan to deploy MARS as a service that can be accessed remotely, facilitating its use by both researchers and practitioners. We also plan to conduct additional experiments on larger and more complex microservice-based systems to validate our theoretical analysis of the scalability of MARS and demonstrate the practicality and efficiency of our approach for industrial microservice-based systems.

6.2. Limitations of MARS

Our microservice antipattern identification approach, while effective, does have limitations that should be acknowledged. One of these limitations pertains to the identification of the circular dependency antipattern. Unlike existing approaches in the literature (Walker et al., 2020; Pigazzini et al., 2020), MARS detects circular dependencies only between pairs of microservices. While this is useful, there may be cases in which circular dependencies involve more than two microservices. Our current approach does not capture such complex circular dependencies involving three or more microservices. We aim in future work to address this limitation and improve MARS to detect complex circular dependencies involving three or more microservices.

Additionally, it is worth noting that the identification of certain antipatterns, such as local logging and insufficient monitoring, depends strongly on the specification of a list of related libraries. This aspect introduces another limitation to our approach. However, existing approaches for detecting microservice antipatterns (Borges and Khan, 2019; Walker et al., 2020; Pigazzini et al., 2020) have not considered the identification of those antipatterns. Thus, we have made a novel contribution by incorporating the detection and the analysis of the local logging, insufficient monitoring, and no health check antipatterns within our approach. Furthermore, distributed logging and monitoring tools may be configured when the microservice-based system is deployed and may not be explicit in the configuration or dependency files. Even if distributed logging and monitoring tools exist in the deployment infrastructures (e.g., Kubernetes), MARS will only detect occurrences of the local logging and insufficient monitoring antipatterns when it cannot find dependencies to logging or monitoring tools in the source code of the microservice-based system. MARS has this limitation because we have no visibility into the deployment infrastructure of the microservice-based systems. MARS only statically analyses the source code of microservice-based systems, independently of their deployment infrastructures. Still, reporting the presence of these antipatterns as detected by MARS could encourage developers to perform a more in-depth investigation of logging and monitoring tools in their deployment infrastructure.

Finally, the identification of certain antipatterns, such as wrong cuts, nano services and mega services, requires the specification of thresholds for their identification. The determination of the optimal size for a microservice is challenging. It involves various factors such as functional cohesion, service autonomy, scalability, maintainability, and deployment flexibility (Vural et al., 2018). Finding the right balance of microservice sizes requires careful consideration and understanding of the context. Considering this complexity, our approach relies on expert customisation to set thresholds for detecting antipatterns related to microservices sizes. We acknowledge that this reliance on expert customisation introduces subjectivity and requires domain expertise, thereby posing a limitation (Vural et al., 2018). As part of our future work, we intend to explore automated techniques that can assist in determining these thresholds. By leveraging automated approaches, we aim to reduce the reliance on manual customisation and human expertise, thereby improving the objectivity and scalability of our antipattern identification process.

6.3. Implications for researchers and practitioners

This study provides a comprehensive approach for specifying and detecting antipatterns in microservice-based systems, which can benefit both researchers and practitioners interested in improving the quality of such systems. By identifying the presence of antipatterns and evaluating the adoption of microservice-related best practices, MARS enables practitioners to pinpoint areas for improvement and perform necessary refactorings as needed. These refactorings may include changes to the source code of individual microservices or the implementation of monitoring, CI/CD, or health check tools to enhance system performance and resilience.

By analysing the systems in our dataset, we have identified several recommended practices that developers tend to ignore in microservice-based systems. Notably, we found that API versioning, CI/CD, health checks, and API gateways are often ignored. In contrast, circular dependencies among microservices are generally avoided, and libraries, as well as databases, are not shared among microservices. These observations are preliminary but provide valuable insights into developers' preferences and practices in microservice architecture. They can also serve as a starting point for further research to understand why certain practices are favoured over others and how to address the challenges associated with adopting best practices in microservices. Some of these observations are consistent with the findings of a recent industrial survey on microservice-based systems (Waseem et al., 2021), which reported that developers tend to avoid sharing databases between microservices. However, it was also found that health checks for microservices and API gateways are commonly used in industrial microservice-based systems (Waseem et al., 2021). To confirm and expand on our findings, we recommend performing further analyses on larger datasets and conducting interviews with developers to gain a more nuanced understanding of their perspectives and experiences.

Based on this work, we can make several recommendations, both for developers of microservice-based systems and for researchers in the field of microservice architecture and their antipatterns. First, several technologies and concepts used in microservicescan, in certain contexts, bring excessive complexity in comparison to the problems that they solve. This is the case, for example, with API gateways (Taibi and Lenarduzzi, 2018). It is common in the industry to view this design pattern as bringing unhelpful complexity to systems with a small number of microservices. Indeed, some existing studies (Walker et al., 2020; Taibi and Lenarduzzi, 2018) have reported that developers can adequately manage up to 50 distinct microservices without needing to rely on an API gateway. It would be interesting for researchers to study the impact of the complexity added by an API gateway in a small system compared to direct communication with microservices. However, it is always preferable to design a microservice-based system by following as many best practices as possible if it is considered likely to grow in the future.

Second, we recommend automating as many tasks as possible in the development process of microservice-based systems,

because automation is a fundamental tenet of microservices and many of their advantages rely on automation. We recommend automation of testing, configuration, deployment, and monitoring for increased agility and responsiveness.

Finally, we recommend the logging and tracking of all information and events emitted by microservices, not only to improve the discovery of errors and failures but also their correction and the building of knowledge bases to prevent their future reproduction.

7. Threats to validity

We now discuss threats to the validity of our study.

Construct Validity The rules that we used to detect antipatterns have been specified according to our interpretation of the antipatterns. They are based on practices within the development community, analysis of microservices, and our own experience and understanding. We tried to minimise any bias by carefully considering previous work Tighilt et al. (2020), describing the rules in this paper, and providing an open-source implementation of all antipattern detection rules to allow them to be updated, refined, and compared with other works.

Although we extensively reviewed the literature to identify the most common antipatterns in microservice-based systems, other antipatterns may exist. In future work, we aim to extend MARS to allow the specification of additional antipatterns, by us or by others. Making MARS and its source code publicly available will facilitate this.

Internal Validity Three authors were involved in manually and independently identifying occurrences of the microservice antipatterns in the 24 systems and reconciling them to obtain a ground truth for the validation. We thus tried to remove any bias towards our rules. We have also published the ground truth online so that other researchers can vet and use it.

External Validity Microservice-based systems are volatile. They can be built using multiple technologies, deployed on multiple hosts, and changed easily Dragoni et al. (2017). Although we tried to identify and consider the most common technologies for microservices in MARS, we considered only Java microservices, potentially omitting some important systems and technologies. We minimised this threat by building and providing a tool that can be extended with additional parsers.

The limited number of detected microservice antipatterns may threaten the generalisability of our results. However, it is important to highlight that our study represents the most extensive empirical analysis conducted to date on the detection of microservice antipatterns. In our research, we used automated techniques to accurately detect a total of 16 distinct microservice antipatterns within a dataset consisting of 24 microservice-based systems. These systems exhibited 172 occurrences of microservice antipatterns, which constitute a larger validation dataset than in existing works Borges and Khan (2019), Walker et al. (2020), Pigazzini et al. (2020). To mitigate this threat, we also considered diverse microservice-based systems of different sizes. We aim in future work to consider more systems and identify a larger spectrum of microservice antipattern occurrences.

We relied on lists of dependencies and frameworks to identify some antipatterns. We assembled these lists by considering the technologies used most widely to develop microservices, but they are certainly not exhaustive. To mitigate this threat, we designed MARS to be able to use 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 wrongly conclude that a file is not a configuration file and exclude it. We reduced this threat by relying on file extensions and also on the file names and their content to perform the classification.

8. Conclusion

We proposed MARS, a tool-based approach to automatically detect antipatterns in microservice-based systems. We provide (1) an extensible and technology-agnostic metamodel for detecting microservices antipatterns, and (2) an accurate and open-source tool that we empirically validated on 24 microservice-based systems using a ground truth of 172 occurrences of microservice antipatterns.

Manual validation of the detected occurrences showed that MARS allowed us to specify and detect microservice antipatterns with an average precision of 82% and a recall of 89%. Our work is useful to both practitioners and researchers. It provides the first complete approach for specifying and detecting microservice antipatterns and can be used as guidance to assess the quality of microservice-based systems.

In future work, we plan to focus on improving the detection of specific antipatterns, such as circular dependencies, and expanding our analysis to identify additional antipatterns and examine their prevalence in existing microservice-based systems. We also aim to empirically and quantitatively study the presence of microservice antipatterns in a larger dataset and study their impact on maintenance. This should enable us to make further recommendations to developers and researchers on good and bad practices to consider when developing microservice-based systems.

CRediT authorship contribution statement

Rafik Tighilt: Software, and writing. **Manel Abdellatif:** Supervision, software, validation, writing, and reviewing. **Imen Trabelsi:** Validation, writing, and reviewing. **Loïc Madern:** Software, validation, and writing. **Naouel Moha:** Supervision, and reviewing. **Yann-Gaël Guéhéneuc:** Supervision, and reviewing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data available at https://github.com/LoicMadern/MARS

Acknowledgments

This work was funded by the Fonds de Recherche du Québec–Nature et Technologie (FRQNT).

References

Alshuqayran, N., Ali, N., Evans, R., 2018. Towards micro service architecture recovery: An empirical study. In: IEEE International Conference on Software Architecture. ICSA, pp. 47–4709.

Anon, 2020. Microservices: a definition of this new architectural term. https://martinfowler.com/articles/microservices.html, (Accessed 01 August 2020).

Anon, 2022a. Kubernetes. https://kubernetes.io. (Accessed 14 December 2022).

Anon, 2022b. Openliberty. https://openliberty.io, (Accessed 14 December 2022).

Borges, Rodrigo, Khan, Tanveer, 2019. Algorithm for detecting antipatterns in microservices projects. In: Joint Proceedings of the Inforte Summer School on Software Maintenance and Evolution. CEUR-WS, pp. 21–29.

Dragoni, Nicola, Lanese, Ivan, Larsen, Stephan Thordal, Mazzara, Manuel, Mustafin, Ruslan, Safina, Larisa, 2017. Microservices: How to make your application scale. In: International Andrei Ershov Memorial Conference on Perspectives of System Informatics. Springer, pp. 95–104.

Granchelli, Giona, Cardarelli, Mario, Francesco, P., Malavolta, Ivano, Iovino, L., Salle, Amleto Di, 2017. Towards recovering the software architecture of microservice-based systems. In: IEEE International Conference on Software Architecture Workshops. ICSAW, pp. 46–53.

Hecht, Geoffrey, Rouvoy, Romain, Moha, Naouel, Duchien, Laurence, 2015. Detecting antipatterns in android apps. In: Proceedings of the 2nd ACM International Conference on Mobile Software Engineering and Systems. IEEE, pp. 148–149.

Hunkeler, Urs, Truong, Hong Linh, Stanford-Clark, Andy, 2008. MQTT-S—A publish/subscribe protocol for wireless sensor networks. In: The 3rd International Conference on Communication Systems Software and Middleware and Workshops. IEEE, pp. 791–798.

Kitchenham, Barbara, 2004. Procedures for performing systematic reviews, vol. 33. Keele University, Keele, UK, pp. 1–26.

Marquez, Gaston, Astudillo, Hernan, 2018. Actual Use of Architectural Patterns in Microservices-Based Open Source Projects. In: Proceedings of the 25th Asia-Pacific Software Engineering Conference (APSEC). IEEE, pp. 31–40.

Moha, Naouel, Guéhéneuc, Yann-Gaël, Duchien, Laurence, Le Meur, Anne-Francoise, 2009. Decor: A method for the specification and detection of code and design smells. IEEE Trans. Softw. Eng. 36 (1), 20–36.

Monroe, Christopher, Meekhof, DM, King, BE, Wineland, David J, 1996. A "Schrödinger cat" superposition state of an atom. Science 272 (5265), 1131–1136.

Newman, Sam, 2015. Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, Inc., ISBN: 978-1491950357.

Osses, Felipe, Marquez, Gaston, Astudillo, Hernan, 2018. 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, pp. 256–257.

Pahl, Claus, Jamshidi, Pooyan, 2016. Microservices: A Systematic Mapping Study. In: Proceedings of the 6th International Conference on Cloud Computing and Services Science. SCITEPRESS - Science and and Technology Publications, pp. 137–146.

Palma, Francis, 2013. Detection of SOA antipatterns. In: The 11th International Conference on Service-Oriented Computing (ICSOC) Workshops. Springer Berlin Heidelberg, pp. 412–418.

Palomba, Fabio, Di Nucci, Dario, Panichella, Annibale, Zaidman, Andy, De Lucia, Andrea, 2017. Lightweight detection of android-specific code smells: The adoctor project. In: IEEE 24th International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, pp. 487–491.

Pigazzini, Ilaria, Fontana, Francesca Arcelli, Lenarduzzi, Valentina, Taibi, Davide, 2020. Towards microservice smells detection. In: Proceedings of the 3rd International Conference on Technical Debt. pp. 92–97.

Pulnil, Sermsook, Senivongse, Twittie, 2022. A microservices quality model based on microservices anti-patterns. In: The 19th International Joint Conference on Computer Science and Software Engineering. JCSSE, IEEE, pp. 1–6.

Rademacher, Florian, Sachweh, Sabine, Zündorf, Albert, 2020. A modeling method for systematic architecture reconstruction of microservice-based software systems. In: Enterprise, Business-Process and Information Systems Modeling. Springer, ISBN: 978-3-030-49418-6, pp. 311–326.

Rahman, Mohammad Imranur, Panichella, Sebastiano, Taibi, Davide, 2019. A curated dataset of microservices-based systems. In: SSSME-2019. CEUR-WS.

Taibi, D., Lenarduzzi, V., 2018. On the definition of microservice bad smells. IEEE Softw. 35, 56–62.

Taibi, Davide, Lenarduzzi, Valentina, Pahl, Claus, 2020. Microservices anti-patterns: A taxonomy. In: Microservices: Science and Engineering. Springer, pp. 111–128.

Tighilt, Rafik, Abdellatif, Manel, Moha, Naouel, Mili, Hafedh, Boussaidi, Ghizlane El, Privat, Jean, Guéhéneuc, Yann-Gaël, 2020. On the study of microservices antipatterns: A catalog proposal. In: The 2020 Proceedings of the European Conference on Pattern Languages of Programs. pp. 1–13.

Vural, Hulya, Koyuncu, Murat, Misra, Sanjay, 2018. A case study on measuring the size of microservices. In: The 18th International Conference of Computational Science and Its Applications. Springer, pp. 454–463.

Walker, Andrew, Das, Dipta, Cerny, Tomas, 2020. Automated code-smell detection in microservices through static analysis: A case study. In: Appl. Sci.. 10, (21), MDPI AG, (ISSN: 2076-3417) p. 7800.

Waseem, Muhammad, Liang, Peng, Shahin, Mojtaba, Di Salle, Amleto, Márquez, Gastón, 2021. Design, monitoring, and testing of microservices systems: The practitioners' perspective. 182, (ISSN: 0164-1212) 111061. http://dx.doi.org/10.1016/j.jss.2021.111061, URL https://www.sciencedirect.com/science/article/pii/S0164121221001588,

Zimmermann, Olaf, 2016. Microservices tenets: : Agile approach to service development and deployment. Comput. Sci. Res. Dev. 21, 301–310.