The Journal of Systems & Software 206 (2023) 111829

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Catalog and detection techniques of microservice anti-patterns and bad smells: A tertiary study☆

Tomas Cerny [a], Amr S. Abdelfattah [b], Abdullah Al Maruf [b], Andrea Janes [c], Davide Taibi [d,e,*]

[a] SIE, University of Arizona, Tucson, AZ, USA
[b] Baylor University, Waco, TX, USA
[c] FHV Vorarlberg University of Applied Sciences, Dornbirn, Austria
[d] Tampere University, Tampere, Finland
[e] University of Oulu, Oulu, Finland

ARTICLE INFO

Article history:
Received 5 January 2023
Received in revised form 13 June 2023
Accepted 29 August 2023
Available online 15 September 2023

Keywords:
Microservices
Anti-patterns
Antipatterns
Anti patterns
Bad smells
Software maintenance

ABSTRACT

Background: Various works investigated microservice anti-patterns and bad smells in the past few years. We identified seven secondary publications that summarize these, but they have little overlap in purpose and often use different terms to describe the identified anti-patterns and smells.
Objective: This work catalogs recurring bad design practices known as anti-patterns and bad smells for microservice architectures, and provides a classification into categories as well as methods for detecting these practices.
Method: We conducted a systematic literature review in the form of a tertiary study targeting secondary studies identifying poor design practices for microservices.
Results: We provide a comprehensive catalog of 58 disjoint anti-patterns, grouped into five categories, which we derived from 203 originally identified anti-patterns for microservices.
Conclusion: The results provide a reference to microservice developers to design better-quality systems and researchers who aim to detect system quality based on anti-patterns. It also serves as an anti-pattern catalog for development-aiding tools, which are not currently available for microservice system development but could mitigate quality degradation throughout system evolution.

1. Introduction

Microservices have gained significant popularity as an appealing architectural choice, particularly in situations where independent scalability and changeability of system components are desired. They have garnered significant acceptance within the industry since 2014 (Lewis and Fowler, 2014). The 2022 Service Mesh Adoption Survey—an annual survey mapping how organizations are adopting microservices—determined that 85% of companies are modernizing their applications with the microservice architecture (solo.io, 2022), which indicates the importance of advancing knowledge in microservice system design, defined microservices as "the fueling architecture" for cloud-native systems (Balalaie et al., 2016; Carnell and Sánchez, 2021). Cloud-native systems (as opposed to cloud-ready) are designed specifically for a cloud computing architecture and are able to take advantage of its benefits and services.

To facilitate system development, practitioners and researchers identified various proven solutions to common recurring design problems. These are typically named and referred to as *design patterns*. Furthermore, such design patterns apply to specific software architectures (Carnell and Sánchez, 2021; Zimmermann, 2022; Zimmermann et al., 2019, 2020c,a,b; Stocker et al., 2018). Unfortunately, not all design patterns identified for a particular software architecture can be transferred to other architectures. Similarly, `Anti-patterns` (Brown et al., 1998; Taibi et al., 2020a; Walker et al., 2020) are recurring design *practices, choices or solutions to common problems* despite appearing reasonable and effective, lead to negative consequences and undermine the system's overall quality. Depending on the context (monolith, SOA, microservices, etc.), a design might be an anti-pattern in one case and a sound design in another case. Apart from anti-patterns, the term `bad smell` describes a design characteristic that indicates a potential problem or violation of good practices. A smell is *suspicion that something is not right* (external observations). It is a warning sign that suggests potential issues in the design but does not provide specific solutions. They prompt further analysis and consideration to identify the underlying problems and propose appropriate design improvements.

☆ Editor: Jacopo Soldani.

* Corresponding author at: University of Oulu, Oulu, Finland.
E-mail addresses: tcerny@arizona.edu (T. Cerny), amr_elsayed1@baylor.edu (A.S. Abdelfattah), maruf_maruf1@baylor.edu (A.A. Maruf), andrea.janes@fhv.at (A. Janes), davide.taibi@oulu.fi (D. Taibi).

https://doi.org/10.1016/j.jss.2023.111829

Researchers have been using the smell metaphor to describe patterns and decisions associated with bad design (van Emden and Moonen, 2002) that may lead to difficulties in maintaining, evolving, or scaling the system (Yamashita and Moonen, 2013). Various works use anti-pattern and bad smell synonymously (Bogner et al., 2019a), for example: "shotgun surgery" is called an "antipattern" by Wikipedia (Wikipedia contributors, 2023), but a "code smell" by RefactoringGuru (Refactoring.Guru, 2023); others see these terms as distinct concepts (Garcia et al., 2009a).

Anti-patterns should be seen as *root causes* for smells, therefore, in this study, we use the term `MS anti-pattern` to refer to microservice antipattern, "bad" smell, and poor "design" practice. In addition, literature uses three forms of this term: `antipattern`, `anti pattern` and anti-pattern; however, the last term form is more commonly used in the scientific literature.

In our previous research (Taibi et al., 2020a; Walker et al., 2020), we encountered various anti-patterns and smells specific to microservices. While many mapping studies and secondary studies (SS) have been performed on this topic, we identified disjoint and partial overlaps across such works. Anyone interested in this topic and its relevance to microservices will likely be interested in a comprehensive literature study combining the current knowledge on this topic. For this reason, the present study catalogs smells and anti-patterns identified in literature reviews and combines them to track their origin and source. To provide reasonable indexing, we categorize these patterns and look into mechanisms that have been mentioned in the literature to detect them, providing a broad overview to the community.

Our approach to conducting this study is to perform a *tertiary* study based on best practice guidelines for reporting secondary studies by Kitchenham et al. (2022). We also consider existing tertiary studies as an established practice to perform and report these studies.

1.1. Objectives

The objective of this tertiary study comprises of four questions related to microservice anti-patterns investigating the following Research Questions (RQs):

Research Question 1: *What secondary studies have been published about MS anti-patterns?*

Rationale: The aim is to identify the list of secondary studies summarizing, and classifying MS anti-patterns.

Research Question 2: *Which distinct MS anti-patterns have been identified in the secondary studies?*

Rationale: This investigation aims to extract the list of unique MS anti-patterns reported in the secondary studies.

Research Question 3: *How are the MS anti-patterns classified in the secondary studies?*

Rationale: This question aims at extracting information on how MS anti-patterns were classified in the secondary studies, so as to understand what problems did they address and if there are overlaps across existing works (e.g., performance patterns, organizational patterns, etc.).

Research Question 4: *How can MS anti-patterns be detected?*

Rationale: This investigation summarizes the detection method reported in the secondary studies. Moreover, we propose a comprehensive framework to classify the detection techniques.

These questions are related and have dependencies among themselves. They are investigated in the consequential order. The process to answer the research question is depicted in Fig. 1, which also illustrates the dependencies within the different steps in this work.

Other tertiary studies

Other tertiary studies have been conducted in software engineering. Examples are tertiary studies on technical debt management (Junior and Travassos, 2022), risk mitigation in global software development (Verner et al., 2014), gamification in software engineering (García-Mireles and Morales-Trujillo, 2020), software product lines and variability modeling (Raatikainen et al., 2019), agile software development (Hoda et al., 2017), and test artifact quality (Tran et al., 2021). However, to the best of our knowledge, no tertiary study exists on MS anti-patterns.

Contribution

We recognize the importance of this research, given community interest in this topic and the current gaps in anti-pattern detection tools for microservices. Since one has to deal with decentralization and possibly heterogeneity, it is more challenging for detection tools to operate. The connection between microservices is non-obvious from source code analysis, which disables current tools to detect issues relevant to the more holistic system perspective. Obviously, ideal detection tools would be capable of identifying a wide range of anti-patterns; however, at this point, the knowledge of MS anti-patterns is scattered across multiple works that have various overlaps, use different terms for the same anti-patterns or combine anti-patterns with component-based development or legacy architectures.

The main contribution of this work is twofold:

- A comprehensive catalog of MS anti-patterns, classified into five categories.
- A framework to classify the different anti-pattern detection techniques, extending the existing body of knowledge reported in secondary studies.

The resulting anti-pattern catalog will help train the skilled workforce in proper practices and avoid outdated design practices that might be carried out from monolith system development. This catalog can also help to bridge current gaps in automated anti-pattern detection when implementing comprehensive quality assessment software, which is currently missing for microservices (Walker et al., 2020). The anti-pattern detection approaches are further analyzed, and a classification framework is provided in this study. Such a framework takes into account established methods for information extraction, intermediate representation, and actual detection.

As a result, the progress made in this research contributes to a deeper comprehension of inadequate design practices associated with microservices from diverse viewpoints. Furthermore, it demonstrates how these anti-patterns can be automatically detected within the decentralized nature of the system, even with independent microservice evolution and distributed

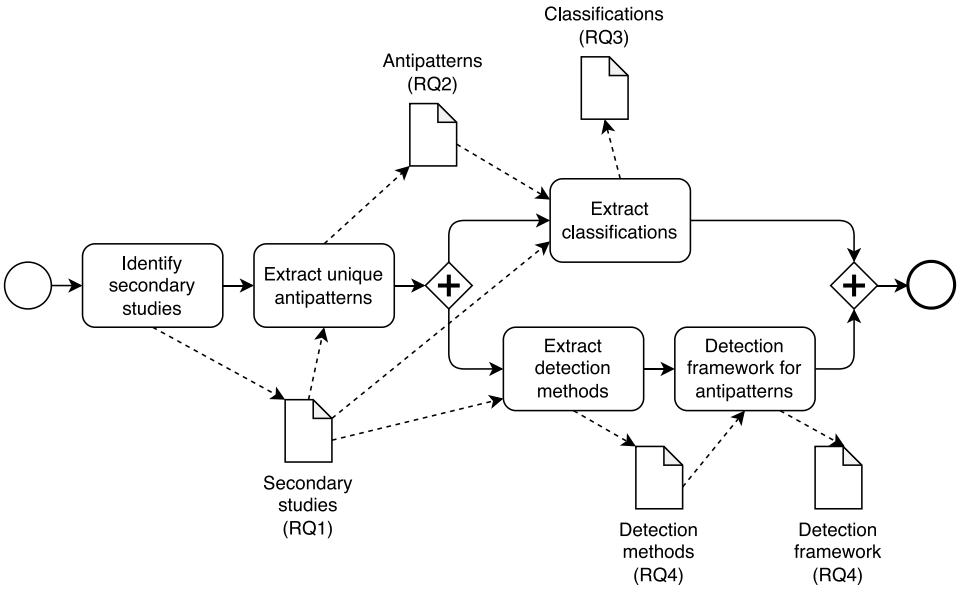

Fig. 1. BPMN diagram illustrating the process we followed to answer the research questions.

infrastructure. Automating this process would enhance the efficiency of addressing system quality degradation, technical debt, and suboptimal practices that are often challenging to identify manually.

Organization of this paper

The remainder of this study is organized as follows: Section 2 introduces the tertiary study execution method and details data analysis procedures. The results are presented in Section 3, along with the answers to the research questions. The proposed MS anti-pattern classification framework is introduced and reasoned in Section 4. The discussion about findings and the study result validity threats are provided in Section 5, followed by a conclusion.

2. Methods

In order to investigate the aforementioned questions, we performed a tertiary study of the literature classifying the Secondary Studies (SS) on MS anti-patterns (RQ1), we select (RQ2) and classify them (RQ3) and we investigate detection methods proposed by researchers (RQ4). Finally, we derive a framework to understand potential detection strategies. The overall process is summarized in Fig. 1.

Tertiary studies aim to synthesize results from secondary studies to provide a comprehensive view of a given topic. To accomplish this, we followed the guidelines for carrying out tertiary studies in software engineering defined by Kitchenham et al. (2022). The individual method stages are detailed in the following subsections.

2.1. Search and selection process

The established approach for evidence identification is to construct a search string for scientific indexers that return relevant literature. Furthermore, a follow-up snowballing process can identify literature missed in the search.

Search string

The specific goal of this work is to analyze and categorize identified anti-patterns in the domain of microservices. This itself translates into a basic search string. In addition, the specific research questions could possibly extend the search string design. However, most research questions are implied from the study goal without impact on the search string.

The obvious search terms related to this study's aims are "`microservice`" and "`anti-pattern`" as well as its mutations, and variants (i.e., "`smell`"). When we consider design patterns we use the alias of best practices, which in the inverse leads to poor/bad practice. To broaden the search results we also include the term "`practice`", which targets both bad and best practices. We were also interested in the identification of anti-pattern detection techniques; however, excluding such a term from the search string, which would only further restrict the results, enable us to maintain results relevant to all RQs and apply this aspect in inclusion criteria.

The result of the search string contained the following search terms:

```
((microservice  OR  micro-service  OR  "micro service")
                    AND
 (anti-pattern  OR  "anti pattern"  OR  antipattern
                OR  smell OR practice))
```

To increase the likelihood of finding publications addressing MS anti-patterns, we applied the search string to both title and abstract.

Data sources

We selected the list of relevant bibliographic sources following the suggestions of Kitchenham and Charters (2007) since these sources are recognized as the most representative in the software engineering domain and used in many reviews. The list includes *ACM Digital Library,*[1] *IEEE Xplore Digital Library,*[2] *and Scopus.*[3]

[1] https://dl.acm.org
[2] https://ieeexplore.ieee.org
[3] https://www.scopus.com

Table 1
Inclusion and exclusion criteria.

Criteria	Assessment criteria	Step
Inclusion	Papers that report Literature Reviews on microservice anti-patterns, bad smells, and bad practices	All
	Papers that report Literature Reviews on microservice {anti-pattern, bad smell, or bad practice} detection techniques	All
Exclusion	Papers not fully written in English	T/A[a]
	Papers, not peer-reviewed (e.g. surveys, the proposal of patterns, …)	
	Duplicate papers (only consider the most recent version)	T/A
	Position papers and work plans (i.e., papers that do not report results)	T/A
	Publications where the full paper cannot be located (i.e., if the database user does not have access to the full text of the publication)	T/A
	Publications that do not mention MS anti-patterns and do not fully or partly focus on it	All
	Paper published before Fowler's (Lewis and Fowler, 2014) definition (older than 2014)	All
	Only the latest version of the papers (e.g., journal papers that extend conference papers are excluded if they refer to the same dataset)	All

[a] Title and Abstract.

While excluding gray literature in research is a common practice, it is important to note that gray literature holds an important value in this field and domain. This work only considered secondary studies from peer-reviewed sources; however, gray literature has been already included within the selected secondary studies. For instance, Neri et al. (2020) performed a systematic review of the white and gray literature. Ding and Zhang (2022) performed a comprehensive systematic review of white and gray literature (DZone, StackOverflow, InfoQ, and TeachBeacon). Finally, Ponce et al. (2022) conducted a multivocal review of the existing white and gray literature on the topic. Yet, a comprehensive gray literature review could bring additional anti-patterns not yet mentioned in peer-reviewed literature.

Inclusion and exclusion criteria

For the literature selection, we defined inclusion and exclusion criteria based on our RQs and applied them to the title and abstract (T/A) or to the full text (F), or to both cases (All), as reported in Table 1. This established a consistent selection for the next stage among co-authors.

Search process

The search process in this study involved four steps:

- *Secondary Study Collection:* We run the search string in all the selected bibliographic sources, and we removed all the duplications by title.
- *Applying inclusion and exclusion criteria to title and abstract:* Two different reviewers read the title and abstract to find if the publication meets our inclusion and exclusion criteria. If at least one reviewer was affirmative, we included the publication for the full read.
- *Full read:* Full reading was performed on all the papers included by title and abstract excluding the ones that do not meet the inclusion and exclusion criteria.
- *Snowballing:* After a full read, snowballing was performed. The snowballing process (Wohlin, 2014) considered all the references presented in the retrieved papers and evaluated all the papers referencing the retrieved ones as well as papers recommended by authors as relevant to this topic.

The search and selection process is depicted in Fig. 2. Please note that the results are discussed and provided in Section 3, this section describes the process.

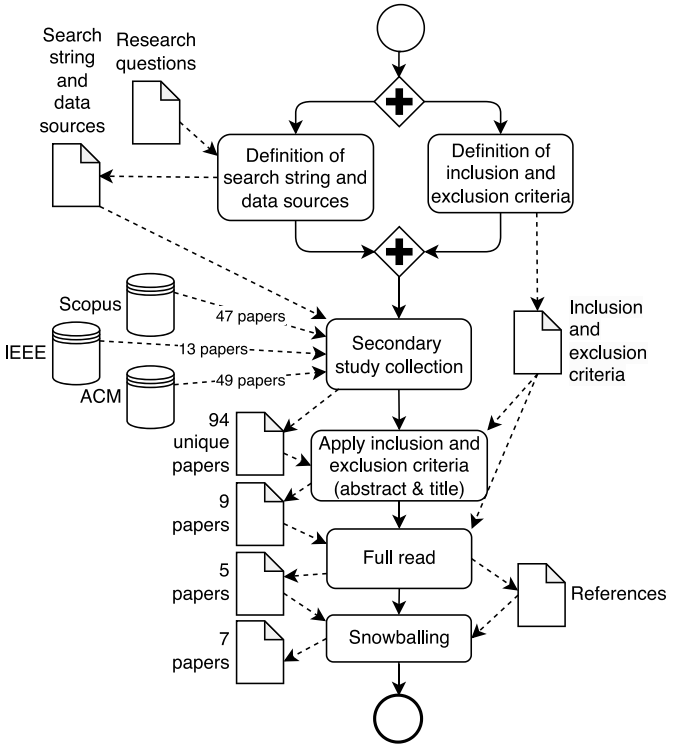

Fig. 2. The applied search process.

2.2. Quality assessment

We evaluated the quality of each selected secondary study by means of the DARE method (Anon, 2007). The DARE method proposes to evaluate the quality based on four quality assessment (QA) questions:

- QA1 Are the review's inclusion and exclusion criteria described and appropriate?
- QA2 Is the literature search likely to have covered all relevant studies?
- QA3 Did the reviewers assess the quality/validity of the included studies?
- QA4 Were the basic data/studies adequately described?

Table 2
Results of search and selection and application of quality assessment criteria.

Attribute	# Details
Anti-patterns	This included the anti-pattern names This included a search for referenced materials (i.e. GitHub, JSON)
Description	This included the anti-pattern descriptions
Aliases	Anti-patterns can be known by multiple names
Classification	If classification was given we would extract it
Detection methods	If detection method was given we would extract them

The DARE quality criteria propose to score the papers according to the following scheme:

- `QA1` Y (yes), the inclusion criteria are explicitly defined in the study, P (Partly), the inclusion criteria are implicit; *N* (no), the inclusion criteria are not defined and cannot be readily inferred.
- `QA2` Y, the authors have either searched 4 or more digital libraries and included additional search strategies or identified and referenced all journals and conference proceedings addressing the topic of interest; P, the authors have searched 3 digital libraries with no extra search strategies, or searched a defined but restricted set of journals and conference proceedings; N, the authors have searched up to 2 digital libraries or an extremely restricted set of journals.
- `QA3` Y, the authors have explicitly defined quality criteria and extracted them from each primary study; P, the research question involves quality issues that are addressed by the study; N, no explicit quality assessment of individual primary studies has been attempted.
- `QA4` Y, Information is presented about each study; P, only summary information about primary studies is presented; N, the results of the individual primary studies are not specified.

We adopted a scoring value of Y = 1, P = 0.5, *N* = 0, or unknown in case the information is not reported.

We conducted the quality assessment in two steps. The quality assessment was performed by two researchers independently. In case of disagreements, a third author intervened to remove the conflict.

2.3. Data extraction

The data extraction considered the identification and collection of anti-patterns from each secondary study that we identified in the selection process and passed our quality assessment criteria. The data extraction included attributes specified in Table 2.

From each included paper, we manually extracted the list of anti-patterns and the other related attributes (if available). In the majority of the cases, the information is available in the section results, in a few cases, we looked in the appendix. In case of the list of anti-patterns is included in an online appendix (e.g. in form of a JSON file on GitHub,[4]) we created a Python script to fetch and parse the data. In case of descriptions, secondary studies were often time too brief and we reached to primary studies for a detailed anti-pattern description.

2.4. Analysis, synthesis methods, and bias assessment

In this Section, we describe the methods adopted to analyze the results. **RQ1** was analyzed by counting the number of selected papers published. As for the identification of the unique anti-patterns (**RQ2**), the authors of this work collaboratively clustered similar anti-patterns and identified alternative names. They also assessed their descriptions to determine relevance to microservices and possibly overlap with other identified anti-patterns. When multiple names were defined for the same pattern, they selected the most frequently adopted name. The classification of the patterns (**RQ3**) was first performed by the authors and then validated by a set of experts. The classification process which we detail in the next two subsections is depicted in Fig. 3.

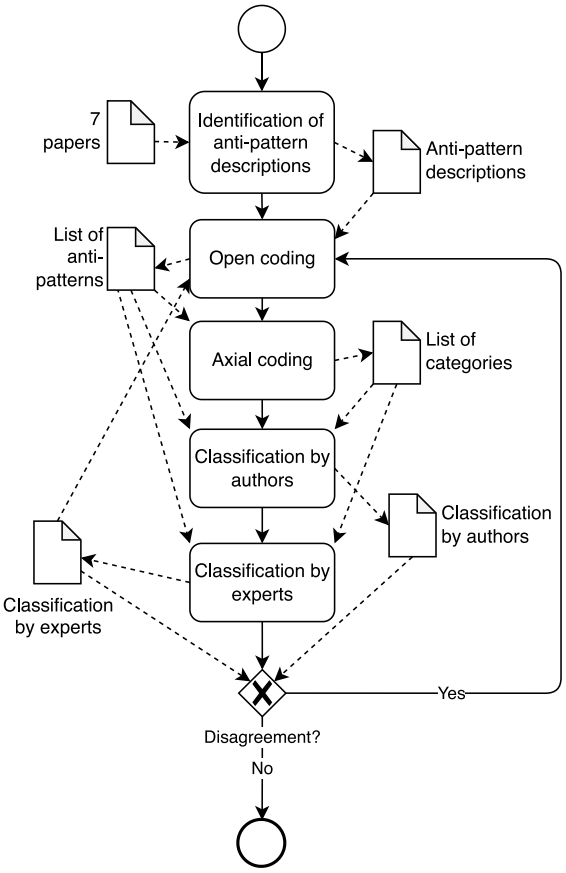

Fig. 3. The applied categorization process.

2.4.1. Anti-patterns coding

The first classification step, executed by the authors of this work, was performed by using open and axial coding methods.

The **open coding** (Strauss and Corbin, 1998; Kendall, 1999) process facilitates an exploratory analysis of the anti-patterns data, allowing for the generation of initial codes pertaining to anti-patterns and categories. This stage was performed in three phases. We first performed a pilot study. In this phase, we randomly selected 10% of the anti-patterns extracted from the secondary studies to establish an initial set of codes (the unique anti-patterns) using open coding. Then we performed a second

[4] https://github.com

pass on the pilot anti-patterns to generate a final list of codes to minimize inconsistencies during the coding process. In the second phase, an initial theory about the relationship between codes was developed based on a pilot study that utilized a previously established set of codes. In the last phase, we organized the anti-patterns into a hierarchical structure based on emerging relationships between concepts. This structured list of codes was then used to code the remaining anti-patterns, with three different coders executing the coding collaboratively during a face-to-face session. Each anti-pattern name, description, proposed category, and detection method were analyzed line by line using the list of codes, with codes being applied if they reflected a concept in an anti-pattern description or name.

The process of **axial coding** (Kendall, 1999) is a qualitative data analysis technique that focuses on establishing relationships between initial codes and categories identified during the open coding process. It involves organizing and structuring the anti-patterns by identifying core categories and exploring the relationships and connections between them.

The use of axial coding supports the robust generation of a catalog of anti-patterns for two main reasons. Firstly, it provides a systematic and structured approach to analyzing the data, ensuring a comprehensive understanding of the classification under study. Identifying relationships and connections between different anti-patterns enables a more nuanced and holistic categorization. This ensures that the catalog captures the complexity and interdependencies within the domain of microservices. Secondly, axial coding helps in identifying common themes and patterns within the data, which may reveal higher-level categories or themes that were not initially apparent. This allows for the development of a more comprehensive and meaningful classification system for anti-patterns in microservices, capturing the underlying principles and characteristics of these patterns.

The performed axial coding process consists of two phases. In the first phase, a comparison was made among the classified anti-patterns to inductively generate relationships between them, such as the relationship between intra-service decomposition and inter-service decomposition. In the second phase, the definitions of all the coded concepts were compared to identify any aliases or overlapping concepts, ensuring clarity and consistency in the catalog.

As a result of this coding process, a proposed comprehensive collection of anti-pattern categories and a recommended classification of the anti-patterns within these chosen categories were derived. Additionally, axial coding allows for the refinement and validation of the categorization through expert input and feedback. By involving external experts in the process, the reliability and credibility of the classification can be enhanced.

2.4.2. Expert validation and pattern classification

To validate the proposed classification, we conducted a survey involving ten external participants. The recruitment process entailed distributing the questionnaire via email to ten individuals from our contact list. All ten practitioners willingly agreed to participate. These participants have a range of 1–4 years of experience in microservices development and research, with an average experience of $\approx$ 2 years. Additionally, all participants have indicated that their latest contribution to a microservice project was within the past 6 months. Moreover, the number of microservices that participants have worked with ranges from 15 to 50, highlighting the diverse experience and exposure they have had in the field of microservices development. The information about the participants can be found in the dataset[5] mentioned in Section 3.

The survey requested the participants to categorize the anti-patterns based on the categories obtained from our coding process. The questionnaire, distributed via email, consisted of a list of anti-patterns, their descriptions, and a dropdown menu allowing participants to classify the anti-patterns into different categories. Additionally, we provided the participants with descriptions of the categories to aid them in accurately classifying the anti-patterns.

The study has been designed as a between-subject method, with each participant randomly assigned a subset of identified anti-patterns to classify individually. After each round, we collected and analyzed the data to refine the classification process, identifying possible inconsistencies with our original classification.

The study involved two rounds of this survey, during which the authors gathered input from participants on their classification of anti-patterns. After each round, the authors conducted further open and axial coding (as described in Section 2.4.1) to refine category definitions and anti-pattern classification. This iterative process of coding and expert validation (Section 2.4.2) continued until there were no more disagreements.

Table 3
Results of search and selection.

Activity	# Papers
Secondary study collection	109
Remove duplicates	94 (−15)
Inclusion and exclusion criteria to title and abstract	9 (−85)
Full read	5 (−4)
Snowballing	7 (+2)
Quality assessment	7 (0)
Selected secondary studies	**7**

2.5. Executing the study

This subsection presents the incremental results of our search process that followed the previously described methods. It also elaborates on the data extraction and synthesis.

2.5.1. Search process results

The search process led to reductions or additions in the candidate paper set in each considered phase. The results we received from particular bibliographic sources (IEEE/ACM/Scopus) along specified phases. Fig. 2 highlighted the search process and it also contains the particular paper counts throughout the progress of the process, which we explain here. In addition, the summary of the search process in Table 3 may help the reader with individual stages. The complete details of the search process stages and their outcomes are also available at shared dataset.[5]

We first *executed the search query* on the selected bibliographic sources. This search yielded a total of 109 papers, including 47 papers from Scopus, 13 papers from IEEE, and 49 papers from ACM. To ensure the integrity of the dataset, we proceeded to eliminate duplicate papers based on their titles, resulting in a final selection of 94 unique papers for further processing.

The application of *inclusion and exclusion criteria* after reading the abstracts and titles, resulted in a reduction of the selected papers to 9 secondary studies.

The *full read* of all filtered papers resulted in a total of only five papers that met the inclusion criteria. This stage excluded four publications by Bogner et al. (2020), Guo and Wu (2021), Osses et al. (2018), and de de Oliveira Rosa et al. (2020). These studies were excluded because they did not provide a list of MS anti-patterns as required for our research.

Next, we performed *snowballing* process (Wohlin, 2014), which allowed us to add two more secondary studies by Sabir et al. (2019) and Ponce et al. (2022).

[5] https://zenodo.org/record/7993516

Table 4
Results of extraction and classification process.

Extraction rounds	# Anti-patterns
Initially collection	287
Basic name deduplication:	203 - (−84 name duplicates)
Classification rounds	
First round (open & axial coding)	77 - (microservice-only & names/alias merge)
Second round (expert validation, open & axial coding)	63 - (merge: 9, delete: 5, misclassified: 32)
Third round (expert validation, open & axial coding)	58 - (merge: 5, delete: 0, misclassified: 0)
Total anti-patterns	**58**

Finally, we applied the quality assessment to the secondary studies, which did not exclude any study. As a result, we selected seven secondary studies to perform data extraction and synthesis on.

2.5.2. Data extraction and synthesis results

As for the *data extraction and synthesis* process, the authors extracted the anti-patterns from the seven secondary studies, as well as their descriptions from the original primary studies defining them. The extraction and classification process rounds and their results are summarized in Table 4 to help the reader with individual stages.

Initially, we collected 287 MS anti-patterns, of which 84 were name duplicates. After removing the duplicates, we obtained 203 anti-patterns. We made sticky note-style cards and clustered them with information available in the secondary studies (e.g., anti-pattern name, source paper, aliases, and categorization proposed by the original papers).

The classification of the anti-patterns into categories (RQ3) was performed in three compound rounds; three open/axial coding rounds performed by the authors (Section 2.4.1), and two rounds of experts validation (Section 2.4.2):

The *first round* started with a collaborative study, wherein the authors engaged in the open coding process. During this phase, they analyzed the anti-patterns, addressing any *duplicate entries based on descriptions* (i.e., name alias) and excluding those that were *not relevant to microservices*. The result was an initial set of 77 anti-patterns, which we used to perform the first validation with the experts. We randomly distributed ≈ 15 anti-patterns among the ten participants (each pair had the same assignment), asking them to classify the anti-patterns based on their descriptions. We ensured that the same subset of anti-patterns was assigned to participants with varying levels of experience. By doing so, we aimed to gauge the common understanding of both novice and experienced participants regarding the classification of anti-patterns.

The *second round* used the validation feedback from the expert to conduct a second round of open and axial coding to adjust our initial classification and anti-pattern descriptions. It ended with five deleted, nine merged, and 32 unconsented and misclassified anti-patterns. We manipulated our classifications accordingly. Next, we conducted a second round of the survey with the expert to get validation feedback on the 32 misclassified anti-patterns that do not meet the consensus.

The *third round* of the open/axial coding resulted in five more merged anti-patterns; therefore, the final set of classified anti-patterns contains 58 items, each with a clear description and assigned category.

The final catalog comprises 58 anti-patterns, which have been effectively categorized into five distinct categories according to their relationships and expert validations. These categorized anti-patterns, along with the outcomes of each phase of the study and the survey data, have been published and are accessible in the dataset[5].

3. Results

In this Section, we answer our research questions.

3.1. RQ1: What secondary studies have been published in the area of MS anti-patterns?

Seven secondary studies on MS anti-patterns were published from the introduction of microservices. A total of 340 primary studies were included in these seven selected secondary studies. Table 5 reports the list of selected studies. Moreover, they extract and categorize multiple anti-patterns as detailed in Table 6.

The first two studies were published in 2019 by Sabir et al. (2019) and Bogner et al. (2019b). Sabir et al. listed 49 service-oriented and 56 object-oriented anti-patterns, while Bogner et al. provided a list of 23 anti-patterns. In our study, we excluded the object-oriented anti-patterns and the first-time mentioned service-oriented ones, including only the 16 anti-patterns that are service-oriented and applicable to microservices. We also discarded two unrelated anti-patterns from the business category listed by Bogner et al.

Neri et al. (2020) and Tighilt et al. (2020) both published their studies in 2020, considering 55 and 27 primary studies, respectively. Neri et al. listed seven anti-patterns, all of which were included in our study. Tighilt et al. identified 16 anti-patterns, and we discarded one of them in our analysis.

Mumtaz et al. (2021) identified the most primary studies, with 85 studies found and 103 anti-patterns listed. However, 65 of these anti-patterns were not identified as service-related, so we included only 39 of them in our study as microservice-related. This was the only study that appeared in 2021.

Ding and Zhang (2022) and Ponce et al. (2022) reported 22 and 10 anti-patterns, respectively, in 2022. They extracted the anti-patterns from 23 and 58 primary sources, respectively, and we included all of these anti-patterns in our study as microservice-related.

3.2. RQ2: Which distinct MS anti-patterns have been identified in the secondary studies?

In our study to create a comprehensive catalog of microservice-related anti-patterns, we took steps to ensure accuracy and minimize redundancy. This included removing duplicates, consolidating similar anti-patterns, and introducing aliases for those identified in the seven secondary studies. In total, we identified 58 distinct anti-patterns related to microservices.

We found that some of these anti-patterns were frequently mentioned in the seven secondary studies we examined. Table 6 provides details on the number of anti-patterns extracted from each study and their respective categories. The results revealed significant overlap between the selected anti-patterns from multiple sources, as shown in Table 7. The highest number of common anti-patterns between any two sources was 15, which occurred between Mumtaz et al. (2021) and both Sabir et al.

Table 5
The selected secondary studies.

Ref.	Title	Year	#Primary studies included
Sabir et al. (2019)	A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems	2019	78
Bogner et al. (2019b)	Towards a collaborative repository for the documentation of service-based antipatterns and bad smells	2019	14
Neri et al. (2020)	Design principles, architectural smells and refactorings for microservices: a multivocal review	2020	55
Tighilt et al. (2020)	On the study of microservices antipatterns: A catalog proposal	2020	27
Mumtaz et al. (2021)	A systematic mapping study on architectural smells detection	2021	85
Ding and Zhang (2022)	How can we cope with the impact of microservice architecture smells?	2022	23
Ponce et al. (2022)	Smells and refactorings for microservices security: a multivocal literature review	2022	58

Table 6
The secondary studies categories.

Ref.	Categorization technique	Categories	#Antipatterns reported	#Antipatterns included[a]	Total
Sabir et al. (2019)	Classifying based on two types of smells, namely object-oriented and service-oriented smells. Furthermore, they divided the service-oriented smells into two groups based on whether they were frequently reported in services literature or whether they were mentioned for the first time.	1- Service-oriented[b] 2- Object-oriented	49 56	16 0	16
Sabir et al. (2019)	Categorizing items based on the effects they have, such as whether they are related to the design, the way the different components of the application interact with each other, or how users interact within the business context.	1- Architecture 2- Application 3- Business	16 4 3	16 4 1	21
Sabir et al. (2019)	The classification of the smells is based on how they violate the principles of microservices architecture.	1- Deployability 2- Scalability 3- Decentralization 4- Isolation of failures	1 2 3 1	1 2 3 1	7
Sabir et al. (2019)	Categorizing the smells according to the stages of development in a system based on microservices.	1- Design 2- Implementation 3- Deployment 4- Monitoring	4 2 7 3	4 2 7 2	15
Sabir et al. (2019)	This classification technique involves categorizing the smells based on the same order they are presented in the primary papers.	1- Service 2- Performance 3- Dependency 4- Package 5- MVC 6- Component 7- Other smells	38 10 6 8 14 8 19	33 3 1 0 1 0 1	39
Sabir et al. (2019)	Categorizing according to various characteristics and attributes specific to the architecture of microservices considering the development lifecycle.	1- Design 2- Deployment 3- Monitor & Log 4- Communication 5- Team & Tool	6 3 5 6 3	6 3 4 6 3	22
Sabir et al. (2019)	This classification technique groups security smells based on the security properties they relate to, using the ISO/IEC 25010 standard as a reference. The technique focuses on three security properties: confidentiality, integrity, and authenticity.	1- Security	10	10	10

[a] The numbers include merged anti-patterns.
[b] It consists of 19 repeated anti-patterns and 30 first time mentioned. The process filtered out many of that are mentioned for the first time.

Table 7
Number of anti-patterns in secondary studies that appear in this work and their overlaps.

	Sabir et al. (2019) (16)	Bogner et al. (2019b) (21)	Neri et al. (2020) (7)	Tighilt et al. (2020) (15)	Mumtaz et al. (2021) (39)	Ding and Zhang (2022) (22)	Ponce et al. (2022) (10)
Sabir et al. (2019) (16)	–	10	1	3	15	4	0
Bogner et al. (2019b) (21)	10	–	3	7	15	9	0
Neri et al. (2020) (7)	1	3	–	4	5	6	0
Tighilt et al. (2020) (15)	3	7	4	–	8	12	0
Mumtaz et al. (2021) (39)	15	15	5	8	–	12	0
Ding and Zhang (2022) (22)	4	9	6	12	12	–	0
Ponce et al. (2022) (10)	0	0	0	0	0	0	–

The numbers in parentheses indicate the total count of anti-patterns for each study that appear in this work.
The numbers in each cell represent the count of anti-patterns that are common between the two studies.

Table 8
Intra-service design anti-patterns.

SCat	Index	Anti-pattern name	Defined in	Referenced by
Granularity	1	Nano-service (**AKA**, Nano microservice, Tiny/nano/fine-grained service)	2003, 2009, 2012–2015, 2017, 2020, 2021	Sabir et al. (2019), Bogner et al. (2019b), Tighilt et al. (2020), Mumtaz et al. (2021), Ding and Zhang (2022)
	2	Mega service (**AKA**, Mega microservice, Blob or god object/component, God object web service, Multi-service, Bloated service)	2003 −2005, 2010, 2013–2018, 2020	Sabir et al. (2019), Bogner et al. (2019b), Tighilt et al. (2020), Mumtaz et al. (2021), Ding and Zhang (2022)
Service interface	3	CRUDY service (**AKA**, Crudy interface, Crudy URI)	2014, 2015, 2017, 2019	Sabir et al. (2019), Mumtaz et al. (2021)
	4	Nobody home (**AKA**, Unused interface)	2014	Mumtaz et al. (2021)
	5	Data service (**AKA**, Data web service)	2014–2015, 2017	Sabir et al. (2019), Mumtaz et al. (2021)
	6	No API-versioning (**AKA**, API versioning)	2018, 2020, 2021	Bogner et al. (2019b), Tighilt et al. (2020), Mumtaz et al. (2021), Ding and Zhang (2022)
Cohesion	7	Whatever types (**AKA**, Ignoring MIME types, Forgetting hypermedia)	2007, 2011, 2013, 2014, 2019	Sabir et al. (2019), Mumtaz et al. (2021)
	8	Low cohesive operation	2007, 2011, 2013–2015, 2017	Sabir et al. (2019), Bogner et al. (2019b)
	9	Ambiguous service (**AKA**, Ambiguous name, Ambiguous interface)	2009, 2011, 2013–2015, 2017	Sabir et al. (2019), Bogner et al. (2019b), Mumtaz et al. (2021)

(2019) and Bogner et al. (2019b). However, there were no common anti-patterns mentioned in Ponce et al. (2022) with any of the other sources.

Moreover, Fig. 4 depicts the intersection of anti-patterns between the most four sources containing anti-patterns (Sabir et al., 2019, Bogner et al., 2019b, Mumtaz et al., 2021, and Ding and Zhang, 2022) out of the seven sources. It shows that these four sources share three common anti-patterns: Wobbly service interactions, Nano service, and Mega service. Moreover, Sabir et al. (2019), Bogner et al. (2019b), and Mumtaz et al. (2021) contain nine common anti-patterns, while Bogner et al. (2019b), Mumtaz et al. (2021), and Ding and Zhang (2022) contain eight common anti-patterns. Additionally, it is evident that almost all of the anti-patterns that were included in Sabir et al. (2019) are already encompassed in Mumtaz et al. (2021), except for a single anti-pattern, which is Scattered parasitic functionality.

All of the anti-patterns that were identified can be found in Table 13, and they are categorized in Tables 8–12.

3.3. RQ3: How are MS anti-patterns classified in the secondary studies?

The secondary studies consider different techniques for the categorization of the anti-patterns, as summarized in Table 6. Sabir et al. (2019) adopted a classification perspective combining two categories of *object-oriented* and *service-oriented* paradigms to categorize anti-patterns. They classified object-oriented related anti-patterns into five subcategories: anti-patterns, architectural smell, code and design, code smell, and design smell. However, some of these categories were not applicable to the service-oriented context, such as the code and design category. Service-related anti-patterns were classified into those first mentioned in primary studies and those mentioned multiple times.

Bogner et al. (2019b) considered three categories of *architecture*, *application*, and *business*. The *architecture* category impacts architecture and design-related aspects of the system. The *application* category impact interactions of application components

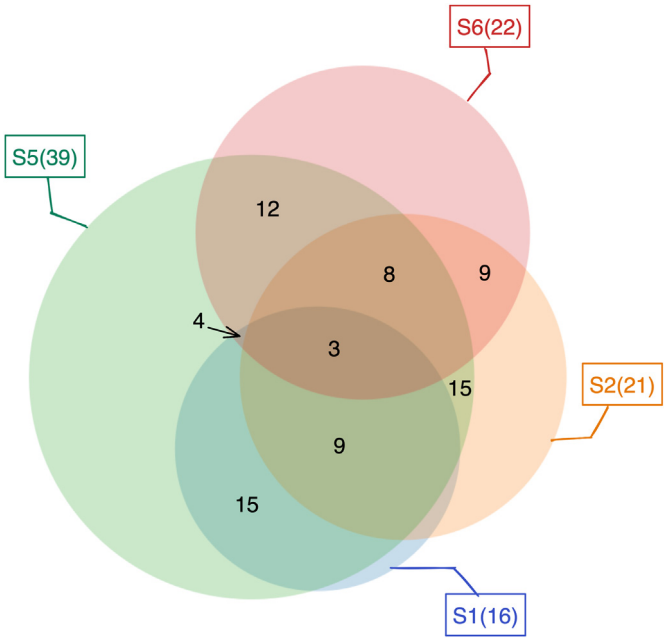

Fig. 4. Intersection of anti-patterns between sources (a condensed depiction of the top four sources due to space limitations).

and application-level functionality. Finally, the *business* category coped with interactions of users, businesses, and data.

Neri et al. (2020) utilized a 4+1 architectural viewpoint scheme to identify anti-patterns related to the dynamic aspects of interactions and their violations. Their focus was on four categories: *deployability* for ensuring independent deployability of microservices, *scalability* to address horizontal scalability, *decentralization* to overcome bottlenecks, and *isolation of failures* to monitor system resilience.

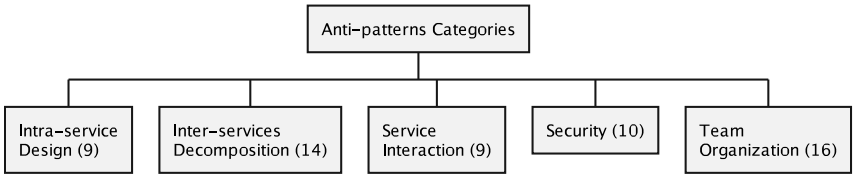

Fig. 5. The proposed anti-patterns categories.

Tighilt et al. (2020) organized anti-patterns into four categories with regard to the development cycle. *Design* category coping with the specification of the architectural design. *Implementation* category about how the microservices are implemented. *Deployment* category with packaging and deployment of micro-service-based systems. Finally, a *monitoring* category related to the monitoring of microservices, their behavior, and changes.

Mumtaz et al. (2021) categorized detection approaches and also included anti-patterns categories comprised of seven categories: *service*, *performance*, *dependency*, *package*, *MVC*, *component* and *other smells*. The categorization includes anti-patterns not relevant to microservices (i.e., package, MVC, component). The granularity of the service category is not further divided. It combines single service granularity, communication, architecture, or organization.

Ding and Zhang (2022) utilized five categories to classify anti-patterns. These categories are related to different aspects of microservice architecture and also include the development cycle by comparing violated principles. The categories used are *design*, *deployment*, *monitor & log*, *communication*, and *team & tool*.

Ponce et al. (2022) considered the security aspect of microservices, an exclusive theme with respect to other studies. They considered tagging anti-patterns with *confidentiality*, *integrity*, and *authenticity*. Confidentiality is the degree to which a system ensures that data are accessible to those authorized users. Integrity is the degree to which a system prevents unauthorized data modification. Authenticity is the degree to which the identity of a subject or resource can be proved to be the one claimed.

3.3.1. Classified categories of anti-patterns

After examining the resulting secondary studies, it was found that each study categorized anti-patterns differently, some from a development cycle perspective and others from a scalability and deployability perspective.

We have considered anti-patterns assigned to categories from the secondary studies when analyzing them. However, it is obvious that they lack greater overlap that could be generalized across the studies. As a result, it was deemed necessary to adopt a consistent perspective related to Microservices Architecture.

As described in Section 2.4.1, axial coding resulted in an inductive classification of anti-patterns according to their relationships. To provide a higher-level categorization, we considered the *basic* perspective of software architecture. Software architecture contains software elements of certain properties and relations among them, again with certain properties. At the same time, architected systems have certain non-functional aspects. Furthermore, to build a system with certain architecture, there is a process of doing so, including development and operations.

The microservice architecture emphasizes the creation of small, independent services that collaborate to create a complete application. Thus to apply our anti-pattern categorization perspective to the microservice context, we must recognize that elements are services with specific structures (1), the decentralization perspective allows us to (2) decompose the problem into multiple microservices, the connection is realized through the service communication (3). The non-functional perspective contains various quality aspects, such as security (4). Finally, to build and operate such systems, we need to organize teams (5). Therefore, we translate these perspectives into the following five categories, as depicted in Fig. 5 and subsequently we assigned the classes of anti-patterns identified through axial coding to these five high-level categories.

- Intra-service design category: The intra-service design category considers a single service component and its design. We have further subdivided the anti-patterns into the service interface, granularity, and functional cohesion perspectives as subcategories. This category has nine items highlighted in Fig. 6 and detailed in Table 8. This table lists the subcategories followed by an index that we used to connect anti-patterns and anti-pattern names; it also gives their aliases (also known as - AKA) and details when they were defined and which secondary studies they referenced. The indices connect the anti-patterns with a comprehensive description provided in Appendix. In addition, the appendix also provides references to primary studies.

With regard to the sub-categories: the granularity considers the sizing of services (*nano-service, mega service*); the interfaces point to problems apparent from service interfaces (*CRUDY service, nobody home, data service, no API-versioning*), and the cohesion then considers clarity and comprehensibility of the service or service properties (*whatever types, low cohesive operation, ambiguous service*).

- Inter-service decomposition category: The inter-service decomposition considers the system's structural division involving two or more microservices. This can consider service integration, decomposition approaches resulting in improper modularity, and service relationships. This category has 14 items highlighted in Fig. 7 and detailed in Table 9 with the structure introduced previously referencing the appendix for full description.

Besides anti-patterns which we classified generally fitting this category (*transactional integration, co-change coupling, duplicated service, and microservice greedy*), we also considered subcategories. One of the specific aspects we identified across anti-patterns in this category is a topology, where we consider how the decomposition predetermines microservice topological connections, which are unintended (*service chain, hub-like dependency, cyclic dependency*). Topology often time imply communication as well, but the major perspective is decomposition rather than communication. Another perspective we clustered in this category is violated modularity across services (*chatty service, shared persistency, sand pile, shared libraries, wrong cuts, knot service, scattered parasitic functionality*). Modularity can be violated by various means, including unintended sharing of components, coupled services, their improper scope, or too fine-grained decomposition with remaining dependencies.

- Service interaction category: The service interaction category looks into how services interact. Often times this might be predetermined by the topology given by service decomposition; however, the violation here involves interaction aspects. We identified anti-patterns that indicate improper communication routes or disable system scalability or harm resilience.

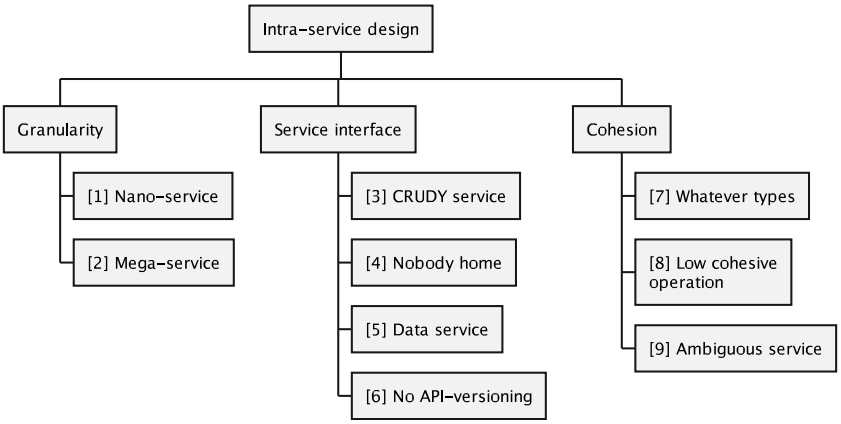

Fig. 6. Intra-service design anti-patterns (9 anti-patterns).

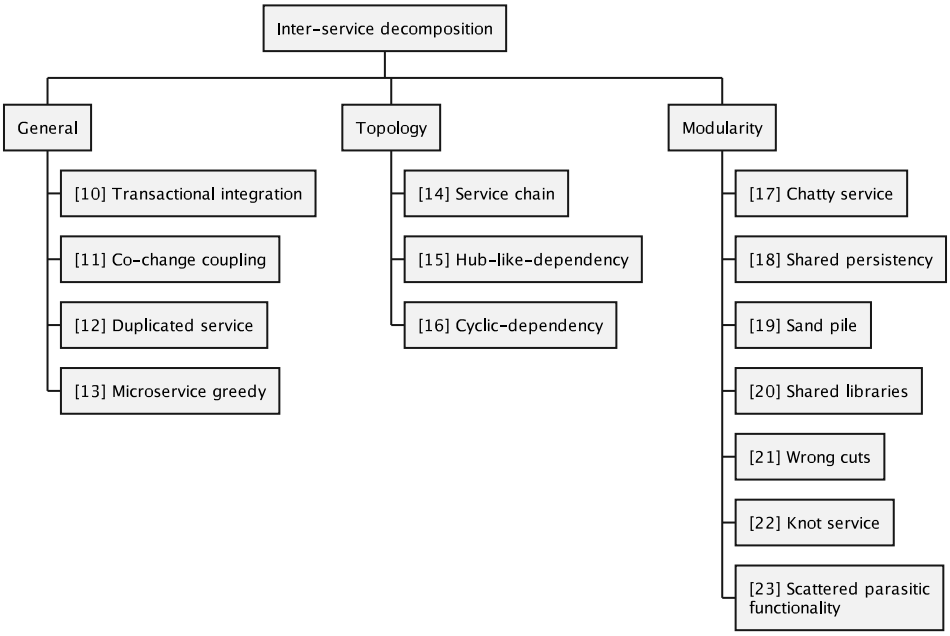

Fig. 7. Inter-service decomposition anti-patterns (14 anti-patterns).

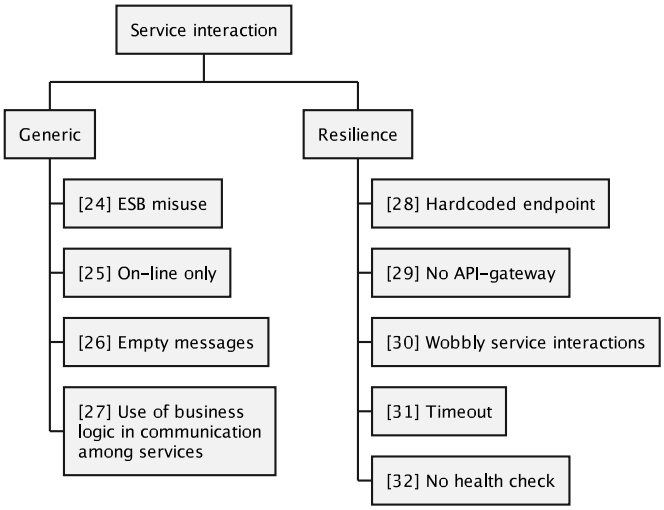

Fig. 8. Service Interaction anti-patterns (9 anti-patterns).

This category has nine items highlighted in Fig. 8 and detailed in Table 10 with the established structure and references to the appendix for full descriptions.

Besides anti-patterns classified at the top category level (*ESB misuse, on-line only, empty messages, use of business logic in communication among services*), we also clustered these specific to resilience. Resilience patterns (Carnell and Sánchez, 2021) deal with good citizen services following *circuit breaker*, *bulkhead*, *fallback*, *rate limiter*, *timeouts*, etc., which promote resilience. Anti-patterns typically violate resilience patterns and typically hide under *wobbly service interaction*. Another perspective here is related to missing communication checkpoints, indirection, or lack of health checks with timeouts (*hardcoded endpoint, no API gateway, wobbly service interactions, timeout, no health check*). While one may assume health checks might be unrelated, they are strongly dynamic and drive *circuit breaker* pattern (i.e., Spring hystrix Carnell and Sánchez, 2021).

- Security category: The security category consists of violations of three basic security aspects authentication, authorization, and encryption. Authentication is a process or action of verifying the identity of a user or process. Authorization is a process to determine whether a given user profile or identity is allowed to

Table 9
Inter-services decomposition anti-patterns.

SCat	Index	Anti-pattern name	Defined in	Referenced by
	10	Transactional integration	2012	Bogner et al. (2019b)
	11	Co-change coupling	2018	Mumtaz et al. (2021)
	12	Duplicated services (**AKA**, Nothing new)	2006, 2009, 2012–2015, 2017	Sabir et al. (2019), Bogner et al. (2019b), Mumtaz et al. (2021)
	13	Microservice greedy	2003, 2018	Mumtaz et al. (2021)
Topology	14	Service chain (**AKA**, Pipe and filter, Message chain)	2006, 2013, 2014	Sabir et al. (2019), Bogner et al. (2019b), Mumtaz et al. (2021)
	15	Hub-like dependency	2019	Mumtaz et al. (2021)
	16	Cyclic dependency (**AKA**, Cyclic between namespaces)	2018–2020	Bogner et al. (2019b), Tighilt et al. (2020), Ding and Zhang (2022)
	17	Chatty service (**AKA**, Empty semi-trucks, Circuitous treasure hunt)	2013–2017	Sabir et al. (2019), Bogner et al. (2019b), Mumtaz et al. (2021)
Modularity	18	Shared persistency	2014, 2016–2021	Bogner et al. (2019b), Neri et al. (2020), Tighilt et al. (2020), Mumtaz et al. (2021), Ding and Zhang (2022)
	19	Sand pile	2007, 2013–2015	Sabir et al. (2019), Mumtaz et al. (2021)
	20	Shared libraries (**AKA**, Shared dependencies)	2018–2021	Sabir et al. (2019), Tighilt et al. (2020), Mumtaz et al. (2021), Ding and Zhang (2022)
	21	Wrong cuts	2018–2020	Bogner et al. (2019b), Tighilt et al. (2020), Mumtaz et al. (2021), Ding and Zhang (2022)
	22	Knot service	2012–2015	Sabir et al. (2019), Bogner et al. (2019b), Mumtaz et al. (2021)
	23	Scattered parasitic functionality (**AKA**, Stove pipe service)	2009, 2014	Sabir et al. (2019), Bogner et al. (2019b)

access a system or perform a specific action. Finally, encryption is the process of converting human-readable plaintext to incomprehensible text, also known as ciphertext. Anti-patterns in this category violate these processes or make them hard to manage, weak or vulnerable. This category has ten items highlighted in Fig. 9 and detailed in Table 11 with the established structure and references to the appendix for full descriptions.

The three sub-categories of authentication, authorization, and encryption are rather exclusive. While some security experts might see encryption as part of authorization, we clustered them independently. The authentication subcategory (*unauthenticated traffic, multiple user authentication*) considered system violations related to this aspect and did not consider centered authority which might lead to ambiguities. The authorization subcategory (*publicly accessible microservices, unnecessary privileges to microservices, insufficient access control, centralized authorization*) is considered granularity or lack of privileges and access control. It is important to note that microservices should honor bounded context, and they have exclusive knowledge about a part of the domain, which leads to their responsibility to determine fine-grained authorization that is decentralized in the system; otherwise, the centralized authorization would violate the decentralized and encapsulated knowledge across individual microservices. The encryption subcategory (*non-secured service-to-service communications, non-encrypted data exposure, own crypto code, hardcoded secrets*) looked at violations in secure communication, unintended data exposure, weak cryptography, and secrets in plain text.

- Team organization category: The team organization category is different from other categories as it moves away from the architectural perspective that considers structure and dynamic system aspects and rather copes with development teams' decisions for implementation techniques, organization of migration strategies, and operations, which can further contain monitoring. This category considers the violation of DevOps practices.

This category is the largest, with 16 items highlighted in Fig. 10 and detailed in Table 12 with the continued structure and references to the appendix for full descriptions.

The anti-patterns are classified into development and operation subcategories. The development subcategory considers practices and standards adopted by development teams (*shiny nickel, golden hammer, lack of communication standards among microservices, too many standards*), or common planning (*inadequate techniques support, no legacy*), or organization pitfalls (*single-layer teams*). We also considered a sub-category related to system migration (*data-driven migration, big bang*).

The operation subcategory considers violations of DevOps practices related the continuous integration, deployment, and configuration management in general (*multiple service instances*

Table 10
Service interaction anti-patterns.

SCat	Index	Anti-pattern name	Defined in	Referenced by
	24	ESB misuse (**AKA**, ESB usage)	2014, 2016–2018, 2020	Neri et al. (2020), Mumtaz et al. (2021), Ding and Zhang (2022)
	25	On-line only (No Batch Systems)	2007	Bogner et al. (2019b)
	26	Empty messages	2007, 2011, 2013, 2014	Neri et al. (2020)
	27	Use of business logic in communication among services	2021, 2019	Ding and Zhang (2022)
Resilience	28	Hardcoded endpoint (**AKA**, Endpoint-based service interactions)	2006, 2018, 2019, 2020, 2021	Bogner et al. (2019b), Neri et al. (2020), Tighilt et al. (2020), Mumtaz et al. (2021), Ding and Zhang (2022)
	29	No API-gateway	2014–2021	Neri et al. (2020), Tighilt et al. (2020), Mumtaz et al. (2021), Ding and Zhang (2022)
	30	Wobbly service interactions (**AKA**, Bottleneck service, Traffic jam, Ramp)	2014–2019, 2021	Sabir et al. (2019), Bogner et al. (2019b), Mumtaz et al. (2021), Neri et al. (2020), Ding and Zhang (2022)
	31	Timeout	2021	Tighilt et al. (2020), Ding and Zhang (2022)
	32	No health check (for automated reasoning)	2013, 2014	Sabir et al. (2019), Mumtaz et al. (2021)

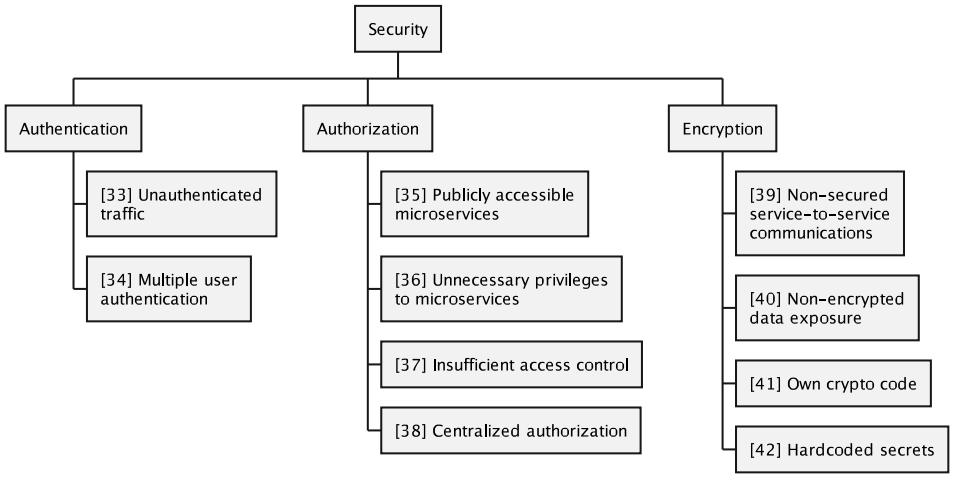

Fig. 9. Security anti-patterns (10 anti-patterns).

Table 11
Security anti-patterns.

SCat	Index	Anti-pattern name	Defined in	Referenced by
Authentication	33	Unauthenticated traffic	2015, 2016, 2018, 2019	Ponce et al. (2022)
	34	Multiple user authentication	2016–2020	Ponce et al. (2022)
Authorization	35	Publicly accessible microservices	2017–2021	Ponce et al. (2022)
	36	Unnecessary privileges to microservices	2017–2020	Ponce et al. (2022)
	37	Insufficient access control	2015–2020	Ponce et al. (2022)
	38	Centralized authorization	2015–2021	Ponce et al. (2022)
Encryption	39	Non-secured service-to-service communications	2015–2020	Ponce et al. (2022)
	40	Non-encrypted data exposure	2015–2020	Ponce et al. (2022)
	41	Own crypto code	2016–2020	Ponce et al. (2022)
	42	Hardcoded secrets	2016–2020	Ponce et al. (2022)

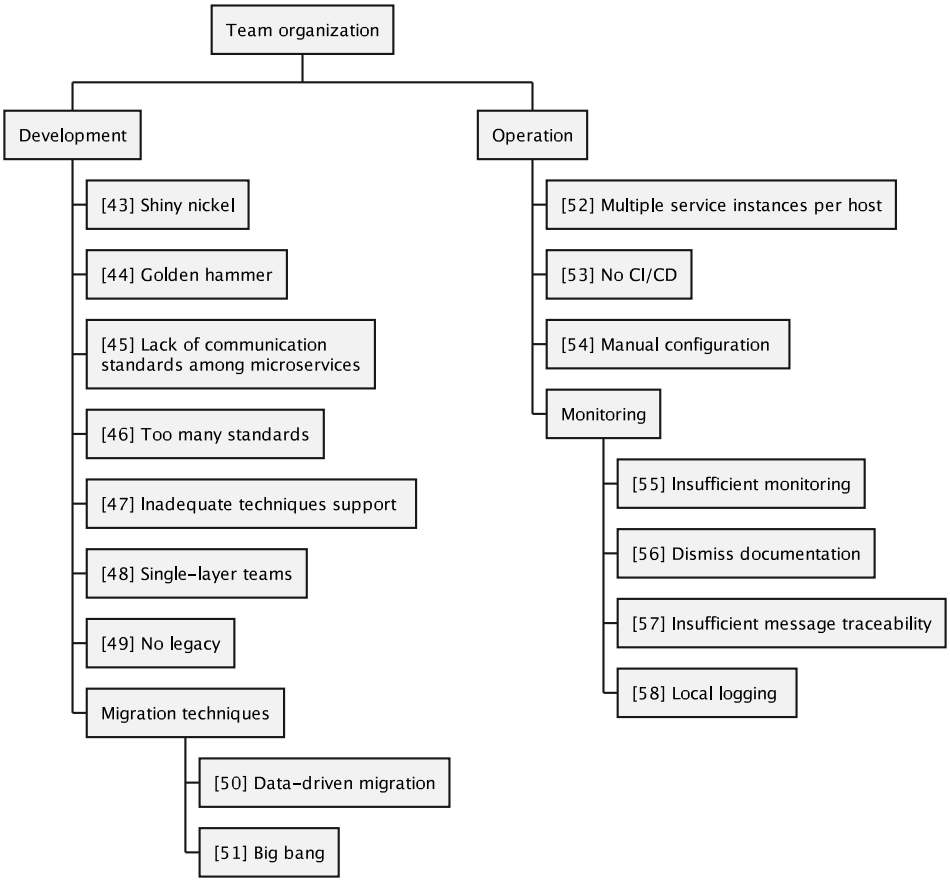

Fig. 10. Team Organization anti-patterns (16 anti-patterns).

per host, no CI/CD, manual configuration). We also clustered another subcategory related to monitoring related to tracing, logging, and documentation (*insufficient monitoring, dismiss documentation, insufficient message traceability, local logging*)

3.4. RQ4: How can we automatically detect anti-patterns, and what techniques could be used?

The next logical question after we catalog microservice anti-patterns is how we can detect them. The most basic approach one could think of is through source code and the development process review. However, it is not very practical given the scale of microservice systems. We could also involve testing to get answers to some concerns. However, even then, we should look for a more efficient approach. Microservices evolve decentrally, and to provide robust quality assurance upon each update, we should check whether the system did not degrade with an unintentional anti-pattern.

Sabir et al. (2019) mentioned *static and dynamic code analysis* approaches. However, they did so on the level of object-oriented and service-oriented paradigms. They identified 70 primary studies related to static analysis and eight primary studies related to dynamic analysis. They further divided the static analysis into six subcategories.

The *behavioral source code analysis* (17 studies) examines the program behavior without code execution. This analysis uses various source code metrics to check program behavior (i.e., cohesion, coupling, depth of inheritance, and lines of code). They highlight that smell detection is not possible without a system intermediary representation. Typically, this analysis is based on the description of flaws or rules. This is also applicable to version control histories. The intermediate representation can take the form of a metamodel or an ontology.

The *empirical source code analysis* (23 studies) considers fetching information by using already established tools. This analysis considers both repository history mining and code parsing.

The *algorithm-based source code analysis* (16 studies) considers multiple detection techniques and may involve machine learning, image processing, or genetic algorithms; it is quite common to set threshold when smells apply. This approach noted multiple codebase repository detections.

The *methodology-based source code analysis* (9 studies) uses an existing methodology in an alternative manner. It sits between the previous two approaches with existing tools and algorithms.

Finally, the *linguistic source code analysis* (5 studies) considers the linguistic quality assessment of wrong naming conventions in class names, methods, etc. This direction spans to natural-language processing discipline.

Sabir et al. (2019) noted that dynamic code analysis considered execution states under real execution scenarios. It combines domain-specific languages with algorithms to perform over service interfaces.

In summary, Sabir et al. found out that information extraction considers two perspectives – static and dynamic – and highlighted the need for intermediate representation that can be used

Table 12
Team organization anti-patterns.

SCat	SSCat	Index	Anti-pattern name	Defined in	Referenced by
Development		43	Shiny nickel (**AKA**, Silver bullet, Focus on latest technologies)	2006. 2018	Bogner et al. (2019b), Mumtaz et al. (2021)
		44	Golden hammer (**AKA**, Same old way)	2003, 2018	Bogner et al. (2019b), Mumtaz et al. (2021), Ding and Zhang (2022)
		45	Lack of communication standards among microservices	2019, 2021	Ding and Zhang (2022)
		46	Too many standards	2018–2021	Mumtaz et al. (2021), Ding and Zhang (2022)
		47	Inadequate techniques support	2018, 2020, 2021	Ding and Zhang (2022)
		48	Single layer team	2014, 2016–2018, 2020	Neri et al. (2020), Ding and Zhang (2022)
		49	No legacy (**AKA**, Everything must be new)	2007, 2018	Mumtaz et al. (2021)
	Migration	50	Data-driven migration	2016	Bogner et al. (2019b)
		51	Big bang	2009	Mumtaz et al. (2021)
Operation		52	Multiple service instances per host (**AKA**, Multiple services in one container)	2015–2018	Neri et al. (2020), Tighilt et al. (2020)
		53	No CI/CD	2020	Tighilt et al. (2020)
		54	Manual configuration	2021	Tighilt et al. (2020), Ding and Zhang (2022)
	Monitoring	55	Insufficient monitoring	2020–2021	Tighilt et al. (2020)
		56	Dismiss documentation	2018	Ding and Zhang (2022)
		57	Insufficient message traceability	2021	Ding and Zhang (2022)
		58	Local logging	2020–2021	Tighilt et al. (2020), Ding and Zhang (2022)

when applying roles or algorithms to detect anti-patterns. However, many researchers depend on existing tools which presents limitations for microservices given their distributed nature.

Mumtaz et al. (2021) proposed a classification of the detection methods for MS anti-patterns. They defined nine categories: *rules-based*, *graph-based*, *design structure matrix*, *model-driven*, *code smells analysis*, *reverse engineering and history-based*, *search-based*, *visualization*, and *others*.

The *rules-based* approach uses metrics with thresholds (rules), and pre-defined frameworks, heuristics, or guidelines to detect structural problems. This approach assesses enforced architectural guidelines, compliance checking, and identification of anti-patterns. It is often used with modularity metrics to identify modularity violations or complexity evaluations. They noted a common overlap with other methods in their classification framework.

The *graph-based* approach recognizes entities or components of a system as nodes, and the relationships are represented through edges. It is an intuitive method of representing problematic relationships between the architecture entities in graphical form. A dependency graph is often used to represent the system to assess modularization. Moreover, social network analysis can be used to predict undesired dependencies. When contrasting these findings with Sabir et al. (2019), this approach uses a graph as an intermediate representation. Mumtaz et al. (2021) mention that the *graph-based* approach often combines with the *rules-based* approach, which indicates different phases of where these two detection approaches belong to.

The *design structure matrix* approach uses a two-dimensional matrix representing the software's structural relationships. One side considers components, and the other considers relationships (i.e., dependency, coupling). However, such a matrix is another representation of a graph. Similarly, clustering and rules apply to this, which is similar to the previous.

The *model-driven* approach considers abstraction and modeling of architectural structures. One common mechanism used here is model transformation to generate intermediate representations which could be rendered in XML. This could also operate with metamodel extracted from the system by introspection or reflection. Similar to the previous two categories, this is another intermediate representation of the system. When we put to the context highlighted by Neri et al. (2020) and architecture viewpoints, it is obvious that multiple architectural perspectives are needed to describe the system. In this case, multiple system intermediate representations might co-exist, and to integrate these representations to answer complex questions, it must be possible to employ model transformations.

The *code smells analysis* approach involves the detection of lower-level anti-pattern or code metrics of the object-oriented paradigm to identify architectural anti-patterns. This approach involves correlation analysis. For instance, multiple code-level metrics can be combined to detect performance anti-patterns.

The *reverse engineering and history-based* approach uses multiple system versions, possibly mining software repositories, and could consider involved developers and issue tracking. The usage applies to assessing architectural guidelines and compliance checking to identify clusters with problematic dependencies and connections.

The *search-based* approaches identified by Mumtaz et al. were exclusively used in combination with the *rules-based* approach

and aimed at optimizing the search with genetic programming. Similarly, rules formulate using metrics, thresholds, or patterns.

The *visualization* approach involves human experts. They aid in the understanding of complex systems with multivariate and multidimensional data. In existing work, the visualization technique is used in combination with detection rules or to show the system architecture so that practitioners can detect issues themselves. It is relevant to enable interactive navigation to explore the system architecture.

The *other* approach does not represent commonalities and relates to rather unique techniques. These would include change scenarios, correlation analyses, practitioner interviews, architecture description languages, model transformations, profiler testing,

Mumtaz et al. (2021) did not consider the information extraction, which is considered by Sabir et al. (2019). Instead, one approach is considered for the entire process, which introduces gaps. For instance, it is not detailed in their classification framework how is the graph or model generated. Similarly, the *model-driven*, *graph-based*, and *design structure matrix* approaches have one commonality — they all are intermediate representations of the system. Furthermore, the *search-based* approach is a subcategory of *rules-based* approach. Finally, the *rules-based*, *code smells analysis*, *reverse engineering and history-based*, and *visualization* approaches likely all operate on top of an intermediate representation of the system.

4. The proposed MS anti-pattern detection classification framework

While the categories proposed by Mumtaz et al. (2021) may correspond to different keywords in literature, not all the MS anti-patterns can be mapped to their categories. Similarly, Sabir et al. (2019) presented another perspective of the classification, highlighting information extraction and intermediate representation for the analysis.

Given the gaps introduced in RQ 3.4, a more contextual understanding is needed if we were to build an analytical tool for microservices. For instance, given the Mumtaz et al. (2021) categorization, we may use a model that we reverse engineer from the source code to search for and to detect smells by using rules and visualizing the results for the human in the loop. Obviously, we just combined the majority of the categories suggested by Mumtaz et al. in order to achieve a single goal. Second of all, Sabir et al. (2019) add more insight to the detection with more holistic details. For the above reasons, we propose a different classification framework that takes into account different phases of such a process. At the same time, we provide the backward mapping to the Mumtaz et al. categorization and align with Sabir et al. in the holistic perspective which we augment.

Suppose we were to develop an automated analysis tool. First, we need to understand what the input is to extract information from on an automated basis. Next, we need to specify our focus and transform the input information into a more abstract model; we typically call this the intermediate representation. Having the intermediate representation reduces a lot of complexity, but it also implies we lose some detail to make the process more straightforward. The final step is to apply an appropriate strategy to identify the patterns.

Based on the aforementioned considerations, we based our classification framework on three phases:

- Phase 1: Information extraction (the input)
- Phase 2: Intermediate representation
- Phase 3: Detection techniques

Fig. 11 depicts the proposed process, together with examples of the generated artifacts derived from the literature. In the remainder of this Section, we describe these three phases in detail.

4.1. Phase 1: Information extraction

Literature might lead us to use system documentation (Rademacher et al., 2020) when extracting system information to analyze. However, we must be a step ahead. The system evolution factors easily make the documentation outdated (Cerny et al., 2022a). We must also recognize that there are always two perspectives of the system — structural one and behavioral one Sabir et al. (2019). What remains up to date with no question is the source code of the system or the runtime of the system itself. The approaches used for this include static and dynamic system analysis Sabir et al. (2019) or their hybrid combination. Our primary categorization thus considers these two approaches for automated system analysis for anti-patterns. However, there are even more fine-grained details about how we can execute these analyses.

Static analysis. This can involve analysis of the source code (Schiewe et al., 2022), involving code parsers turning the input into an Abstract-Syntax Tree (AST). For instance, Java Parser (https://javaparser.org) can be used when parsing Java source code. Similar parsing tools are available for other modern languages; for example, Python and Golang have a built-in parser package to obtain AST from the source code.

However, with cloud-native approaches, important descriptors are stripped of the code. Thus, it might be considered to analyze the entire codebase rather than just source code, including build files, deployment descriptors, or other configurations, unless they are part of the configuration server (which should also be a subject of interest). For instance, Ibrahim et al. (2019) used container descriptors to build microservice-based container networks to analyze plausible attack paths.

Apart from source code, binary code or bytecode might be available. When vendors wade into the "no source available" pool, there is no other path than this. However, with a bytecode decompiler, one can create source code from the bytecode. But there are other avenues for bytecode like using Javassist[6] ar ASM[7] libraries. Besides, GraalVM[8] (Wimmer, 2021) makes it clear that bytecode is not only the domain of Java but is also applicable to Python, JavaScript, R, or Ruby. The internal format of the GraalVM intermediate representation used for control and data flow analysis can be used to extract information from the system.

When the binary is the only path forward, Low-Level Virtual Machine (LLVM) (Lattner and Adve, 2004) can be used for control analysis, but that path is the most challenging given code structure information is no longer in its raw form but compiled.

Apart from all these perspectives is Mining Software Repository (MSR). The major difference is that MSR considers changes in time, organizational structures, change comments, authors, or even connected to ticketing systems like Jira.[9] Naturally, MSR would utilize parsers to extract information but then also version control commands to mine the history or change commits.

All these options, including *source code analysis*, *codebase analysis*, *bytecode/binary analyses*, or *code repository mining* could act as input for static analysis.

Dynamic analysis. This analysis requires the system to operate in order to collect logs, states, or metrics like CPU, memory,

[6] Javassist http://www.javassist.org.
[7] ACM https://asm.ow2.io.
[8] GraalVM https://www.graalvm.org.
[9] Jira https://www.atlassian.com/software/jira.

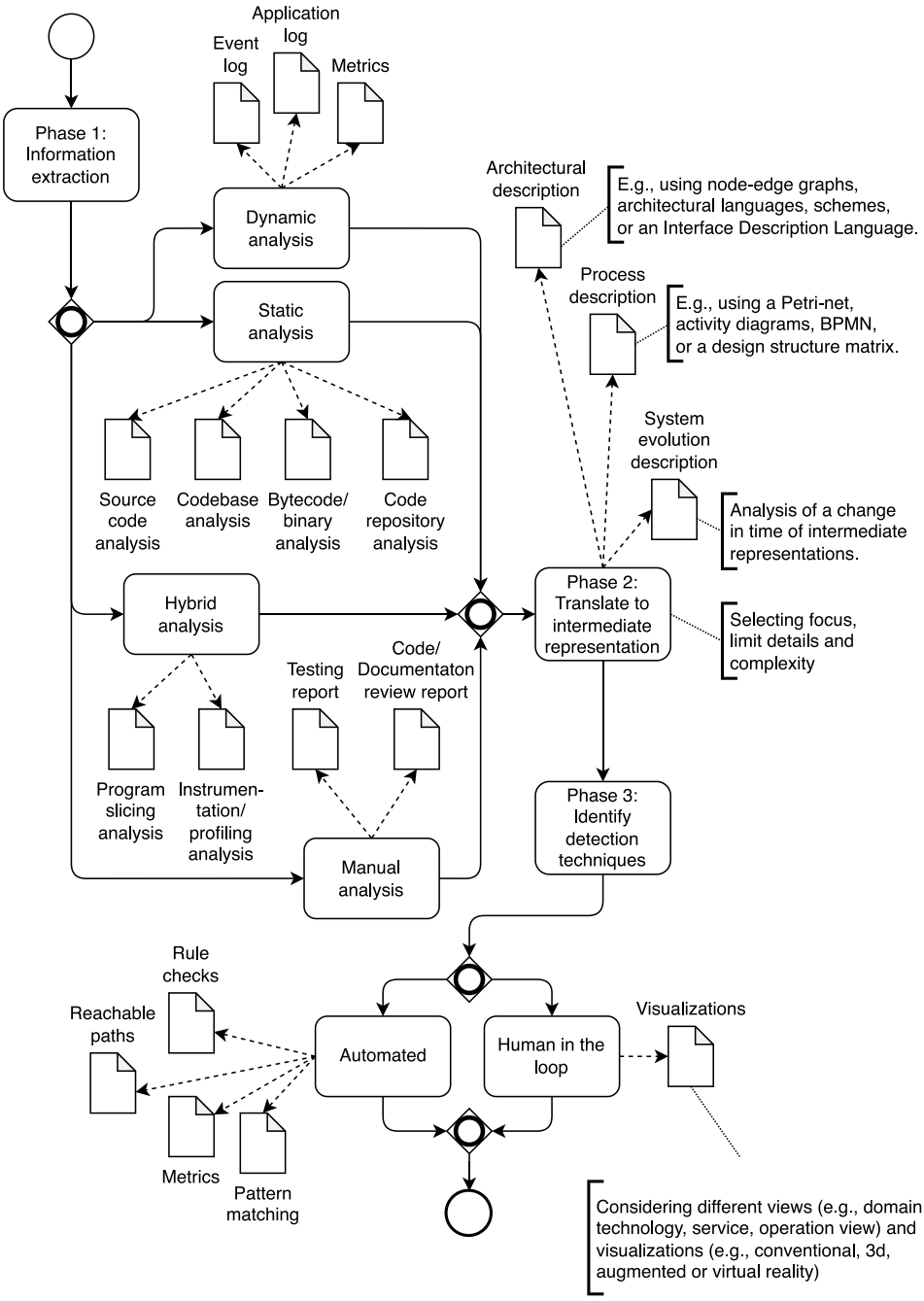

Fig. 11. BPMN diagram of the proposed anti-pattern detection process.

or latency between calls. There are multiple granularity levels for this as well. The first important thing to note is that application logs are typically centralized. Thus, we have a single focal point to observe what is happening.

However, the primary challenge with analyzing the application log is that we do not know what the log statements are relevant to apart from information about the producing source. In log analysis, we are particularly interested in understanding the sequence of actions triggered by a user. Given that we have concurrent environments, it becomes complex to identify the sequence of logged actions in the log related to one another. There have been scientific attempts (Schipper et al., 2019; Zhao et al., 2014) to understand log message sequences produced from concurrent runtime, but it is difficult.

To simplify the process of collecting the sequence of log statements in a business transaction, the mainstream direction is to augment log messages with user request-scoped business transitions. This can be accomplished by what is called tracing (i.e., OpenTelemetry[10]). The result of tracing is an event log. The tracing fabricates a randomly generated tracing identifier (ID) to

[10] OpenTelemetry https://opentelemetry.io.

each user's request, which is added to each logged statement (an event) and promoted across the entire system of microservices when intermediate calls occur. We need mediation either through the API gateway or the underlying service mesh.

With tracing ID, one can centralize the events, sort them by time and cluster them by tracing ID to understand the big picture in terms of interaction graphs and dependencies since we understand the sequence of events from user actions. In addition, if we connect the tracing ID with the user browser's cookies, we could understand the entire activity flow of the user's interaction with the system, not just the scope of individual requests.

The most basic tracing perspective is adding a log message for each endpoint. This way, we can reconstruct dependency graphs; however, it does not say anything about the internal control flow in the system across internal structures. We can see that dynamic analysis tends to be more of a black-box view.

However, the next question to ask is what granularity of logging should be maintained by each microservice or what the standard format of logging should look like. As a result, we might be dependent on what developers consider important to log, likely lacking consistency. For this reason, a hybrid approach can be used with instrumentation that generates the log statements automatically, which we mention in the next segment.

Apart from centralized logging, cloud systems typically employ health checks for resilience to avoid wobbly services. This involves every microservice to collect statistics on method call counts, timing, and success. Such statistics are centralized through similar means as tracing or used by service discovery. This allows for resilience mechanisms to temporarily disconnect services from the system to recover, apply rate limits, or auto-scale. Likely these statistics are great information to be analyzed for anomalies.

Hybrid approaches. When combined with static and dynamic analysis, hybrid approaches emerge. For instance, program slicing connects log statements to the code (Xu et al., 2005) to give more code insights upon log analysis, pointing log statements to specific code locations and parameters that cause a given anomaly. This instrument can be used when combining information from both approaches.

Code instrumentation is a common technique used to track application behavior. It refers to the task of including code in programs to monitor their performance. We can use code instrumentation to display messages or write to event logs in case of a failure during the execution of an application at run time. This goes hand in hand with tracing to ensure a homogeneous logging statement. Instrumentation is essential for metric collection in sufficient detail. Instrumentation involves an automated extension of source code or binary code. However, it introduces performance overhead, and developers might be sensitive to third-party code manipulation.

Prometheus[11] is a popular monitoring tool. It collects metrics from microservices for analytics. Instrumentation is essential for it to work. Before any service is monitored, code instrumentation is performed on their code via one of the Prometheus clients based on over 20 language-specific libraries.[12]

Manual approach. Yet, none of the above could inform about the configuration and process details of the system, and thus semi-automation could be necessary to add more comprehensive input. Among examples. How to determine that CI/CD is in place? How can we fetch configuration information from the configuration server unless available in the code repository of the configuration server? Besides, a test-based approach could be added here.

[11] Prometheus http://prometheus.io.

[12] Prometheus instrumentation http://prometheus.io/docs/instrumenting/clientlibs.

4.2. Phase 2: Intermediate representation

When gathering information about the microservice structure, its distributed traces, or metrics that are used for consequent analysis, we typically involve an intermediate representation to connect individual parts of information together. It can be an intermediary step for transformation to verification performed at a particular system perspective.

The intermediate representation could base on established graphs. The most basic AST can be converted to Control-flow graphs (CFG), Call graphs (CG), or other dependency graphs. These graphs consist of nodes and edges. CFG and CG are often misinterpreted as the same, but they are not. CFG is intra-procedural, where nodes represent program statements, including called subroutines but also conditionals and loops; the edges represent the flow of the program. CG is inter-procedural, where nodes represent subroutines, and edges represent the relationship caller-called between two subroutines (e.g., the caller subroutine A calls B subroutine). These two can combine to inter-procedural CFG. CG can be used in connection with instrumentation and tracing. However, CFG and AST cannot be determined from dynamic analysis.

Furthermore, research has been done to perform process mining on event logs produced by tracing. For these approaches, we typically use Petri-net. These are then used in activity diagrams or by the business-process model notation (BPMN). As Mumtaz et al. (2021) highlighted, the graph can be expressed as a matrix, and thus the intermediate representation can use a design structure matrix instead of a graph. Moreover, the graphs can represent something abstract or more detailed, whether temporal transitions across endpoints or the structure of objects or components, endpoints, or microservices themselves.

Specifically in microservices, we often see the service view perspective expressed in directed graphs. For instance, a service dependency graph is commonly used. The domain perspectives can represent schemas to represent data entities similar to what is used when modeling systems. Alternative representations can use component call graphs (Svacina. et al., 2022) spanning from endpoints, services, and repositories involving remote/messaging calls, data entities, and transfer objects.

The intermediate representation can also be expressed in the form of architectural languages (Lelovic et al., 2023) or schemes (i.e., JSON). When the system changes, and we analyze changes in time, we may consider a repository of graphs to determine deltas. This can be generalized to an N-dimensional structure of intermediate representations to system versions.

Given the common approach of graph representation, it might be feasible to involve graph databases like Neo4j.[13] As a result of static analysis of each microservices codebase, we get a forest of intermediate representations. These need to be combined. Bushong et al. (2022) illustrated that combining microservice intermediate representations to a holistic perspective can identify and map remote calls to the endpoint based on a signature match or user data similarity matches across microservice-bounded contexts. With a comprehensive system perspective, we can address anti-patterns related to *service interaction* or *inter-service decomposition*.

4.3. Phase 3: Detection techniques

When we look at the detection techniques, we are likely to traverse the intermediate representations and match patterns (i.e., `bottleneck service`), calculate metrics for nodes and edges (i.e., structural coupling), search for reachable paths

[13] Neo4j https://neo4j.com.

(i.e., `cyclic dependency`), perform specific rule checks (i.e., `shared persistency`, `no API versioning`). These all lead to automated technical reasoning.

However, the other perspective is to include humans in the loop and visualize the intermediate representation suggesting the human actor reasons (Cerny et al., 2022a) about the issues of the analyzed system. Then it leads to a completely different problem of which visual system perspective should be used or which system architectural perspective should be rendered. It is common to look at this from the software architecture perspective with the N+1 model to start with Neri et al. (2020), and we could see this in literature recognizing four common views of

1. Domain View: which covers the domain concerns of the system and describes the entity objects of the system as well as the data-source connections of those objects.
2. Technology View: that focuses on the technology aspect of the system and describes the technologies used for microservice implementation and operation.
3. Service View focuses on service operators and describes the service models that specify microservices, interfaces, and endpoints.
4. Operation View (topology): considering the system operation concerns and describing the service deployment of the infrastructure, such as containerization, service discovery, and monitoring

Unfortunately, this is not where it ends. The next question is what visualization technique to use to render these views. Other than conventional models rendered in two-dimensional space, there are opportunities for three-dimensional graphs. We could also consider augmented and virtual reality led by the metaverse initiative. We like leaving this as an opportunity for future researchers to fill the area with new works.

4.4. Sample solutions and tools

Sample solutions and tools are mentioned to provide more detail to the reader. Using source code analysis, Tighilt et al. (2023) presented a tool MARS to detect 16 microservice anti-patterns, similarly Walker et al. (2020) built the MSANose tool to detect 11 microservice anti-patterns. Approaches to these tools differ in the way they gather information, but both are performed using static analysis.

MARS collects information through search scripts performed on source files, imports, HTTP requests in the source code, database, and call graph using another tool called *Understand!*. They build a metamodel that serves as an intermediate representation. Finally, for the detection, they perform rule checks on top of the metamodel.

On the other hand MSANose approach involves a Java Parser that builds AST as the initial intermediate representation. The AST is, however, only an initial intermediate representation used to identify component call graphs. Individual results from each microservice are then merged based on combination rules described by Bushong et al. (2022). The limitation of operability on the Java platform has been addressed by Schiewe et al. (2022), who introduced a language-agnostic approach to detecting components needed for converting the code into component call graphs. In the end, specific rules are performed to match anti-patterns.

Bytecode analysis was utilized in the RAD tool by Das et al. (2021) to detect authorization inconsistencies in microservices. This approach used remote calls matched to the endpoint of other microservices to operate on the holistic system perspective.

Static analysis can be used to analyze intra-service design, inter-services decomposition, and service interaction and security. Still, some perspectives of development and operation can be covered.

It was also demonstrated that selected anti-patterns could be detected on traces (Al Maruf et al., 2022), especially when it involves inter-services decomposition and service interaction. This can extend the current tracing tooling with additional perspective.

Furthermore, Cerny et al. (2022b) illustrated that intermediate representations of microservices can be visualized in 3D space, which rendered beneficial for management and property identification tasks in large microservice systems (Abdelfattah et al., 2023).

4.5. Backwards mapping to Mumptaz et al.

Mumtaz et al. (2021) presented nine categories to detect anti-patterns: *rules-based*, *graph-based*, *design structure matrix*, *model-driven*, *code smells analysis*, *reverse engineering and history-based*, *search-based*, *visualization*, and *others*. We have proposed another framework on how to divide the perspective into three phases of the process. We map the categories as follows:

- The *rules-based* category maps to our phase three automated strategy.
- The *graph-based* category maps to our phase two, and the intermediate representation of many options are detailed.
- The *design structure matrix* is just another representation of the graph similar to the other category listing architectural languages.
- The *model-driven* category maps to our phase two and the intermediate representation where we apply phase three.
- The *code smells analysis* category maps to our phase three automated strategy.
- The *reverse engineering and history-based* category are two categories. Reverse engineering is the entire process, and the history-based also spans across all phases considering MSR and a repository of graphs. Furthermore, in the techniques, it would be a repetitive automated detection of deltas and their dependencies, and for the visual part, we would need to introduce provenance tracking (Burgess et al., 2022) to trace changes on the model in time.
- The *search-based* category maps to our phase three automated strategy.
- The *visualization* category maps to our phase three human in the loop.
- The *other* category maps to our phase one with manual analysis, including testing, but some outliers like architectural languages map to phase two.

This details how the proposed framework addresses the anti-pattern detection categories proposed in Mumtaz et al. (2021). Each category is mapped to a specific phase of the proposed method, either automated or involving human-in-the-loop, or manual analysis. This highlights the comprehensive approach of the proposed framework in addressing various aspects of microservice architecture from different perspectives.

5. Discussion

This section provides a discussion of our findings but also examines open challenges. It also elaborates on the validity threats.

5.1. Main findings

This study included seven secondary studies, which extracted evidence from 340 primary studies. This represents robust scientific evidence and knowledge. It also plots community scientific engagement in this topic. With the identified scientific resources, we could extract 58 distinct microservice anti-patterns grouped into five categories by their addressed-problem nature. While

some general violations apply to internal structures, topologies, etc., we considered the specifics of microservices when reducing the anti-patterns. This is often considered the cloud-native perspective that microservice architecture can offer when systems are properly designed. In such cases, we cannot consider structural or topological bottlenecks or hardcoded decisions and concerns.

When we consider system architecture as the main drive, the *intra-service design, inter-service decomposition, service interaction* categories are directly implied. From the definition, we could consider the design of individual elements (*intra-service design*) forming the system structure, which is decomposing the problem domain (*inter-service decomposition*) and considering the connection between elements (*service interaction*). The constraints can be of many forms, and *security* is clearly one of them. On the other hand, *team organization* for development and organization is more related to the development process.

The largest considered generalized category would thus be *system architecture* containing four of the introduced categories. However, the reduced abstraction to an individual perspective provided by the four categories brings the opportunity to focus on specific details (i.e., individual service granularity as opposed to system decomposition or security). The categorization of particular aspects will better serve particular interest groups. For instance, the *intra-service design* category directly impacts microservice developers who, by various interpretations of Conway's law, should take care of a single microservice. At the same time, architects might pay more attention to *inter-service decomposition* and *service interaction* categories. Security analysts then can then emphasize the focus to the *security* category. DevOps are essential for the cloud-native system and will likely pay the most attention to the *team organization*, which should also draw the attention of the system architect.

Naturally, we would expect automated tools integrated into development pipelines or development environments that can detect the catalog of 58 anti-patterns, but as of now, it is a question of tomorrow. There are a few pioneering works (Walker et al., 2020; Fontana et al., 2017; Al Maruf et al., 2022), but they often times detect a handful of anti-patterns. Certainly, our catalog will expedite the production of such tooling, which is urgently needed to provide feedback to practitioners and mitigate architecture degradation. To further promote this motion, this study introduced various detection techniques mentioned by the secondary studies and presented a generalized anti-pattern detection process with core phases that involved different techniques to extract information, form an intermediate representation, and then detect the particular anti-patterns that can automate processes or be used to advice human experts.

5.2. Implications

Our findings lead to various implications for microservice practitioners and researchers, including architects, developers, security analysts, testers, DevOps, etc. We list the implications in the following list:

- I1. Considering the existence of 58 MS anti-patterns, it is hard to expect that practitioners would recognize all of them. However, different practitioner roles (developers, architects, DevOps) can concentrate on particular categories of their closest interests.
- I2. The anti-pattern catalog can be extended with newly identified anti-patterns using the proposed categorization framework. It can be extended with proposed refactoring solutions to improve the system/process quality.
- I3. The proposed categorization framework can use additional tags to connect perspectives of interest for particular practitioner roles.
- I4. To properly use the proposed catalog, it is necessary for the microservice community to establish new detection tools that would assist them in recognizing poor practices, and the presented catalog facilitates determining what to look for.
- I5. While many studies on MS anti-patterns exist, they pay little attention to how to detect these anti-patterns. Both these directions need to be further investigated in the context of decentralized systems.
  - I5.1. We introduce a framework to detect anti-patterns given our previous experience with their detection (Walker et al., 2020; Al Maruf et al., 2022).
  - I5.2. The detection framework illustrates how to drive the means to stage and execute the detection process.
  - I5.3. The MS anti-pattern catalog can be extended with detection mechanisms demonstrated by newly developed tools.
- I6. Given that microservices can be polyglots and these anti-patterns are platform-independent, it is obvious that an intermediate representation is necessary to represent systems.
  - I6.1. The implication of the static analysis is that multiple language-specific language analyzers might be needed to cover commonly used language frameworks.
  - I6.2. The implication of polyglots can lead to prioritizing dynamic analysis over static analysis since tracing ID can be introduced into and instrumented into event traces.
- I7. If we could detect all 58 MS anti-patterns, the next question would be how to interact with developers and architects. Obviously, each pattern has a different critical aspect, and prioritization is necessary for advancements in this domain.
- I8. Given the complexity of MS systems, it might be difficult to detect anti-patterns. Moreover, with the involvement of many development teams, as suggested by Conway's law, it might be even difficult to assign responsibility for refactoring.
  - I8.1. If we had a proper architectural visualization of the system, we could consider interaction with human experts and render the detected anti-patterns via such visualization of the overall holistic system perspective.
- I9. MS anti-patterns play a certain role in system maintenance and evolution; however, the role is rather vague. Despite other studies on architectural degradation and technical debt, it remains a challenge to quantify anti-patterns and their impact on system maintenance and evolution.
- I10. The secondary studies considered some common design quality metrics as MS anti-patterns. For instance, the basic principles category operated with coupling, cohesion, and instability. To illustrate, what is the upper boundary when we call coupling high? It might be relative, but relative to what value or factors? Assessed studies did not put a contrast between these metrics and MS anti-patterns. However, it is necessary to investigate further whether common metrics could act as indicators for underlying problems, as they may play a different role when identifying weak spots in the system design.

- I11. With well established MS anti-pattern catalog, how can it be utilized to train the skilled workforce and continuously engage practitioners with new knowledge?
  - I11.1. As suggested in I4. MS anti-pattern detection tools could provide one trajectory to accomplish this by teaching practitioners about issues in projects they are familiar with.
  - I11.2. With proper anti-pattern visualization tools for sample project benchmarks, we could illustrate the residence of anti-patterns and possibly their impact on the system.
- I12. Certain challenges related to microservice system evolution (Bogner et al., 2021) could be considered in connection to anti-patterns.
  - I12.1. The ripple effect is an issue when one change to the system requires changes in other parts of the system. Obviously, *duplicated services* would introduce inconsistency. Similarly, the *co-change coupling* would be impacted, but perhaps some other changes could follow other dependencies. *Shared persistency* is another fragile place for this effect. We could also mine such dependencies from the history of code repositories and perform a correlation analysis of various codebases assessing change impacts.
  - I12.2. *Wrong cuts* could resonate with microservice coupling and cohesion in the system.
  - I12.3. Technological heterogeneity could be another indicator of organizational issues.
  - I12.4. While researchers like to consider the ideal case of cloud-native systems, we must accept that microservice are often strangled from legacy monoliths (*Strangler* pattern (Carnell and Sánchez, 2021)) and the system integration perspective might consider a different measure on parts that we actively develop and parts meant for legacy support.
- I13. Can there be anti-patterns found in a way that developers interact with particular microservice code-bases to enrich the organizational/operational category? For instance, does microservice development in a given organization conform to Conway's law ("organizations should design systems that mirror their own communication structure"), or does it depend on a single principle?

5.3. Threats to validity

Every similar type of work to ours has validity threats. Given this is a tertiary study, we carry the validity threats to the secondary studies we identified. To address these threats, this section elaborates on how we addressed them in this study and discusses limitations. Similar to other tertiary studies, we use the classification scheme proposed by Ampatzoglou et al. (2019). This classification considers the validity of the study selection (search strategy, selection criteria, extraction), data validity, and research validity.

Study selection validity The objectives relate to the risks of missing relevant studies, using relevant sources identification, and study inclusion and exclusion use. Initially, we were aware of multiple studies, which we used as the control sample for the later formed search string to evaluate its suitability for this study goal. We searched multiple peer-reviewed literature sources to reduce the threat of missing relevant secondary studies. Since this approach may still have missed relevant studies available in other literature sources, we also applied backward and forward snowballing. At the same time, it is important to acknowledge that we did not conduct a grey literature review. Therefore, any recently discussed anti-patterns that have not yet been documented in peer-reviewed literature or included in secondary studies might have been overlooked. Moreover, our protocol, research questions, and extraction attributes were defined before the study execution to limit bias. Three researchers assessed the protocol to ensure inclusion and exclusion criteria appropriateness. The quality assessment process is also subject to threats. We did not exclude any study, since they all passed the criteria. However, the DARE-4 framework adopted in the quality assessment does not cover all quality aspects (Costal et al., 2021). This is a common threat of quality assessment frameworks (Costal et al., 2021). For this reason, we selected DARE-4, which we considered the most appropriate framework for our evaluation, and is also the most frequently adopted in tertiary studies in Software engineering (Costal et al., 2021).

Data validity Identified secondary studies were only considered from peer-reviewed sources of accepted/published materials. Two co-authors independently analyzed titles and abstracts of identified studies towards the inclusion and exclusion criteria. All points of disagreement, including the third author, to resolve the disagreement. Our quality assessment process used a common framework for tertiary studies. The information extraction was consistent with the defined research questions. Given our data extraction process, two secondary studies did not provide anti-patterns which led to their exclusion from the study.

Some anti-patterns identified in our catalog may ultimately lead to disagreements between practitioners. Some instances could be more of a philosophical question, as a community will not settle on a single consensus. One such example could be *CRUDy service* on data-oriented services, which might be unavoidable, even desired, depending on the business scenario and project requirements (i.e., management of a big data Kubernetes cluster). A similar is the case with *shared persistency* or *shared libraries*; some experts exclude it, and others are fine with it. It was not our intention to invent new anti-patterns but to identify and report them in the search process according to the secondary studies. Specific contexts may imply particular needs and general guidelines may not apply. The potential exclusion of an anti-pattern based on one example could mean that practitioners would interpret it as a good design in general practice. Similarly, it was our intention to preserve identified anti-pattern names.

Another data validity threat arises from the definition of the categories of the anti-patterns classification scheme (Fig. 5). Specifically, the different categories of microservices anti-patterns might be broader or more fine-grained. We decided to use these categories because we considered that they would complement the results with additional useful information. Moreover, we applied an iterative coding process, followed by a validation step performed by experts. To this end, all anti-patterns were successfully assigned to a category.

Research validity Partial objective concerns the extent to which the results of our review can be generalized. Our study relies on previously identified and defined anti-patterns referenced by primary studies. Each anti-pattern was extracted with additional information, including categorization. All these attributes were used when collectively in person (three authors), determining the same anti-patterns with multiple alias names or when determining anti-pattern categories. Any conflict was resolved by the fourth author.

However, we acknowledge that qualitative analysis procedures are very subjective and, therefore, difficult to be reproduced identically by different researchers. One such perspective is anti-pattern categorization. The other is our detection framework.

It must be considered that other categorizations reflecting different perspectives could co-exist. In addition, some anti-patterns could occur in multiple categories.

Our MS anti-pattern detection classification framework has the base in the existing secondary study but considers the process behind tools performing such a detection. Our perspective possibly influences this framework. However, the main motivation for such a framework is to influence the audience to actively detect anti-patterns rather than defend them. Still, we believe that our anti-pattern catalog and detection classification framework presented in multiple categories can promote further empirical studies and find usefulness for a broad range of practitioners (i.e., system architects, developers, DevOps). In addition, we expect it to extend and evolve. Furthermore, we expect an evaluation of its relevance to microservice system evolution management.

6. Conclusion

Anti-patterns, a concept well-established in the field of software engineering, are common design solutions that, regrettably, prove to be ineffective or counterproductive. They have a detrimental impact on the system architecture, causing degradation over time. However, due to the market's emphasis on prioritizing the development of new functionality over quality design, these anti-patterns often remain unnoticed for extended periods, resulting in increased costs and maintenance complexity.

To promote high-quality system design and mitigate degradation, it is essential to detect anti-patterns early in their introduction to the codebase and undertake necessary refactoring. While anti-patterns are prevalent in various types of systems, the automated detection of anti-patterns in microservice (MS) systems is a relatively new domain. By employing automated means, it becomes possible to identify and address anti-patterns in MS systems, contributing to improved system quality and preventing long-term negative consequences.

In this work, we perform a tertiary study summarizing the evidence on MS anti-patterns, classifying them, and identifying detection methods. We considered seven secondary studies published between 2019 and 2022, listing a total of 58 different MS anti-patterns.

The catalog of MS anti-patterns we provide in this study has the potential to serve as a single focal point reference to practitioners active in this discipline. The provided classification can help readers to route through a large number of anti-patterns.

The proposed detection framework aims to engage the community in automated detection means. It also promotes separation of duty. Static or dynamic analysis experts can design tools that can extract the system's intermediate representation from a holistic perspective. There are two major perspectives to approach detection. The first involves automation and the other depends on human experts who could use the abstraction of the system model to more easily detect weak spots

To enable automation through technical reasoning, various graph representations of the system dependencies or structures can be used to detect a subset of anti-patterns from our catalog. However, the effectiveness of the reasoning process is contingent upon the quality and accuracy of the analyzed data, making it an approximation rather than an exact science. Therefore, although a combination of static and dynamic analysis offers greater reliability, it is still insufficient in addressing organizational perspectives or DevOps considerations.

Another detection technique might solely provide system models for human experts to do reasoning themselves. Yet these models are limited these days. We can find service dependency graphs as a product of dynamic tracing and monitoring, yet other models are necessary to provide a comprehensive perspective to such experts.

Yet with the identified catalog of anti-patterns along with detection techniques, there is a great opportunity for the scientific and practitioner community to develop tools to aid with anti-pattern detection to mitigate architectural degradation sourcing poor design practices that are left unseen to developers. We look forward to scientific peers using and extending this catalog. For instance, there is an opportunity for a grey literature review to augment this catalog. We also look forward to works analyzing the reasons behind the introduction of anti-patterns to the system design and a better understanding of the role of requirements for the design mismatch in the field.

In future work, we anticipate extending this catalog and implementing our detection framework, which will serve as a crucial contribution to the community and us as members within it. Additionally, we expect that new shifts may occur in the realm of microservices, potentially leading to the obsolescence of some of these anti-patterns. Future work will need to further investigate the complexity of understanding if an anti-pattern is present in the system as well as how useful it is to discuss about a given anti-pattern. Moreover, new contexts may give rise to the emergence of new anti-patterns.

Funding

This material is based upon work supported by the National Science Foundation, United States under Grant No. 1854049 and Grant No. 2245287, a grant from Red Hat Research https://research.redhat.com, a grant from the Ulla Tuominen Foundation (Finland), and a grant from the Academy of Finland (grant n. 349488 - MuFAno).

CRediT authorship contribution statement

Tomas Cerny: Conceptualization, Methodology, Validation, Investigation, Resources, Data curation, Writing – original draft, Visualization, Supervision, Project administration, Funding acquisition. **Amr S. Abdelfattah:** Conceptualization, Perform study, Resources, Data curation, Writing – original draft, Visualization. **Abdullah Al Maruf:** Conceptualization, Methodology, Investigation, Resources, Data curation, Writing – original draft, Visualization. **Andrea Janes:** Conceptualization, Methodology, Data curation, Visualization, Validation. **Davide Taibi:** Conceptualization, Methodology, Validation, Investigation, Resources, Data curation, Writing – original draft, Visualization, Supervision, Funding acquisition.

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 will be made available on request.

Appendix

See Table 13.

Table 13
The identified anti-patterns description.

Index	Anti-pattern name	Description	Example
		Intra-service design	
1	Nano-service (Rotem-Gal-Oz et al., 2012; Dudney et al., 2003; Palma et al., 2014b; Král and Ž.emlicka, 2009; Ouni et al., 2015, 2017; Nayrolles et al., 2013; Palma and Mohay, 2015; de Toledo et al., 2021; Taibi et al., 2020b; Schirgi, 2021) (**AKA**, Nano micro-service, tiny/fine-grained service)	The service is too fine-grained and only has a few operations. Its overhead (communications, maintenance, and so on) outweighs its utility. Nanoservices cause fragmented logic and performance issues due to communication overhead (i.e., finite bandwidth and transport costs). This pattern is related to migration anti-patterns like "big bang" or "data-driven migration".	Suppose we have a simple desk calculator with operations like add, subtract, multiply, and divide, where each operation is implemented as a separate microservice.
2	Mega service (Taibi and Lenarduzzi, 2018; Dudney et al., 2003; Ouni et al., 2017; Palma et al., 2014b; Ouni et al., 2015; Nayrolles et al., 2013; Palma and Mohay, 2015; Azadi et al., 2019; Carrasco et al., 2018a; Taibi et al., 2020b; Moha et al., 2010; Palma et al., 2013, 2014a; Marinescu, 2005, 2004; Marinescu and Rajiu, 2004; Palma et al., 2019; Ordiales Coscia et al., 2014) (**AKA**, Mega microservice, Blob or god object/component, God object web service, Multi-service, Bloated Service)	Microservices should be small, independent, independently deployable units and serve a single purpose. A mega service has a high number of lines of code, modules, or files, as well as a high fan-in. Mega service could be a result of poor system decomposition when the microservice combines multiple functionalities that should be handled by multiple services. Having a mega microservice creates maintenance issues, reduced performance, and difficult testing, in addition to the complexity of the microservices infrastructure.	An extreme example is a large monolithic service that tries to handle all functionality and business logic within a single codebase.
3	CRUDY service (Ouni et al., 2017; Palma and Mohay, 2015; Palma et al., 2019, 2014b) (**AKA**, Crudy interface, Crudy URI)	The design encourages services the RPC-like behavior by declaring create, read, update, and delete (CRUD) operations, e.g., createX(), readY(), etc. Interfaces designed in that way might be "chatty" and "nano" because multiple operations need to be invoked to achieve one goal. In general, CRUD operations should not be exposed via interfaces. This anti-pattern occurs when services declare CRUD-like verbs (e.g., create, read, update, or delete) are used in the APIs.	Consider an e-commerce application. In a well-designed microservice architecture, we might have services such as "User Management," "Inventory Management," and "Order Processing," each responsible for their respective functionalities. However, in the case of the nano service anti-pattern, the application may be decomposed into numerous excessively small services, such as "CreateUser," "UpdateUser," "DeleteUser," "GetUserDetails," "SearchUser," and so on. Each of these services would handle only a single CRUD (Create, Read, Update, Delete) operation related to user management.
4	Nobody Home (Palma et al., 2013, 2014b; Nayrolles et al., 2013; Vidal et al., 2016, 2019; Oizumi et al., 2015; Le et al., 2018a) (**AKA**, Unused interface)	Nobody Home corresponds to a functioning service that is defined but is actually never used by clients. In other words, it is not performing any useful work or providing any value. The methods from this service are never invoked, even though they may be coupled to other services. Yet, it still requires deployment and management, despite its non-usage. It hinders maintenance.	Consider a web app designed to collect user feedback and suggestions. Users can submit feedback through a form on the website, and the application is supposed to process and store that feedback for analysis by the development team. When a user submits their feedback, they receive a success message indicating that their feedback has been successfully submitted. However, behind the scenes, the feedback is essentially lost and not stored or processed in any way.

(*continued on next page*)

Table 13 (*continued*).

Index	Anti-pattern name	Description	Example
5	Data service (Palma et al., 2014b; Ouni et al., 2015, 2017; Palma and Mohay, 2015; Palma et al., 2013, 2014a; Nayrolles et al., 2013) (**AKA**, Data web service)	Data services usually contain accessor methods (i.e., getters and setters) with small parameters of primitive types. They may hold application state for other interacting services. Other, typically "nano," microservices may perform simple operations on data services like information retrieval or data access. On top can exist a "sand pile" service that performs a more complex task.	Assume a business process that involves multiple services in the pipeline, and each service retrieves and stores the computation state into a data service. Such data service lacks business logic and only serves other services to accomplish the business process.
6	No API-versioning (Taibi and Lenarduzzi, 2018; Taibi et al., 2020b; Schirgi, 2021) (**AKA**, API versioning) not semantically versioned. In the case of new versions of non-semantically-versioned APIs, API consumers may face connection issues. For example, the returning data might be different or might need to be called differently.	A bank system has multiple dependent clients (branch offices). The bank system upgrades one of the services with more advanced functionality which changes the semantics of certain endpoints. However, clients were not informed of the change and their system fails upon the rollout. If the system kept the original endpoint as version 1 and rolled out version 2 of the same service, the new clients could utilize new advancements while others would not experience disruptions.	
7	Whatever types (Palma et al., 2019; Mateos et al., 2015; Ordiales Coscia et al., 2013; Kitchenham and Charters, 2007; Coscia et al., 2012; Ordiales Coscia et al., 2014) (**AKA**, Ignoring MIME types, Forgetting hypermedia)	Service message exchange without paying attention to return types. A special data type is used for representing any object of the problem domain.	Service endpoint is returning different outcomes based on the context and input parameters, the return type is a generic class or object assuming the clients know all the underlying logic of the service endpoint to implement proper handling. It can lead to security vulnerabilities and unexpected behavior. For instance, when uploading images and checking file suffixes rather than MIME, we might accept executable files with fabricated suffixes (i.e., jpg). When we process the image, it could inadvertently execute the malicious code within the disguised executable file, potentially compromising the server or user's data.
8	Low cohesive operation (Palma and Mohay, 2015; Taibi and Lenarduzzi, 2018; Palma et al., 2014b; Mateos et al., 2015; Ordiales Coscia et al., 2013; Kitchenham and Charters, 2007; Coscia et al., 2012; Ouni et al., 2017; Ordiales Coscia et al., 2014)	It occurs when developers place very low cohesive operations (not semantically related) in a single microservice. A service that provides many low cohesive operations that are not related to each other. This can lead to a lack of clarity, maintainability issues, and difficulty in understanding and modifying the code.	A "DataProcessor" service can be responsible for processing different types of data, such as parsing CSV files, generating PDF reports, sending emails, and performing statistical calculations. The DataProcessor service is burdened with a variety of unrelated operations combined that do not belong together.

(*continued on next page*)

Table 13 (*continued*).

Index	Anti-pattern name	Description	Example
9	Ambiguous service (**AKA**, Ambiguous name, Ambiguous interface) (Garcia et al., 2009b; de Andrade et al., 2014a; Palma et al., 2014c; Palma and Mohay, 2015; Ouni et al., 2015)	For a service with unclear responsibilities, the service's responsibilities are not well-defined, leading to confusion and overlapping functionality. Developers may use ambiguous or meaningless names for denoting the main elements of interface elements (e.g., operations, messages). Ambiguous names are not semantically and syntactically sound and affect the discoverability and reusability of web services.	A company is developing a customer management system and decides to break down the system into several microservices for scalability and maintainability. One of the microservices they create is called "CustomerProfileService." However, during the development process, the team fails to clearly define the responsibilities of the CustomerProfileService. As a result, the service ends up having ambiguous functionality, making it difficult for other services and developers to understand its exact purpose and scope. For instance, the CustomerProfileService initially handles basic customer information, such as name, address, and contact details. But over time, developers start adding additional features to it, such as order history, payment preferences, and loyalty program details, without properly documenting or communicating these changes. As a consequence, other services in the system become uncertain about which functionalities they should handle themselves and which ones they can rely on the CustomerProfileService to provide. This ambiguity leads to duplicated efforts, inconsistent data, and potential bugs throughout the system.
		Inter-services Decomposition	
10	Transactional integration (Rotem-Gal-Oz et al., 2012)	Transactions extend across service boundaries instead of being isolated inside services. Transactions involve two or more separate services in a transaction assuming ACIDity, short time span, and pessimistic or optimistic locking for dual writes (i.e., data source write and logging). Rollback becomes distributed and difficult, the hold of resources becomes overhead, and state coordination is hard. We must ensure consistency that needs synchronization; however, holding locks for a long time can cause partial failures. Since services evolve independently, it is easy to expand the transaction beyond the controlled span.	Suppose a customer places an order, and the system needs to perform various tasks such as inventory management, payment processing, and notification sending. In a well-designed system, these tasks would typically be handled by separate services or components, each responsible for its own domain. However, in the case of this anti-pattern, a single transaction is initiated to encapsulate all of these operations. This means that if any part of the transaction fails, the entire transaction is rolled back, potentially undoing successful operations and causing inconsistencies.
11	Co-change coupling (Le et al., 2018b) logical coupling that occurs when changes to a service also require changes to another service.	An e-commerce application that consists of multiple microservices, including a Product Service, Order Service, and Payment Service. These microservices are responsible for handling product management, order processing, and payment transactions, respectively. In the current implementation, there is a high degree of co-change coupling between the Order Service and the Payment Service. Whenever there is a change in the Order Service, such as modifying the order data model or adding a new feature, it directly impacts the Payment Service. This is because the Payment Service heavily relies on the order information provided by the Order Service to process payments accurately. Now, suppose a new requirement arises to introduce a loyalty program for customers. This requires adding a loyalty point system to the Order Service, where customers can earn and redeem points. However, implementing this change would not only require modifications in the Order Service but also in the Payment Service. The development team has to coordinate their efforts between the two services, slowing down the implementation process.	

(*continued on next page*)

Table 13 (*continued*).

Index	Anti-pattern name	Description	Example
12	Duplicated services (Palma et al., 2014b; Palma and Mohay, 2015; Jones, 2006; Král and Ž.emlicka, 2009; Rotem-Gal-Oz et al., 2012; Palma et al., 2013, 2014a; Ouni et al., 2017; Nayrolles et al., 2013; Ouni et al., 2015) (**AKA**, Nothing new)	A set of highly similar services, dynamically or syntactically. These services perform similar or identical functions (semantic duplication), leading to redundancy and increased complexity. Services might be implemented multiple times with common or identical methods with similar names and/or parameters (syntactic duplication). It goes against the principles of modularity, reusability, and maintainability.	An e-commerce application that consists of several microservices. One of these microservices is responsible for handling user authentication and authorization, named "Auth-Service." Another microservice, named "User-Service," is responsible for managing user profiles and personal information. However, due to poor communication and coordination among the development teams, the User-Service team decides to include authentication and authorization functionality within their microservice. As a result, both the Auth-Service and User-Service end up providing similar authentication and authorization capabilities.
13	Microservice greedy (Dudney et al., 2003; Taibi and Lenarduzzi, 2018)	When in doubt, make it a service. Despite the absence of any measurable advantages, a system provides its functions as services. The result is the explosion in the number of services that makes the system hard to understand.	Instead of analyzing existing services to find which of them are related to a desired new functionality and selecting a candidate for extension, a new microservice is introduced possibly adding broad interaction with established services.
14	Service chain (Nayrolles et al., 2013; Jones, 2006; Palma et al., 2013, 2014a; Ordiales Coscia et al., 2014; Cortellessa et al., 2014) (**AKA**, Pipe and filter, Message Chain)	A chain of service calls that fulfills common functionalities resembling a transitive manner. It appears when clients request consecutive service invocations to fulfill their goals.	An e-commerce application consists of several microservices: Order Management, Inventory Management, Payment Processing, and Shipping. In a "tightly coupled" service chain, the flow of actions would be as follows: The Order Management service receives an order request from a customer and invokes the Inventory Management service to check product availability. If the product is available, the Order Management service calls the Payment Processing service to process the payment. Once the payment is successful, the Order Management service requests the Shipping service to ship the order. Each service depends on the successful completion of the previous service to proceed. If any service fails or experiences delays, the entire chain may break down, leading to a poor user experience and system instability.
15	Hub-like dependency (Azadi et al., 2019)	A service has (outgoing and ingoing) dependencies with a large number of other services. The service becomes a central point of dependency for many other services.	A system provides various functionalities, such as user management, content management, and analytics. In this system, we have a central "Application" class that handles all the core logic and acts as a hub for other services.
16	Cyclic dependency (Taibi and Lenarduzzi, 2018; Azadi et al., 2019; Pigazzini et al., 2020; Taibi et al., 2020b; Bogner et al., 2019c) (**AKA**, Cyclic between namespaces)	A cyclic chain of calls between services exists. When two or more services depend on each other directly or indirectly. The services involved in a dependency cycle can be hard to release and maintain. This dependency implies that there are two pieces of code that are highly coupled to each other in a direct or indirect way. This situation might suggest that the responsibilities are not separated correctly across services. Various cyclic dependency shapes can be recognized. This leads to problems with deployment, scalability, and co-change coupling.	In an e-commerce system: Order Service and Customer Service. The Order Service is responsible for handling order processing and relies on the Customer Service to retrieve customer information. On the other hand, the Customer Service is responsible for managing customer data and relies on the Order Service to retrieve order history.
17	Chatty service (Palma et al., 2014b; Ouni et al., 2015, 2017; Nayrolles et al., 2013; Palma and Mohay, 2015; Palma et al., 2013, 2014a; Cortellessa et al., 2014) (**AKA**, Empty semi-trucks) (**AKA**, Circuitous treasure hunt)	One service excessively communicates with other microservices. Service may need to perform multiple fine-grained operations or look at several places to find the information that it needs. As a result, it degrades the overall performance with a higher response time. Batching or transfer-objects that combine items into messages can address the problem.	An e-commerce system has inventory management and order processing. Whenever a customer places an order, the Order service needs to check the availability of items in the Inventory service before proceeding with the order. The communication flow might look something like this: The Order service sends a request to the Inventory, requesting item availability. The Inventory processes the request and sends a response back to the Order with the availability status. Based on the availability status, the Order decides whether to proceed with the order or not. Now, imagine that the Order frequently sends requests to the Inventory for every single item in the customer's order, one by one. This means that for an order with multiple items, there will be multiple round trips between the two microservices.

(*continued on next page*)

Table 13 (*continued*).

Index	Anti-pattern name	Description	Example
18	Shared persistency (Bhojwani, 2018; Carnell, 2017; Carrasco et al., 2018b; Furda et al., 2018; Golden B., 2018; Indrasiri, 2017; Indrasiri and Siriwardena, 2018a; Kalske et al., 2018; Knoche and Hasselbring, 2018; Nadareishvili et al., 2016; Richards, 2016; Richardson, 2014, 2018; Saleh, 2016; Soldani et al., 2018; Taibi and Lenarduzzi, 2018; Taibi et al., 2017, 2018; Wolff, 2016; de Toledo et al., 2021; Taibi et al., 2020b; Toledo et al., 2020)	Different microservices are accessing the same database. In the worst kind, different services use the same entities of a service. This approach couples the microservices connected to the same data and reduces the service independence. As a result, it introduces tight coupling requiring service coordination upon deployment, data inconsistency with concurrent updates, and performance bottlenecks limiting scalability and limited flexibility when modifying the data model (Schirgi, 2021; Taibi and Lenarduzzi, 2018; Tighilt et al. (2020)).	An e-commerce system composed of two microservices: "Order Service" responsible for managing customer orders and "Inventory Service" responsible for managing product inventory. Initially, both microservices have their separate databases, ensuring independent data management. However, as the system evolves, the development team decides to implement a new feature that requires real-time synchronization between the Order Service and Inventory Service. Specifically, they want to prevent customers from placing orders for products that are out of stock. To achieve this, the team decides to introduce a shared database table named "ProductStock" accessible by both microservices. Whenever an order is placed, the Order Service updates the stock quantity in the ProductStock table, and the Inventory Service reads from this table before approving an order.
19	Sand pile (Kral and Zemlicka, 2007; Palma and Mohay, 2015; Palma et al., 2013, 2014a; Ordiales Coscia et al., 2014)	It appears when a service is composed of multiple "nano services" sharing common data and facing "shared persistency" or "data service" anti-patterns. The anti-pattern blocks many good practices like information hiding (what is well known) but also the agility of business processes, well-usable logging, etc.	An e-commerce platform consisting of several microservices: Product Catalog, Inventory Management, Order Processing, Payment Gateway, and Shipping Logistics. Initially, the microservices are designed to be loosely coupled and have minimal dependencies. Each microservice has its own database and communicates with others through well-defined APIs. The system operates smoothly, and new features are developed and deployed rapidly. Over time, new requirements and business needs arise. Each change or new feature introduces more complexity and interdependencies between the microservices. For example, a new customer loyalty program requires the Order Processing service to access the customer's purchase history from the Product Catalog and Inventory Management services. Additionally, the Payment Gateway needs to validate loyalty program discounts, and the Shipping Logistics service requires additional information from the Order Processing service to determine shipping priorities. As more requirements pile up, the microservices start to become tightly coupled, and multiple services need to be updated simultaneously to introduce new features. The team faces challenges in coordinating these changes, and the system becomes brittle. Any small change in one service can inadvertently impact other services, leading to unforeseen issues, such as cascading failures or unintended side effects.
20	Shared libraries (Taibi and Lenarduzzi, 2018; Pigazzini et al., 2020; de Toledo et al., 2019; Taibi et al., 2020b; Bogner et al., 2019c; Schirgi, 2021) (**AKA**, Shared Dependencies)	Microservices should not share runtime libraries and source code directly. This somehow breaks the boundaries between microservices, which then cannot be seen as independent and independently deployable. Runtime assets should not be shared even at the cost of the DRY principle.	Imagine a microservice system where multiple services depend heavily on a shared library that contains business logic and utility functions. Over time, this shared library grows in size and complexity as different services contribute to its codebase. These challenges emerge: there is a coupling between the shared library and services (library API changes force changes in multiple services); versioning issues when managing dependencies to ensure compatibility, increased complexity as the shared library grows, deployment coordination, performance bottlenecks, and finally, not all library functionality is needed by a service which results in draining its resources.

(*continued on next page*)

Table 13 (*continued*).

Index	Anti-pattern name	Description	Example
21	Wrong cuts (Taibi and Lenarduzzi, 2018; Taibi et al., 2020b; Bogner et al., 2019c)	The system is decomposed into microservices following technical aspects, such as the presentation layer, business layer, and data access layer. Microservice should encapsulate functionalities fulfilling a single purpose.	An e-commerce application that consists of several microservices. One of the microservices is responsible for handling product inventory management, and another microservice handles customer orders and payments. Now, imagine that during the initial design phase, the development team decides to divide the services based solely on the UI components of the application. They create separate microservices for the product listing page, product details page, shopping cart, and checkout process. Each microservice is developed and deployed independently.
22	Knot service (Rotem-Gal-Oz et al., 2012; Nayrolles et al., 2013; Palma and Mohay, 2015; Palma et al., 2013, 2014a; Ordiales Coscia et al., 2014)	Where the services are tightly coupled by hardcoded point-to-point integration and context-specific interfaces. The first service is designed well. Then you design the second service, and the two services talk to each other. Then comes a third service, and it has to talk to the other two. The fourth service only talks to some of the previous ones. The twelfth talks to nine of the others, and the fourteenth has to contact them all—yep, your services are tangling up together in an inflexible, rigid knot.	A large e-commerce platform consisting of various microservices such as User Management, Product Catalog, Order Processing, and Payment Gateway. Initially, these microservices were designed to be loosely coupled and communicate through well-defined APIs. However, over time, the developers notice that a central service called "Inventory Management" is essential for multiple microservices. Initially, the Inventory Management service was responsible for keeping track of product availability and stock levels. However, due to increasing business requirements, other microservices also started relying heavily on this service. The User Management service needs to check product availability before displaying items to users, the Order Processing service needs to reserve inventory when an order is placed, and the Payment Gateway service needs to verify stock levels before processing payments. As a result, the microservices become tightly coupled with the Inventory Management service. Any changes or issues with the Inventory Management service have a cascading effect on other microservices.
23	Scattered parasitic functionality (Garcia et al., 2009c; de Andrade et al., 2014b; Ouni et al., 2015; Dudney et al., 2003; Palma and Mohay, 2015; Palma et al., 2013, 2014a; Ordiales Coscia et al., 2014; Garcia et al., 2009c) (**AKA**, Stove pipe service)	Multiple services are responsible for realizing the same high-level concern and, additionally, some of those components are responsible for orthogonal concerns. Additionally, at least one service addresses multiple concerns, which creates a bottleneck for modifiability. Services realizing scattered concerns are dependent on each other, thus having their reusability and modularity reduced.	An e-commerce application that consists of several microservices: User Management, Product Catalog, Order Management, and Payment Processing. Each microservice is responsible for its specific domain. However, due to evolving requirements, the development team decides to introduce a new feature that involves sending email notifications to users for various events, such as order confirmation, shipment updates, and promotional offers. Instead of creating a dedicated Email Notification microservice to handle this functionality, they decide to scatter the email-related code across the existing microservices. In this scenario, each microservice starts to include its own code for sending emails. For example, the Order Management microservice adds email-sending logic to handle order confirmation emails, the Product Catalog microservice includes code for sending promotional emails, and so on. Over time, the code for sending emails becomes duplicated across multiple microservices.

(*continued on next page*)

Table 13 (*continued*).

Index	Anti-pattern name	Description	Example
		Service Interaction	
24	ESB misuse (Bonér, 2016; Indrasiri, 2017; Indrasiri and Siriwardena, 2018a; Lewis and Fowled, 2014; Taibi and Lenarduzzi, 2018; Zimmermann, 2017; Taibi et al., 2020b) (**AKA**, ESB usage)	ESB (enterprise service bus) is positioned as a single central "hub-like dependency", with microservices as spokes. When combined with the "use of business logic in communication among services" the hub becomes a bottleneck both architecturally and organizationally. In a properly designed microservices architecture, each microservice should be autonomous and responsible for its own functionality and data. They communicate with each other through lightweight protocols, such as HTTP or messaging systems like RabbitMQ or Apache Kafka. However, in the ESB misuse functionality anti-pattern, the ESB is used as a heavyweight intermediary for all communication between microservices.	Suppose we have a microservices architecture consisting of three services: User Service, Order Service, and Email Service. The User Service is responsible for managing user information, the Order Service handles order processing, and the Email Service sends notifications to users. In the ESB misuse functionality anti-pattern, instead of allowing direct communication between microservices, all communication is routed through the ESB. So, when the Order Service needs to send an email notification to a user after processing an order, it sends a request to the ESB. The ESB then receives the request, processes it, and forwards it to the Email Service.
25	On-line only (No Batch Systems) (Kral and Zemlicka, 2007)	Microservices are designed with independent services, but the communication between them relies heavily on synchronous calls. Batch mode application parts are actively avoided in the system, even though parts of the system, like long-running tasks, would be primed for a batch system. Some example batch candidates are: legacy systems, activities that require a lot of time to process or need additional user responses, or other performance reasons. The integration can be via message queues or data stores.	Two services interact where the producer takes time to respond, making the consumer wait an extended time. A message queue would be a better form of communication in this format. Another example is when a feed service parses a Twitter feed and contacts three consumers about the most recent information it gathered.
26	Empty messages (Ordiales Coscia et al., 2013; Coscia et al., 2012; Ordiales Coscia et al., 2014)	Service message exchange with empty messages that act like signals. Empty messages are used in operations that do not produce outputs nor receive inputs.	We have two microservices in a system: "OrderService" and "PaymentService." The OrderService is responsible for handling customer orders, and the PaymentService handles payment processing. When a customer places an order, the OrderService needs to communicate with the PaymentService to process the payment. In the empty messages anti-pattern, the OrderService might send an empty message to the PaymentService simply to trigger the payment processing, without including any relevant order information. The PaymentService would then need to retrieve the order details from its own database or another service, which introduces unnecessary overhead and increases the complexity of the system.
27	Use of business logic in communication among services (de Toledo et al., 2021, 2019)	Service communication contains business logic in the communication layer. Microservices should employ what is called a dump pipe or a communication layer without business logic. However, the data transported can change within the communication channel itself. The changes are made by the services communication channel using business logic. Maintaining additional business logic apart from the services is costly, as any changes to the services may also require changes to the communication layer where the business logic is located. Besides, "each time a new system is on-boarded, you need to set up the communication flow, requiring the communication channel team to provide the flow and possibly set up some business logic". In other words, an external team—the communication channel maintainers—must understand details about how the related services work to implement the business logic.	A communication that concerns the transfer of data (e.g., messages, computational results, etc.) between services; coordination that concerns the transfer of control (e.g., the passing of thread execution) between services; conversion concerned with the translation of different interactions between services (e.g., conversion of data formats, types, protocols, etc.); and facilitation that describes the mediation, optimization, and streamlining of interaction (e.g., load balancing, monitoring, and fault tolerance).

(*continued on next page*)

Table 13 (*continued*).

Index	Anti-pattern name	Description	Example
28	Hardcoded endpoint (Taibi and Lenarduzzi, 2018; Jones, 2006; Pigazzini et al., 2020; Taibi et al., 2020b; Schirgi, 2021; Brogi et al., 2019; Alshuqayran et al., 2016; Balalaie et al., 2016, 2018; Bhojwani, 2018; Bonér, 2016; Francesco et al., 2019, 2017; Indrasiri, 2017; Indrasiri and Siriwardena, 2018a; Krause; Lewis and Fowled, 2014; Long J., 2015; Nadareishvili et al., 2016; Newman, 2015; Nygard, 2018; Richardson, 2014; Saleh, 2016; Wolff, 2016) (**AKA**, Endpoint-based service interactions)	Microservice IP addresses, ports, and endpoints are explicitly/directly specified in the source code, configuration files, or environment variables. Running multiple instances of a microservice with a load balancer becomes impossible. Changing the IP address or port number of a microservice requires changing and redeploying other microservices.	When a developer directly embeds a specific API endpoint URL within the source code of an application instead of using a service registry (i.e., HashiCorp Consul or Netflix Eureka, where microservices can register their endpoints dynamically).
29	No API-gateway (Alagarasan V., 2015; Balalaie et al., 2018; Bhojwani, 2018; Bonér, 2016; Carnell, 2017; Francesco et al., 2019, 2017; Indrasiri and Siriwardena, 2018a; Krause; Nadareishvili et al., 2016; Nygard, 2018; Richardson, 2014, 2018; Soldani et al., 2018; Taibi and Lenarduzzi, 2018; de Toledo et al., 2019; Taibi et al., 2020b; Bogner et al., 2019c; Schirgi, 2021; Brogi et al., 2019)	When a microservice-based system lacks an API gateway, the clients of the application necessarily have to invoke its microservices directly. By not having an API gateway, the system lacks a unified entry point that can provide centralized security, routing, protocol translation, and other cross-cutting functionalities. It becomes harder to enforce consistent policies across microservices and complicates the overall management and evolution of the system.	A e-commerce application consists of Product Catalog, Order Management, and User Authentication microservices. Each microservice directly exposes its API to external clients without a centralized API gateway. Product Catalog has: • GET /products: Retrieves a list of products. • POST /products: Creates a new product. Order Management has: • GET /orders: Retrieves a list of orders. • POST /orders: Creates a new order. User Authentication has: • POST /login: Authenticates a user and returns a token. A client application wants to display a product catalog and allow users to add products to their shopping cart. Without an API gateway, the client application needs to make separate API calls to each microservice. The client application sends a request to the Product Catalog microservice to retrieve the list of products. When a user adds a product to their cart, the client application needs to send a request to the Order Management microservice to create a new order. If the user is not authenticated, the client application needs to send a request to the User Authentication microservice to authenticate the user. With this approach, the client application has to handle multiple API calls, manage authentication tokens separately, and deal with potential inconsistencies and complexities arising from direct communication with individual microservices. There is no centralized mechanism to handle cross-cutting concerns like authentication, request validation, logging, and rate limiting.

(*continued on next page*)

Table 13 (*continued*).

Index	Anti-pattern name	Description	Example
30	Wobbly service interactions (Alshuqayran et al., 2016; Balalaie et al., 2016, 2018; Bhojwani, 2018; Bonér, 2016; Carnell, 2017; Dall, 2016; Francesco et al., 2019, 2017; Golden B., 2017; Indrasiri, 2017; Indrasiri and Siriwardena, 2018a; Jamshidi et al., 2018; Kalske et al., 2018; Knoche and Hasselbring, 2018; Krause; Lewis and Fowled, 2014; Long J., 2015; Nadareishvili et al., 2016; Newman, 2015; Nygard, 2018; Richards, 2016; Richardson, 2018; Ruecker, 2019; Saleh, 2016; Soldani et al., 2018; Wolff, 2016; de Toledo et al., 2021; Brogi et al., 2019; Cortellessa et al., 2014; Nayrolles et al., 2013; Palma and Mohay, 2015; Palma et al., 2013, 2014a; Ordiales Coscia et al., 2014) (**AKA**, Bottleneck service, Traffic jam, Ramp)	Occur when a service interacts with another service or with a message router, not including support for tolerating failures; that is, there is a lack of resilience. Various known resiliency patterns support tolerating failures and aim to prevent their cascading, avoid data consistency errors, or mitigate partial service failure. To preserve functionality in the event of service failure, we use various best practice resilience solutions recognized in the form of design patterns (i.e., Resilience4j) like: *Client-side load balancing* - (tolerant of failure) - having the client look up all of a service's individual instances from a service discovery agent and then cache their physical location. *Rate Limiter* - (reduce traffic) - reduces the number of records sent to a service over a given period of time (throttle based on different metrics over time). *Bulkhead* - (tolerant of failure) elements of an application are isolated into pools so that if one fails, the others will continue to function. *Circuit Breaker* - (preventing a service failure from long-lasting requests) - counts the number of recent failures that have occurred and uses that to decide whether to allow the operation to continue or return an exception immediately. *Retry* - (handle transient failures) - tries to connect to a service or network resource by transparently retrying a failed operation (can combine with client-side load balancing). *Timeout* - (limits delay propagation) - to consider this service unavailability issue while designing service dependencies and accounting network issues delaying responses.	An e-commerce system composed of multiple microservices, including a product catalog service, an inventory service, and a payment service. The product catalog service is responsible for managing product information, the inventory service handles stock availability, and the payment service processes payment transactions. However, due to the Wobbly service interactions, the communication between these services becomes unstable, leading to issues. Cascading failures: The product catalog service experiences intermittent connectivity issues with the inventory service. As a result, when a user tries to view a product, sometimes the inventory service fails to respond, causing a timeout in the product catalog service. This timeout then cascades to the user interface, resulting in slow or unresponsive pages. Inconsistent data: The inventory service occasionally fails to update its stock availability in real-time. When a user adds a product to the shopping cart, the inventory service may not immediately reflect the decrement in available stock. Consequently, the user might be able to purchase a product that is actually out of stock, leading to order cancellations and dissatisfied customers. Partial failures: The payment service encounters intermittent issues while communicating with external payment gateways. As a result, some payment transactions fail, while others succeed. This inconsistency creates confusion for both customers and the order management system, leading to delayed order processing and potential financial discrepancies. Performance degradation: The wobbly service interactions cause increased latency and decreased overall system performance. The frequent timeouts, retries, and partial failures result in slower response times, reducing the application's usability and potentially driving away customers.
31	Timeout (Schirgi, 2021; Taibi and Lenarduzzi, 2018)	The service consumer cannot connect to the microservice. Mark Richards (Richards, 2016) recommends using a time-out value for service responsiveness or sharing the availability and the unavailability of each service through a message bus so as to avoid useless calls and potential time-outs due to service unresponsiveness. Request retrial and timeout values are good signs of the presence of this anti-pattern.	A system has three services: Service A, Service B, and Service C. Service A needs to make a request to Service B, which in turn depends on Service C to complete the operation. To handle the communication between services, Service A sets a fixed timeout of 1 s for the request to Service B. However, due to various factors such as network latency, increased load on Service B, or complex data processing in Service C, the response from Service B may take longer than the specified timeout. The timeout anti-pattern occurs in this scenario because Service A does not account for the potential delays and assumes that if the response does not arrive within 1 s, there must be an error. As a result, it either cancels the operation prematurely or returns an error to the client, even though the operation might still be in progress and could eventually succeed.

(*continued on next page*)

Table 13 (*continued*).

Index	Anti-pattern name	Description	Example
32	No health check (Palma et al., 2013, 2014a; Ordiales Coscia et al., 2014)	A microservice can be deployed anywhere and can become unavailable for a particular amount in a particular context. Consumers of a given microservice may experience timeouts and long waiting times without getting a response in case the microservice is down. No periodic HTTP request, no API gateway, or no service discovery can be hinted about the service being down unless endpoints are exposed to check the health of the given microservice.	E-commerce system consists of several services, such as product catalog, user authentication, order processing, and payment gateway. In this architecture, the "No health check" anti-pattern occurs when none of the microservices implement health checks. In this scenario, suppose the payment gateway service encounters a critical issue that prevents it from connecting to the payment provider. Without a health check, the other services in the system will continue to send requests to the payment gateway service assuming it is operational. As a result, the entire system will experience degraded performance or even fail to process orders and payments. Furthermore, since there are no health checks implemented, there will not be any monitoring or alerting mechanisms in place to notify the operations team or developers about the issue. As a result, the problem might go unnoticed until users start reporting errors or failures, causing a negative impact on the user experience and the business.
		Security	
33	Unauthenticated traffic (Sahni, 2020; Boersma, 2019; Chandramouli, 2019; Hofmann et al., 2017; McLarty et al., 2018; Abasi, 2019; Doerfeld, 2015; Budko, 2018; Anon, 2019c; Smith, 2019; Gebel and Brossard, 2018; Lea, 2015; Nkomo and Coetzee, 2019; Nehme et al., 2019a)	When unauthenticated API requests come from external systems or when there are unauthenticated requests between the microservices of the application themselves	One of the microservices, called the "Order Service," is responsible for managing customer orders. The Order Service exposes an endpoint, such as "/createOrder," that does not require any authentication. Any individual or malicious actor could send requests to the "/createOrder" endpoint and create orders on behalf of customers without going through the proper authentication process.
34	Multiple user authentication (Mannino, 2017; Gardner, 2017; Chandramouli, 2019; Anon, 2019b; Hofmann et al., 2017; Pacheco, 2018; da Silva, 2017; Kanjilal, 2020; Dias and Siriwardena, 2020; Douglas, 2018; Behrens and Payne, 2016; Smith, 2019; Gebel and Brossard, 2018; Smith, 2017; Nehme et al., 2019b; Indrasiri and Siriwardena, 2018b; Jackson, 2017; Nkomo and Coetzee, 2019; Nehme et al., 2019a; Mateus-Coelho et al., 2020)	The Multiple User Authentication occurs when a microservice-based application provides multiple access points to handle user authentication. Each access point constitutes a potential attack vector that an intruder can exploit to authenticate as an end-user, and having multiple access points hence results in increasing the attack surface to violate authenticity in a microservice-based application. The use of multiple access points for user authentication also results in maintainability and usability issues since user login is to be developed, maintained, and used in multiple parts of the application.	In this anti-pattern, each microservice manages its own authentication logic and maintains its own user database or identity provider. When a user tries to access a service, they are required to provide their credentials, and the service validates those credentials independently.

(*continued on next page*)

Table 13 (*continued*).

Index	Anti-pattern name	Description	Example
35	Publicly accessible microservices (Troisi, 2017; Mannino, 2017; Gardner, 2017; Anon, 2019a; Krishnamurthy, 2018; Carnell, 2017; Siriwardena, 2019; Pacheco, 2018; Khan, 2018; Kanjilal, 2020; Matteson, 2017b; Dias and Siriwardena, 2020; McLarty et al., 2018; Abasi, 2019; Douglas, 2018; Gebel and Brossard, 2018; Smith, 2017; Nehme et al., 2019b; Indrasiri and Siriwardena, 2018b; Jackson, 2017; Bogner et al., 2019d; Nehme et al., 2019a; O'Neill, 2020; Kamaruzzaman, 2020; Rajasekharaiah, 2020)	The publicly accessible microservices occurs whenever the microservices forming an application are directly accessible by external clients. It is essential to carefully audit all externally-accessible APIs to determine the information they can reveal and the internal systems they touch.	Consider a microservice responsible for an organization's social media message updates. This microservice has low resource demands and likely does not need to scale. Upon system deployment, administrators took a shortcut and enabled interaction with all clients. Thus the service API became public to post social media messages by external users. To address the issue, they could limit accepted IP addresses to the organization only (i.e., nginx, firewall rules, docker networking, Kubernetes network policies, etc.). However, they should also ensure a single sign-on integration (i.e., Keycloak) to authorize authenticated clients to interact with the service API. This way, the administrators ensure access control and management over the service access. If the service needs to scale up, apply global organizational policies, or interact with many other services, it should also integrate an API gateway. However, the API gateway itself does not prevent the service API from being publicly accessible.
36	Unnecessary privileges to microservices (Boersma, 2019; Carnell, 2017; Jain, 2018; da Silva, 2017; Matteson, 2017a; Kanjilal, 2020; Matteson, 2017b; Abasi, 2019; Behrens and Payne, 2016; Mody, 2020; Jackson, 2017; Lea, 2015)	Microservices pose unrequired privilege, granting unnecessary access levels, permissions, or functionalities that are actually not needed by such microservices to deliver their business functions.	An e-commerce application that consists of several microservices, including a Product Catalog service, a User Authentication service, and an Order Management service. Each microservice has its own specific responsibilities and access requirements. In this scenario, the Product Catalog service is responsible for managing the product information, such as adding new products, updating their details, and retrieving product data. The User Authentication service handles user registration, login, and authentication processes. Lastly, the Order Management service deals with processing customer orders and managing the order fulfillment workflow. However, due to an oversight or a lack of proper access control mechanisms, the Order Management service is granted unnecessary privileges. Specifically, it is given read access to the entire database of the Product Catalog service, including sensitive data like product pricing, supplier details, and internal notes. This granting of excessive privileges can pose several problems: Increased attack surface, data privacy concerns, and compliance issues (i.e., GDPR).

(*continued on next page*)

Table 13 (*continued*).

Index	Anti-pattern name	Description	Example
37	Insufficient access control (Troisi, 2017; Anon, 2019a; Krishnamurthy, 2018; Carnell, 2017; Anon, 2019b; Newman, 2016; Hofmann et al., 2017; Khan, 2018; Matteson, 2017b; Dias and Siriwardena, 2020; McLarty et al., 2018; Abasi, 2019; Doerfeld, 2015; Parecki, 2019; Smith, 2019; Gebel and Brossard, 2018; Nehme et al., 2019b; Indrasiri and Siriwardena, 2018b; Sharma, 2016; Wolff, 2016; Ziade, 2017; Lea, 2015; Nehme et al., 2019a; O'Neill, 2020; Raible, 2020)	The insufficient access control occurs whenever a microservice-based application does not enact access control in one or more of its microservices, hence potentially violating the confidentiality of the data and business functions of the microservices where access control is lacking.	A microservices architecture consisting of several services, including a user service and a financial service. The user service is responsible for managing user accounts and authentication, while the financial service handles financial transactions. In this scenario, the insufficient access control anti-pattern occurs when the financial service does not properly authenticate and authorize requests coming from the user service or any other services. ithout proper access control mechanisms in place, any service could potentially access and manipulate financial data without appropriate authorization. This lack of control opens up security vulnerabilities and increases the risk of unauthorized access or malicious activities.
38	Centralized authorization (Mannino, 2017; Anon, 2019a; Hofmann et al., 2017; Khan, 2018; Dias and Siriwardena, 2020; McLarty et al., 2018; Douglas, 2018; Perera, 2016; Nehme et al., 2019b; Yarygina and Bagge, 2018; Indrasiri and Siriwardena, 2018b; Jackson, 2017; Newman, 2015; Ziade, 2017; Richter et al., 2018; Nkomo and Coetzee, 2019; Nehme et al., 2019a; Siriwardena, 2020; Rajasekharaiah, 2020)	Service and authorization management is centralized, without implementing fine-grained authorization control at the level of individual microservices.	When all the authorization logic and decision-making are handled by a single central service or component. Such a service becomes a bottleneck. Also, it needs to have detailed knowledge about details normally encapsulated in decentralized microservices.

(*continued on next page*)

Table 13 (*continued*).

Index	Anti-pattern name	Description	Example
39	Non-secured service-to-service communications (Boersma, 2019; Carnell, 2017; Chandramouli, 2019; Anon, 2019b; Newman, 2016; Jain, 2018; Siriwardena, 2019; Pacheco, 2018; da Silva, 2017; Matteson, 2017a; Anon, 2020; Kanjilal, 2020; Matteson, 2017b; Dias and Siriwardena, 2020; McLarty et al., 2018; Gupta, 2018; Douglas, 2018; Sass, 2017; Mody, 2020; Smith, 2019; Lemos, 2019; Yarygina and Bagge, 2018; Esposito et al., 2016; Indrasiri and Siriwardena, 2018b; Sharma, 2016; Jackson, 2017; Wolff, 2016; Lea, 2015; Nkomo and Coetzee, 2019; Raible, 2020; Rajasekharaiah, 2020; Mateus-Coelho et al., 2020)	Service-to-service communication between two peers is not secured and encrypted. The transferred data can be exposed to man-in-the-middle, eavesdropping, and tampering attacks. Intruders could intercept the communication between two microservices and change the data in transit to their advantage. Secure channel such as Transport Layer Security (TLS) protocol should be in place to ensure peer authentication, data confidentiality, and integrity. The challenge is at the transport level.	Use of plain HTTP for inter-service communication without any encryption or authentication mechanisms in place.
40	Non-encrypted data exposure (Krishnamurthy, 2018; Boersma, 2019; Newman, 2016; Jain, 2018; Hofmann et al., 2017; da Silva, 2017; Dias and Siriwardena, 2020; Gupta, 2018; Smith, 2019; Jackson, 2017; Newman, 2015; Mateus-Coelho et al., 2020)	Service exposes plain data that is not encrypted, exposing sensitive information, e.g., because it was stored without any encryption in the data storage, or because the employed protection mechanisms are affected by security vulnerabilities or flaws. Unlike *non-secured service-to-service communications* with is at the transport level, this anti-pattern manifests at the message level.	A microservice-based e-commerce application handles customer orders. The application includes a payment microservice responsible for processing payment transactions. The payment microservice communicates with an external payment gateway to complete the payment process. In this example, the non-encrypted data exposure anti-pattern could occur if the communication between the payment microservice and the external payment gateway is not properly encrypted. The sensitive payment information, such as credit card details or bank account numbers, is transmitted in plain text over the network. Another example could be a user password persistently stored in plain text, which all administrators can access and see. Similarly, when the password is part of the user details entity, it might be loaded in plain text and exposed to other microservice middleware of the user interface.
41	Own crypto code (Troisi, 2017; Gardner, 2017; Sahni, 2020; Hofmann et al., 2017; Khan, 2018; da Silva, 2017; Lemos, 2019; Newman, 2015; O'Neill, 2020)	Custom encryption methods can expose to confidentiality, integrity, and authenticity of data in microservices, unless they have been heavily tested. The usage of standard and well-known crypto algorithms is always recommended.	Consider a system composed of multiple microservices that exchange sensitive information using encryption. In this scenario, each microservice decides to implement its own cryptographic code instead of utilizing a shared encryption library or service. Each microservice has its own implementation of encryption and decryption functions. This approach leads to the following consequences: Lack of standardization, increased vulnerability, maintenance overhead: complexity and duplication.

(*continued on next page*)

Table 13 (*continued*).

Index	Anti-pattern name	Description	Example
42	Hardcoded secrets (Mannino, 2017; Sahni, 2020; Jain, 2018; Hofmann et al., 2017; Khan, 2018; Sass, 2017; Parecki, 2019; Raible, 2020)	Configuration secrets are hardcoded in its source code, or in the deployment scripts for a microservice-based application, e.g., as environment variables passing secrets in a Dockerfile or a Docker Compose file, use technologies like HashiCorp Vault.	Embedding sensitive information, such as passwords, API keys, or database credentials, directly into the source code or configuration files of microservices. It is recommended to use secure and centralized methods for managing secrets, such as environment variables, configuration files, or secret management tools like HashiCorp Vault or AWS Secrets Manager.
		Team organization	
43	Shiny nickel (Jones, 2006; Taibi and Lenarduzzi, 2018; Dudney et al., 2003; Taibi et al., 2020b) (**AKA**, Silver bullet, Focus on latest technologies)	The newest technological craze is put into the system purely for publicity purposes. The latest technology buzz is incorporated into your system for the sake of telling people about it. It is often caused by soft procurement rules and a lack of a common IT strategy and vision. Often, product procurement takes place independently from projects, which leads to technology decisions on projects being driven by a desire to minimize shelfware.	A team must implement simple system analytics and adopt the current hype library (e.g., Tensor flow). They adopt it without considering the same feature might be implemented by standard libraries (e.g., Java.Math).
44	Golden hammer (Dudney et al., 2003; Rotem-Gal-Oz et al., 2012) (**AKA**, Same Old Way)	This anti-pattern occurs when familiar technologies are used as solutions to every problem. Many times this anti-pattern is perpetrated by individuals who have had past successes with a given technology, but are trying to use that technology to solve a problem that does not require the technology's existence.	A company decides to adopt microservices architecture for their application. They start by selecting a popular message broker, let us call it "XMQ," which is known for its scalability and performance in handling messaging between services. The development team becomes highly proficient in using XMQ and starts using it extensively for communication between microservices. However, as the application grows and evolves, they encounter new requirements that demand real-time streaming capabilities. They explore various options and find that "YStreamer" is a widely adopted and powerful streaming platform. But due to their heavy reliance on XMQ, they try to fit real-time streaming functionality into XMQ, even though it was not designed for that purpose.
45	Lack of communication standards among microservices (de Toledo et al., 2021, 2019)	When autonomous teams do not adopt common guidelines for communication among microservices, including the creation of APIs or message formats. Many APIs or message formats emerge from the various teams because each message producer of messages is left to define the format of the data themselves. This can lead to multiple issues, such as inconsistent data formats non-standardized communication protocols, increased complexity and maintenance, and reduced interoperability.	An e-commerce system with multiple microservices, such as a product catalog service, a shopping cart service, and an order processing service. In this scenario, the lack of communication standards can lead to various issues: Inconsistent data formats: Each microservice may use a different data format to represent similar information. For instance, the product catalog service might use JSON to describe products, while the shopping cart service might use XML. This inconsistency makes it difficult to share and process data seamlessly between microservices. Non-standardized communication protocols: Microservices may communicate with each other using different protocols, such as REST, gRPC, or messaging queues. Without a standardized protocol, integrating new microservices or changing existing ones becomes challenging. It requires additional effort to handle different communication mechanisms and understand how to interact with each service. Increased complexity and maintenance: When microservices lack communication standards, each service needs to implement custom logic to translate and adapt data between different formats and protocols. This additional complexity can lead to increased development effort, higher chances of introducing bugs, and more challenging maintenance tasks. Reduced interoperability: Without standardized communication, it becomes challenging to replace or upgrade individual microservices. If a service needs to be replaced or updated, the other services depending on it may need significant modifications to adapt to the new communication requirements.

(*continued on next page*)

Table 13 (*continued*).

Index	Anti-pattern name	Description	Example
46	Too many standards (de Toledo et al., 2021; Taibi and Lenarduzzi, 2018; de Toledo et al., 2019; Taibi et al., 2020b; Bogner et al., 2019c; Schirgi, 2021)	Multiple development languages, protocols, and frameworks are used. Although microservices allow the use of different technologies, adopting too many different ones can be a problem in organizations, especially in the event of developer turnover. This requires carefully considering any adoption of new technology, assuming different microservices already introduce broad heterogeneity. This term is used in the literature, but more often the term "standards" is also used (inaccurately) instead of "technologies", e.g. when REST, SOAP and GraphQL are used together). As the next antipattern,	A large e-commerce platform has adopted a microservices architecture. The platform consists of several microservices responsible for different functionalities like product catalog, user management, order processing, and inventory management. However, due to various factors like different development teams, evolving technology landscape, and changing requirements, each microservice has ended up adopting its own set of standards and technologies. In this scenario, you might observe the following manifestations of the "Too many standards among microservices" anti-pattern: Inconsistent communication protocols, diverse data storage mechanisms, varied authentication and authorization mechanisms, heterogeneous deployment strategies, incompatible development frameworks and languages (i.e., Java with Spring Boot, Python with Django, or Node.js with Express) that introduces additional complexity for developers who need to switch between various technologies and maintain expertise in multiple languages.
47	Inadequate techniques support (de Toledo et al., 2021; Carrasco et al., 2018a; Taibi et al., 2020b; Schirgi, 2021)	Use of inadequate techniques, which will not support the development of microservices. It refers to a situation where the development team lacks the necessary tools, processes, or skills to manage and operate microservices-based systems effectively. The term "inadequate" describes a situation where somebody wants to implement a certain requirement, there are different ways to do it, but the developer chooses (for various reasons) a method that is associated with disadvantages. This term is used in the literature, but unfortunately, it is imprecise and cannot be identified automatically (if the underlying requirement is unknown).	Poor choice of communication, such as message queues versus streaming, may lead to considerable latency for transferring messages among services impacting system responsiveness. Similarly, a poor integration choice can lead to consequent costs requiring a team to maintain a third-party solution and adjust it to a particular cloud environment instead of working on other priorities (i.e., adopting a content management system not originally meant for cloud deployment)
48	Single layer team (Carrasco et al., 2018b; Gehani, 2018; Golden B., 2018; Kalske et al., 2018; Lewis and Fowled, 2014; Nadareishvili et al., 2016; Taibi et al., 2017; Wolff, 2016; Carrasco et al., 2018a; Taibi et al., 2020b)	Division of teams by layer (e.g. Presentation, Business Layer, Persistence ses, etc.). This adds time and effort for approval whenever a change is needed. This leads to a ripple effect. Violates microservices principles, the independence of each service where teams are cross. To maximize the autonomy that microservices make possible, the governance of microservices should be decentralized and delegated to the teams that own the microservices themselves rather than specializing teams to particular layers.	Single team is responsible for developing, deploying, and maintaining all the microservices in a system. It goes against the principles of microservices architecture, which advocates for decentralized teams and autonomy.
49	No legacy (Kral and Zemlicka, 2007; Taibi and Lenarduzzi, 2018) (**AKA**, Everything Must Be New)	It might be a vision that the newly developed system does not contain any "obsolete" parts (legacy systems). In the service-oriented setting, this can become a costly anti-pattern. The main advantage of service-oriented systems is the enabled integration of autonomous systems, especially legacy systems. Getting rid of legacy systems causes superfluous immense additional investments into the development and implementation of new systems. It is good to leave selected older parts (legacy systems) in a new system as the old parts have useful capabilities and can be very stable (time-proven). One should choose the candidates for integration.	Consider an e-commerce company that decides to transition from a monolithic architecture to microservices to improve scalability and maintainability. They push for a complete system rewrite into microservices which is expensive. Instead of fully decomposing all parts of a monolithic system into independent and autonomous microservices, they should create selected microservices that integrate with the legacy system providing autonomy to all parts.

(*continued on next page*)

Table 13 (*continued*).

Index	Anti-pattern name	Description	Example
50	Data-driven migration (Richards, 2016)	The data-driven migration anti-pattern occurs when migrating from a monolithic application to a microservices architecture. It seems like a good idea at the start to migrate both the service functionality and the corresponding data together when creating microservices, but it leads down a bad path that can result in high-risk, excess cost, and additional migration effort. We will rarely get the granularity of each service right the first time, and it needs to further be split or consolidated produced microservices. In either case, we are faced with two migration efforts, one for the service functionality and another for the database. However, data is a corporate asset, not an application asset; data migrations are complex and error-prone—much more so than source code migrations. Optimally you want to migrate the data for each service only once. Instead, migrate the functionality of the service first, and worry about the bounded context between the service and the data later.	A company is migrating its monolith with a single, shared database to microservices. The company decides to split the database into multiple smaller databases, each dedicated to a specific microservice. The company schedules a maintenance window and performs the migration of all the data from the monolithic database to the new smaller databases in one go. During this migration, the entire system is unavailable. To avoid this data-driven migration anti-pattern, a more recommended approach is to adopt a phased or incremental migration strategy. This involves migrating services and their associated data incrementally, validating each step and ensuring that the system remains functional and available throughout the process.
51	Big bang (Král and Ž.emlicka, 2009)	An entire system is built "at once". It is often combined with the anti-pattern "No Legacy" where everything must be new or newly customized. It is the strategy preferred by large vendors, and it implies a strong dependency on the vendors (Vendor Lock-In). It can cause the services not to be independent enough and fine-grained. The (hidden) dependency further reduces the possibility of incremental development. Refactoring of Big Bang must include the use of coarse-grained user-oriented interfaces. To obtain highly autonomous services, the main tool is a proper decomposition of required capabilities.	E-commerce monolithic system consists of a single codebase and a tightly coupled database. The development team decides to refactor the system into a microservices architecture to achieve greater scalability, flexibility, and independent deployment. In the Big Bang anti-pattern scenario, the team decides to rewrite the entire application from scratch, breaking it down into separate microservices, all in one go. They stop all development and release activities for an extended period while the rewrite is in progress. Once the new microservices architecture is complete, they deploy it as a whole.
52	Multiple service instances per host (Balalaie et al., 2016, 2018; Carnell, 2017; Carrasco et al., 2018b; Dragoni et al., 2017; Indrasiri, 2017; Indrasiri and Siriwardena, 2018a; Jamshidi et al., 2018; Krause; C., 2018; Newman, 2015; Nygard, 2018; Savchenko et al., 2015; Soldani et al., 2018; Taibi et al., 2018; Zimmermann, 2017) (**AKA**, Multiple services in one container)	A single host contains multiple microservices instances deployed to the same host. The hints of the presence of this anti-pattern could be: (1) a single deployment platform; (2) a single version control repository; or (3) a global deployment script. Microservices have to share the same resources that are available inside the host. Moreover, scaling up or down a given host involves scaling all the instances that are inside this host. Finally, possible technology-related conflicts may happen between the microservices instances that share the same host.	An e-commerce application that consists of three microservices: Order Management, Inventory Management, and Payment Processing. Initially, the development team decides to deploy each microservice on separate hosts to ensure isolation and scalability. However, as the application grows, they start facing increased traffic and performance challenges. To address these challenges, the team decides to deploy multiple instances of each microservice on a single host. For instance, they run two instances of Order Management, three instances of Inventory Management, and four instances of Payment Processing on a single physical machine. This approach appears to utilize the available resources more efficiently, as multiple services can share the same hardware. However, over time, following issues are found: Resource contention as all instances experience a spike in traffic simultaneously and they will compete for CPU, memory, etc. Lack of isolation and fault tolerance. Monitoring and troubleshooting becomes challenging to isolate and diagnose issues.
53	No CI/CD (Bucchiarone et al., 2020)	The company does not employ CD/CI tools and developers need to manually test and deploy the system.	Instead of continuously integrating and deploying individual microservices, all the services are bundled together into a monolithic release and deployed as a single unit.
54	Manual configuration (de Toledo et al., 2021; Schirgi, 2021; Carnell and Sánchez, 2021)	Configuration of instances, services, and hosts is done manually. Microservices should separate the core codebase from the configuration management to enable automation.	If we had 1000 Microservice instances deployed and had to manually update a port for the database, it would be difficult to update in production manually. Therefore a configuration file for each Microservice is not the best solution. Instead, a configuration server should be used, which automates the configuration process. A possible solution is to completely separate the configuration of an application from the actual code being deployed, build immutable application images that never change as these are promoted through environments, and finally inject any application configuration information at server startup through either environment variables or a centralized repository that the microservices read on startup.

(*continued on next page*)

Table 13 (*continued*).

Index	Anti-pattern name	Description	Example
55	Insufficient monitoring (Schirgi, 2021; Bucchiarone et al., 2020)	Performance and failure of the microservices are not tracked. Failures become more difficult to catch and tracking performance issues become more tedious. A solution is to adopt a global monitoring tool.	An e-commerce application consists of multiple microservices, including a product catalog service, a shopping cart service, and a payment service. The services communicate with each other to fulfill customer orders. In this case, the development team has implemented the microservices architecture without giving much thought to monitoring and observability. They rely on simple logging statements within each service but lack a centralized monitoring system. As a result, they encounter several issues: lack of service health insights, difficulty in identifying root causes, inability to scale effectively, and reactive approach to issue resolution.
56	Dismiss documentation (Carrasco et al., 2018a)	In case of a lack of documentation of the exposed APIs, the overview of the system can be easily lost. When implementing microservices with multiple teams, cooperation could be hindered without proper documentation.	A company is transitioning from a monolithic architecture to a microservices architecture. The development team is excited about the flexibility and scalability benefits of microservices and starts building services independently. However, they underestimate the significance of documentation. In this case, the team fails to create comprehensive and up-to-date documentation for their microservices. They consider it a low-priority task and believe that the code itself should be self-explanatory. As a result: lack of dependency and purpose visibility of each microservice, onboarding difficulties, maintenance challenges, and reduced scalability (hard to identify the performance bottlenecks or understand the implications of scaling a particular service).
57	Insufficient message traceability (de Toledo et al., 2021; Schirgi, 2021)	When messages contain insufficient metadata, developers might find it difficult to track the messages' source. As an example, if Service A delivers messages through a message bus and no traceability metadata is available; Service B consumes a message from the message bus without knowing which service produced the message.	An e-commerce platform consisting of multiple microservices, such as a product catalog service, an inventory management service, and an order processing service. The communication between these services is done through message-based interactions. In this system, there is insufficient message traceability, which means that the flow of messages is not adequately tracked and monitored
58	Local logging (Taibi et al., 2020b; Schirgi, 2021; Bucchiarone et al., 2020)	Each microservice writes its logs to local storage, instead of using a distributed centralized logging system. Local logs can be very difficult to aggregate and analyze. This slows down the monitoring process proportionally to the number of microservices and log size.	When each microservice independently manages its own logging system without any centralized log aggregation or monitoring.

References

Abdelfattah, A.S., Cerný, T., Taibi, D., Vegas, S., 2023. Comparing 2D and augmented reality visualizations for microservice system understandability: A controlled experiment. ArXiv arXiv:2303.02268.

Al Maruf, A., Bakhtin, A., Cerny, T., Taibi, D., 2022. Using microservice telemetry data for system dynamic analysis. In: 2022 IEEE International Conference on Service-Oriented System Engineering (SOSE). IEEE, pp. 29–38.

Ampatzoglou, A., Bibi, S., Avgeriou, P., Verbeek, M., Chatzigeorgiou, A., 2019. Identifying, categorizing and mitigating threats to validity in software engineering secondary studies. Inf. Softw. Technol. (ISSN: 0950-5849) 106, 201–230.

Anon, 2007. Centre for Reviews and Dissemination, What are the criteria for the inclusion of reviews on DARE?.

Bogner, J., Boceck, T., Popp, M., Tschechlov, D., Wagner, S., Zimmermann, A., 2019a. Towards a collaborative repository for the documentation of service-based antipatterns and bad smells. In: 2019 IEEE International Conference on Software Architecture Companion. (ICSA-C), IEEE, pp. 95–101.

Bogner, J., Fritzsch, J., Wagner, S., Zimmermann, A., 2021. Industry practices and challenges for the evolvability assurance of microservices. Empir. Softw. Eng. 26 (5), 1–39.

Bogner, J., Weller, A., Wagner, S., Zimmermann, A., 2020. Exploring maintainability assurance research for service and micro-service-based systems: Directions and differences. In: Joint Post-Proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019). Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Brown, W., Brown, W., Malveau, R., McCormick, H., Mowbray, T., 1998. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. In: ITPro collection, Wiley.

Burgess, K., Hart, D., Elsayed, A., Cerny, T., Bures, M., Tisnovsky, P., 2022. Visualizing architectural evolution via provenance tracking: A systematic review. In: Proceedings of the Conference on Research in Adaptive and Convergent Systems. RACS '22, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450393980, pp. 83–91. http://dx.doi.org/10.1145/3538641.3561493.

Bushong, V., Das, D., Cerny, T., 2022. Reconstructing the holistic architecture of microservice systems using static analysis. In: Proceedings of the 12th International Conference on Cloud Computing and Services Science - CLOSER. SciTePress, INSTICC, (ISSN: 2184-5042) ISBN: 978-989-758-570-8, pp. 149–157. http://dx.doi.org/10.5220/0011032100003200.

Carnell, J., Sánchez, I.H., 2021. Spring Microservices in Action. Simon and Schuster.

Cerny, T., Abdelfattah, A.S., Bushong, V., Al Maruf, A., Taibi, D., 2022a. Microservice architecture reconstruction and visualization techniques: A review. In: 2022 IEEE International Conference on Service-Oriented System Engineering. (SOSE), pp. 39–48. http://dx.doi.org/10.1109/SOSE55356.2022.00011.

Cerny, T., Abdelfattah, A.S., Bushong, V., Al Maruf, A., Taibi, D., 2022b. Microvision: Static analysis-based approach to visualizing microservices in augmented reality. In: 2022 IEEE International Conference on Service-Oriented System Engineering (SOSE). pp. 49–58. http://dx.doi.org/10.1109/SOSE55356.2022.00012.

Costal, D., Farr'e, C., Franch, X., Quer, C., 2021. How tertiary studies perform quality assessment of secondary studies in software engineering. In: Conferencia Iberoamericana de Software Engineering.

Das, D., Walker, A., Bushong, V., Svacina, J., Cerny, T., Matyas, V., 2021. On automated RBAC assessment by constructing a centralized perspective for microservice mesh. PeerJ Comput. Sci. 7, e376.

de Andrade, H.S., Almeida, E., Crnkovic, I., 2014a. Architectural bad smells in software product lines: An exploratory study. In: Proceedings of the WICSA 2014 Companion Volume. In: WICSA '14 Companion, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450325233, http://dx.doi.org/10.1145/2578128.2578237.

Fontana, F.A., Pigazzini, I., Roveda, R., Tamburri, D., Zanoni, M., Di Nitto, E., 2017. Arcan: A tool for architectural smells detection. In: 2017 IEEE International Conference on Software Architecture Workshops. (ICSAW), pp. 282–285. http://dx.doi.org/10.1109/ICSAW.2017.16.

Garcia, J., Popescu, D., Edwards, G., Medvidovic, N., 2009a. Identifying architectural bad smells. In: 2009 13th European Conference on Software Maintenance and Reengineering. pp. 255–258. http://dx.doi.org/10.1109/CSMR.2009.59.

García-Mireles, G.A., Morales-Trujillo, M.E., 2020. Gamification in Software Engineering: A Tertiary Study. In: Mejia, J., Muñoz, M., Rocha, A., A. Calvo-Manzano, J. (Eds.), Trends and Applications in Software Engineering. Springer International Publishing, Cham, pp. 116–128.

Guo, D., Wu, H., 2021. A review of bad smells in cloud-based applications and microservices. In: 2021 International Conference on Intelligent Computing, Automation and Systems. (ICICAS), IEEE, pp. 255–259.

Hoda, R., Salleh, N., Grundy, J., Tee, H.M., 2017. Systematic literature reviews in agile software development: A tertiary study. Inf. Softw. Technol. (ISSN: 0950-5849) 85, 60–70.

Ibrahim, A., Bozhinoski, S., Pretschner, A., 2019. Attack graph generation for microservice architecture. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing. SAC '19, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450359337, pp. 1235–1242. http://dx.doi.org/10.1145/3297280.3297401.

Junior, H.J., Travassos, G.H., 2022. Consolidating a common perspective on Technical Debt and its Management through a Tertiary Study. Inf. Softw. Technol. (ISSN: 0950-5849) 149, 106964.

Kendall, J., 1999. Axial coding and the grounded theory controversy. West. J. Nurs. Res. 21 (6), 743–757.

Kitchenham, B.A., Madeyski, L., Budgen, D., 2022. SEGRESS: Software Engineering Guidelines for Reporting Secondary Studies. IEEE Trans. Softw. Eng. 1.

Lattner, C., Adve, V., 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004. IEEE, pp. 75–86.

Lelovic, L., Mathews, M., Abdelfattah, A., Cerny, T., 2023. Microservices architecture language for describing service view. In: Proceedings of the 13th International Conference on Cloud Computing and Services Science - CLOSER. SciTePress, INSTICC, (ISSN: 2184-5042) ISBN: 978-989-758-650-7, pp. 220–227. http://dx.doi.org/10.5220/0011850200003488.

Lewis, J., Fowler, M., 2014. Microservices. www.martinfowler.com/articles/microservices.html, Accessed: January 2023.

de Oliveira Rosa, T., Daniel, J.F.L., Guerra, E.M., Goldman, A., 2020. A method for architectural trade-off analysis based on patterns: Evaluating microservices structural attributes. In: Proceedings of the European Conference on Pattern Languages of Programs 2020. EuroPLoP '20, Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/3424771.3424809.

Osses, F., Márquez, G., Astudillo, H., 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 '18, Association for Computing Machinery, New York, NY, USA, pp. 256–257. http://dx.doi.org/10.1145/3183440.3194958.

Raatikainen, M., Tiihonen, J., Männistö, T., 2019. Software product lines and variability modeling: A tertiary study. J. Syst. Softw. (ISSN: 0164-1212) 149, 485–510.

Rademacher, F., Sachweh, S., Zündorf, A., 2020. A modeling method for systematic architecture reconstruction of microservice-based software systems. In: Nurcan, S., Reinhartz-Berger, I., Soffer, P., Zdravkovic, J. (Eds.), Enterprise, Business-Process and Information Systems Modeling. Springer International Publishing, Cham, pp. 311–326.

Refactoring.Guru, 2023. Shotgun surgery. https://refactoring.guru/smells/shotgun-surgery, [Online; accessed 9-June-2023].

Schiewe, M., Curtis, J., Bushong, V., Cerny, T., 2022. Advancing static code analysis with language-agnostic component identification. IEEE Access 10, 30743–30761. http://dx.doi.org/10.1109/ACCESS.2022.3160485.

Schipper, D., Aniche, M., van Deursen, A., 2019. Tracing back log data to its log statement: From research to practice. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories. (MSR), pp. 545–549. http://dx.doi.org/10.1109/MSR.2019.00081.

solo.io, 2022. 2022 Service Mesh Adoption Survey. https://lp.solo.io/service-mesh-adoption-survey.

Stocker, M., Zimmermann, O., Lübke, D., Zdun, U., Pautasso, C., 2018. Interface Quality Patterns – Communicating and Improving the Quality of Microservices APIs. In: 23rd European Conference on Pattern Languages of Programs 2018.

Strauss, A., Corbin, J., 1998. Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory. SAGE Publications, ISBN: 9780803959408, URL https://books.google.fi/books?id=wTwYUnHYsmMC.

Svacina., J., Bushong., V., Das., D., Cerny., T., 2022. Semantic code clone detection method for distributed enterprise systems. In: Proceedings of the 12th International Conference on Cloud Computing and Services Science - CLOSER. SciTePress, INSTICC, (ISSN: 2184-5042) ISBN: 978-989-758-570-8, pp. 27–37. http://dx.doi.org/10.5220/0011032200003200.

Taibi, D., Lenarduzzi, V., Pahl, C., 2020a. Microservices Anti-patterns: A Taxonomy. In: Bucchiarone, A., Dragoni, N., Dustdar, S., Lago, P., Mazzara, M., Rivera, V., Sadovykh, A. (Eds.), Microservices: Science and Engineering. Springer International Publishing, Cham, pp. 111–128.

Tighilt, R., Abdellatif, M., Trabelsi, I., Madern, L., Moha, N., Guéhéneuc, Y.-G., 2023. On the maintenance support for microservice-based systems through the specification and the detection of microservice antipatterns. J. Syst. Softw. (ISSN: 0164-1212) 111755. http://dx.doi.org/10.1016/j.jss.2023.111755, URL https://www.sciencedirect.com/science/article/pii/S0164121223001504.

Tran, H.K.V., Unterkalmsteiner, M., Börstler, J., bin Ali, N., 2021. Assessing test artifact quality—A tertiary study. Inf. Softw. Technol. (ISSN: 0950-5849) 139, 106620.

van Emden, E., Moonen, L., 2002. Java quality assurance by detecting code smells. In: Ninth Working Conference on Reverse Engineering, 2002. Proceedings. pp. 97–106. http://dx.doi.org/10.1109/WCRE.2002.1173068.

Verner, J., Brereton, O., Kitchenham, B., Turner, M., Niazi, M., 2014. Risks and risk mitigation in global software development: A tertiary study. Inf. Softw. Technol. (ISSN: 0950-5849) 56 (1), 54–78, Special sections on International Conference on Global Software Engineering – August 2011 and Evaluation and Assessment in Software Engineering – April 2012.

Vidal, S., Oizumi, W., Garcia, A., Díaz Pace, A., Marcos, C., 2019. Ranking architecturally critical agglomerations of code smells. Sci. Comput. Program. (ISSN: 0167-6423) 182, 64–85. http://dx.doi.org/10.1016/j.scico.2019.07.003, URL https://www.sciencedirect.com/science/article/pii/S0167642318303514.

Walker, A., Das, D., Cerny, T., 2020. Automated Code-Smell Detection in Microservices Through Static Analysis: A Case Study. Appl. Sci. (ISSN: 2076-3417) 10 (21).

Wikipedia contributors, 2023. Shotgun surgery — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Shotgun_surgery&oldid=1159278192, [Online; accessed 9-June-2023].

Wimmer, C., 2021. Graalvm native image: Large-scale static analysis for java (keynote). In: Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages. In: VMIL 2021, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450391092, p. 3. http://dx.doi.org/10.1145/3486606.3488075.

Wohlin, C., 2014. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering. EASE '14, Association for Computing Machinery, New York, NY, USA.

Xu, B., Qian, J., Zhang, X., Wu, Z., Chen, L., 2005. A brief survey of program slicing. SIGSOFT Softw. Eng. Notes (ISSN: 0163-5948) 30 (2), 1–36. http://dx.doi.org/10.1145/1050849.1050865.

Yamashita, A., Moonen, L., 2013. Do developers care about code smells? An exploratory survey. In: 2013 20th Working Conference on Reverse Engineering. (WCRE), pp. 242–251. http://dx.doi.org/10.1109/WCRE.2013.6671299.

Zhao, X., Zhang, Y., Lion, D., Ullah, M.F., Luo, Y., Yuan, D., Stumm, M., 2014. Lprof: A non-intrusive request flow profiler for distributed systems. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. OSDI '14, USENIX Association, USA, ISBN: 9781931971164, pp. 629–644.

Zimmermann, O., 2022. Microservice API patterns. https://www.microservice-api-patterns.org/, Accessed: 2022-02-04.

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

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

Zimmermann, O., Stocker, M., Lübke, D., Pautasso, C., Zdun, U., 2019. Introduction to Microservice API Patterns (MAP). In: International Conference on Microservices (Microservices 2019).

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

Secondary studies

Bogner, Justus, Boceck, Tobias, Popp, Matthias, Tschechlov, Dennis, Wagner, Stefan, Zimmermann, Alfred, 2019b. Toward a collaborative repository for the documentation of service-based antipatterns and bad smells. In: 2019 IEEE International Conference on Software Architecture Companion. (ICSA-C), IEEE, pp. 95–101.

Ding, Xiang, Zhang, Cheng, 2022. How Can We Cope with the Impact of Microservice Architecture Smells? In: 2022 11th International Conference on Software and Computer Applications. pp. 8–14.

Mumtaz, Haris, Singh, Paramvir, Blincoe, Kelly, 2021. A systematic mapping study on architectural smells detection. J. Syst. Softw. 173, 110885.

Neri, Davide, Soldani, Jacopo, Zimmermann, Olaf, Brogi, Antonio, 2020. Design principles, architectural smells and refactorings for microservices: A multivocal review. SICS Softw.-Intensive Cyber-Phys. Syst. 35 (1), 3–15.

Ponce, Francisco, Soldani, Jacopo, Astudillo, Hernán, Brogi, Antonio, 2022. Smells and refactorings for microservices security: A multivocal literature review. J. Syst. Softw. 111393.

Sabir, Fatima, Palma, Francis, Rasool, Ghulam, Guéhéneuc, Yann-Gaël, Moha, Naouel, 2019. A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems. Softw. - Pract. Exp. 49 (1), 3–39.

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: Proceedings of the European Conference on Pattern Languages of Programs 2020. pp. 1–13.

Primary studies

Abasi, Farshad, 2019. Securing modern API- and microservices-based apps by design. URL https://developer.ibm.com/articles/securing-modern-api-and-microservices-apps-1/.

Alagarasan V., 2015. Seven microservices anti-patterns. InfoQ. URL https://www.infoq.com/articles/seven-uservices-antipatterns.

Alshuqayran, Nuha, Ali, Nour, Evans, Roger, 2016. A systematic mapping study in microservice architecture. In: 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications. (SOCA), IEEE, pp. 44–51.

Anon, 2019a. Improving security in your microservices architecture. URL https://www.sumologic.com/insight/microservices-architecture-security/.

Anon, 2019b. Microservices security: Best practices to secure microservicess. URL https://hindi.ava360.com/microservices-security-best-practices-to-secure-microservices-edureka_e39d9d108.html.

Anon, 2019c. Shift to microservices: Evolve your security practices & container security. URL https://lab.wallarm.com/shift-to-microservices-evolve-your-security-practices-container-security/.

Anon, 2020. Microservice architectures challenge traditional security practices. URL https://blog.radware.com/security/2020/01/microservice-architectures-challenge-traditional-security-practices/.

Azadi, U., Fontana, F.A., Taibi, D., 2019. Architectural smells detected by tools: A catalog proposal. In: Proceedings - 2019 IEEE/ACM International Conference on Technical Debt, TechDebt 2019. pp. 88–97. http://dx.doi.org/10.1109/TechDebt.2019.00027.

Balalaie, A., Heydarnoori, A., Jamshidi, P., 2016. Microservices architecture enables devops: migration to a cloud-native architecture. IEEE Softw. 33, http://dx.doi.org/10.1109/MS.2016.64.

Balalaie, A., Heydarnoori, A., Jamshidi, P., Tamburri, D.A., Lynn, T., 2018. Microservices migration patterns. Softw. Pract. Exp. 48, http://dx.doi.org/10.1002/spe.2608.

Behrens, Scott, Payne, Bryan, 2016. Starting the avalanche: Application ddos in microservice architectures. URL https://netflixtechblog.com/starting-the-avalanche-640e69b14a06?gi=51345aa9d068.

Bhojwani, R., 2018. Design patterns for microservices. URL https://dzone.com/articles/design-patterns-for-microservices.

Boersma, Eric, 2019. Top 10 security traps to avoid when migrating from a monolith to microservices. URL https://blog.sqreen.com/top-10-security-traps-to-avoid-when-migrating-from-a-monolith-to-microservices/.

Bogner, Justus, Fritzsch, Jonas, Wagner, Stefan, Zimmermann, Alfred, 2019c. Assuring the evolvability of microservices: insights into industry practices and challenges. In: 2019 IEEE International Conference on Software Maintenance and Evolution. (ICSME), IEEE, pp. 546–556.

Bogner, J., Fritzsch, J., Wagner, S., Zimmermann, A., 2019d. Microservices in industry: Insights into technologies, characteristics, and software quality. In: Proceedings - 2019 IEEE International Conference on Software Architecture - Companion. ICSA-C 2019, pp. 187–195. http://dx.doi.org/10.1109/ICSA-C.2019.00041.

Bonér, J., 2016. Reactive Microservice Architecture: Design Principles for Distributed Systems. O'Reilly, Newton.

Brogi, Antonio, Neri, Davide, Soldani, Jacopo, 2019. Freshening the air in microservices: resolving architectural smells via refactoring. In: International Conference on Service-Oriented Computing. Springer, pp. 17–29.

Bucchiarone, Antonio, Dragoni, Nicola, Dustdar, Schahram, Lago, Patricia, Mazzara, Manuel, Rivera, Victor, Sadovykh, Andrey, 2020. Microservices. Sci. Eng. Springer.

Budko, Renata, 2018. Five things you need to know about API security. URL https://thenewstack.io/5-things-you-need-to-know-about-api-security/.

C., Meléndez, 2018. 7 container design patterns you need to know. TechBeacon. URL https://techbeacon.com/enterprise-it/7-container-design-patterns-you-need-know.

Carnell, J., 2017. Spring Microservices in Action. Manning Publications Co., New York.

Carrasco, Andrés, Bladel, Brent van, Demeyer, Serge, 2018a. Migrating toward microservices: migration and architecture smells. In: Proceedings of the 2nd International Workshop on Refactoring. pp. 1–6.

Carrasco, Andrés, Bladel, Brent van, Demeyer, Serge, 2018b. Migrating toward microservices: migration and architecture smells. In: Proceedings of the 2nd International Workshop on Refactoring. pp. 1–6.

Chandramouli, R., 2019. Security strategies for microservices-based application systems. https://csrc.nist.gov/publications/detail/sp/800-204/final.

Cortellessa, V., Di Marco, A., Trubiani, C., 2014. An approach for modeling and detecting software performance antipatterns based on first-order logics. Softw. Syst. Model. 13 (1), 391–432. http://dx.doi.org/10.1007/s10270-012-0246-z.

Coscia, Jos'e, Mateos, Cristian, Crasso, Marco, Zunino, Alejandro, 2012. Avoiding WSDL bad practices in code-first web services. SADIO Electron. J. Inform. Oper. Res. 11.

da Silva, Rodrigo Candido, 2017. Best practices to protect your microservices architecture. URL https://medium.com/@rcandidosilva/best-practices-to-protect-your-microservices-architecture-541e7cf7637f.

Dall, R., 2016. Performance patterns in microservices-based integrations. URL https://dzone.com/articles/performance-patterns-in-microservices-based-integr-1.

de Andrade, Hugo Sica, Almeida, Eduardo, Crnkovic, Ivica, 2014b. Architectural bad smells in Software Product Lines: An exploratory study. In: ACM International Conference Proceeding Series. p. 12.

de Toledo, Saulo Soares, Martini, Antonio, Przybyszewska, Agata, Sjøberg, Dag IK, 2019. Architectural technical debt in microservices: A case study in a large company. In: 2019 IEEE/ACM International Conference on Technical Debt. (TechDebt), IEEE, pp. 78–87.

de Toledo, Saulo S., Martini, Antonio, Sjøberg, Dag I.K., 2021. Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study. J. Syst. Softw. 177, 110968.

Dias, Wajjakkara Kankanamge Anthony Nuwan, Siriwardena, Prabath, 2020. Microservices Security in Action. Simon and Schuster.

Doerfeld, Bill, 2015. How to control user identity within microservices. URL https://nordicapis.com/how-to-control-user-identity-within-microservices/.

Douglas, Michael, 2018. Microservices authentication & authorization best practice. URL https://codeburst.io/i-believe-it-reallydepends-on-your-environment-and-how-well-protected-the-different-piecesare-7919bfa6bc86.

Dragoni, Nicola, Giallorenzo, Saverio, Lafuente, Alberto Lluch, Mazzara, Manuel, Montesi, Fabrizio, Mustafin, Ruslan, Safina, Larisa, 2017. Microservices: Yesterday, today, and tomorrow. In: Mazzara, Manuel, Meyer, Bertrand (Eds.), Present and Ulterior Software Engineering. Springer International Publishing, Cham, ISBN: 978-3-319-67425-4, pp. 195–216. http://dx.doi.org/10.1007/978-3-319-67425-4_12.

Dudney, Bill, Asbury, Stephen, Krozak, Joseph K, Wittkopf, Kevin, 2003. J2EE Antipatterns. John Wiley & Sons.

Esposito, C., Castiglione, A., Choo, K.-K.R., 2016. Challenges in delivering software in the cloud as microservices. IEEE Cloud Comput. 3 (5), 10–14. http://dx.doi.org/10.1109/MCC.2016.105.

Francesco, P., Lago, P., Malavolta, I., 2019. Architecting with microservices: A systematic mapping study. J. Syst. Softw. 150, http://dx.doi.org/10.1016/j.jss.2019.01.001.

Francesco, Paolo Di, Malavolta, Ivano, Lago, Patricia, 2017. Research on architecting microservices: Trends, focus, and potential for industrial adoption. In: 2017 IEEE International Conference on Software Architecture. (ICSA), pp. 21–30. http://dx.doi.org/10.1109/ICSA.2017.24.

Furda, A., Fidge, C., Zimmermann, O., Kelly, W., Barros, A., 2018. Migrating enterprise legacy source code to microservices: on multitenancy, statefulness, and data consistency. IEEE Softw. 35, http://dx.doi.org/10.1109/MS.2017.440134612.

Garcia, Joshua, Popescu, Daniel, Edwards, George, Medvidovic, Nenad, 2009b. Toward a catalog of architectural bad smells. In: Mirandola, Raffaela, Gorton, Ian, Hofmeister, Christine (Eds.), Architectures for Adaptive Software Systems. Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN: 978-3-642-02351-4, pp. 146–162.

Garcia, Joshua, Popescu, Daniel, Edwards, George, Medvidovic, Nenad, 2009c. Toward a Catalog of Architectural Bad Smells. In: Proceedings of the 5th International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems. QoSA '09, Springer-Verlag, East Stroudsburg, PA, USA, pp. 146–162.

Gardner, Z., 2017. Security in the microservices paradigm. URL https://keyholesoftware.com/2017/03/13/security-in-the-microservices-paradigm/.

Gebel, Gerry, Brossard, David, 2018. Securing APIs and microservices with oauth, openid connect, and ABAC. URL https://curity.io/resources/videos/securing-apis-and-microservices-with-oauth-and-openid-connect/.

Gehani, Neil, 2018. Want to develop great microservices? Reorganize your team. TechBeacon. URL https://techbeacon.com/app-dev-testing/want-develop-great-microservices-reorganize-your-team, Accessed 5 June 2019.

Golden B., 2017. 5 fundamentals to a successful microservice design. TechBeacon. URL https://techbeacon.com/app-dev-testing/5-fundamentals-successful-microservice-design.

Golden B., 2018. Creating a microservice: design first, code later. TechBeacon. URL https://techbeacon.com/app-dev-testing/creating-microservice-design-first-code-later.

Gupta, Natasha, 2018. Security strategies for devops, apis, containers and microservices. URL https://cyware.com/news/security-strategies-for-devops-apis-containers-and-microservices-blog-imperva-b68775a4.

Hofmann, Michael, Schnabel, Erin, Stanley, Katherine, et al., 2017. Microservices Best Practices for Java. IBM Redbooks.

Indrasiri, K., 2017. Microservices in practice: from architecture to deployment. URL https://dzone.com/articles/microservices-in-practice-1.

Indrasiri, K., Siriwardena, P., 2018a. Microservices for the Enterprise: Designing, Developing, and Deploying. A Press, Berkeley, http://dx.doi.org/10.1007/978-1-4842-3858-5.

Indrasiri, Kasun, Siriwardena, Prabath, 2018b. Microservices security fundamentals. In: Microservices for the Enterprise. Springer, pp. 313–345.

Jackson, Nic, 2017. Building Microservices with Go.

Jain, Chintan, 2018. URL https://appsecusa2017.sched.com/event/B2Xh/top-10-security-best-practices-to-secure-your-microservices.

Jamshidi, P., Pahl, C., Mendonca, N., Lewis, J., Tilkov, S., 2018. Microservices: the journey so far and challenges ahead. IEEE Softw. 35, http://dx.doi.org/10.1109/MS.2018.2141039.

Jones, Steve, 2006. SOA Anti-Patterns.

Kalske, M., Mäkitalo, N., Mikkonen, T., 2018. Challenges when moving from monolith to microservice architecture. In: Garrigós, I., Wimmer, M. (Eds.), Current Trends in Web Engineering. Springer, Berlin, http://dx.doi.org/10.1007/978-3-319-74433-9_3.

Kamaruzzaman, M., 2020. Microservice architecture and its 10 most important design patterns. URL https://towardsdatascience.com/microservice-architecture-and-its-10-most-important-design-patterns-824952d7fa41.

Kanjilal, Joydip, 2020. 4 fundamental microservices security best practices. URL https://www.techtarget.com/searchapparchitecture/tip/4-fundamental-microservices-security-best-practices.

Khan, Arif, 2018. How to secure your microservices: Shopify case study. URL https://dzone.com/articles/bountytutorial-microservices-security-how-to-secur.

Kitchenham, B., Charters, S., 2007. Guidelines for performing systematic literature reviews in software engineering. Guidel. Perform. Syst. Lit. Rev. Softw. Eng..

Knoche, H., Hasselbring, W., 2018. Using microservices for legacy software modernization. IEEE Softw. 35, http://dx.doi.org/10.1109/MS.2018.2141035.

Král, Jaroslav, Žemlicka, Michal, 2009. Popular SOA Antipatterns. In: 2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns. pp. 271–276.

Kral, Jaroslav, Zemlicka, Michal, 2007. The Most Important Service-Oriented Antipatterns. In: International Conference on Software Engineering Advances. ICSEA 2007, pp. 29–29.

Krause, L., Microservices: Patterns and applications: Designing fine-grained services by applying patterns, Lucas Krause, 2015.

Krishnamurthy, Thribhuvan, 2018. Transition to microservice architecture - challenges. Transition Microservice Architecture - Chall..

Le, Duc Minh, Link, Daniel, Shahbazian, Arman, Medvidovic, Nenad, 2018a. An empirical study of architectural decay in open-source software. In: 2018 IEEE International Conference on Software Architecture. (ICSA), pp. 176–17609. http://dx.doi.org/10.1109/ICSA.2018.00027.

Le, D.M., Link, D., Shahbazian, A., Medvidovic, N., 2018b. An empirical study of architectural decay in open-source software. In: Proceedings - 2018 IEEE 15th International Conference on Software Architecture. ICSA 2018, pp. 176–185. http://dx.doi.org/10.1109/ICSA.2018.00027.

Lea, G., 2015. Microservices security: all the questions you should be asking. URL https://www.grahamlea.com/2015/07/microservices-security-questions/.

Lemos, Robert, 2019. App security in the microservices age: 4 best practices. URL https://techbeacon.com/app-dev-testing/app-security-microservices-age-4-best-practices.

Lewis, J., Fowled, M., 2014. Microservices: A definition of this new architectural term. ThoughtWorks. https://www.martinfowler.com/articles/microservices.html, Accessed 5 June 2019.

Long J., 2015. The power, patterns, and pains of microservices. DZone. URL https://dzone.com/articles/the-power-patterns-and-pains-of-microservices.

Mannino, Jack, 2017. Security in the land of microservices. URL https://www.youtube.com/watch?v=JRmWlLY8MGE.

Marinescu, R., 2004. Detection strategies: Metrics-based rules for detecting design flaws. In: IEEE International Conference on Software Maintenance. ICSM, pp. 350–359. http://dx.doi.org/10.1109/ICSM.2004.1357820.

Marinescu, R., 2005. Measurement and quality in object-oriented design. In: IEEE International Conference on Software Maintenance, ICSM, Vol. 2005. pp. 701–704. http://dx.doi.org/10.1109/ICSM.2005.63.

Marinescu, R., Rajiu, D., 2004. Quantifying the quality of object-oriented design: The factor-strategy model. In: Proceedings - Working Conference on Reverse Engineering. WCRE, pp. 192–201. http://dx.doi.org/10.1109/WCRE.2004.31.

Mateos, C., Rodriguez, J.M., Zunino, A., 2015. A tool to improve code-first web services discoverability through text mining techniques. Softw. - Pract. Exp. 45 (7), 925–948. http://dx.doi.org/10.1002/spe.2268.

Mateus-Coelho, Nuno, Cruz-Cunha, Manuela, Ferreira, Luis Gonzaga, 2020. Security in microservices architectures. In: CENTRIS Conference. pp. 1–12.

Matteson, S., 2017a. 10 Tips for securing microservice architecture. URL https://www.techrepublic.com/article/10-tips-for-securing-microservice-architecture/.

Matteson, S., 2017b. How to establish strong microservices security using SSL, TLS, and API gateways. URL https://www.techrepublic.com/article/how-to-establish-strong-microservice-security-using-ssl-tls-and-api-gateways/.

McLarty, Matt, Wilson, Rob, Morrison, Scott, 2018. Securing Microservices APIs. OReilly.

Mody, Virag, 2020. From zero to zero trust. URL https://www.forescout.com/blog/from-zero-to-zero-trust-five-tips-to-simplify-your-journey/.

Moha, N., Guéhéneuc, Y.-G., Duchien, L., Le Meur, A.-F., 2010. DECOR: A method for the specification and detection of code and design smells. IEEE Trans. Softw. Eng. 36 (1), 20–36. http://dx.doi.org/10.1109/TSE.2009.50.

Nadareishvili, I., Mitra, R., McLarty, M., Amundsen, M., 2016. Microservice Architecture: Aligning Principles, Practices, and Culture. O'Reilly, Newton.

Nayrolles, Mathieu, Moha, Naouel, Valtchev, Petko, 2013. Improving SOA antipatterns detection in Service Based Systems by mining execution traces. In: Reverse Engineering (WCRE), 2013 20th Working Conference on. pp. 321–330.

Nehme, A., Jesus, V., Mahbub, K., Abdallah, A., 2019a. Fine-Grained Access Control for Microservices. In: Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 11358 LNCS, pp. 285–300. http://dx.doi.org/10.1007/978-3-030-18419-3_19.

Nehme, A., Jesus, V., Mahbub, K., Abdallah, A., 2019b. Securing microservices. IT Prof. 21 (1), 42–49. http://dx.doi.org/10.1109/MITP.2018.2876987.

Newman, S., 2015. Building Microservices. O'Reilly, Newton.

Newman, Sam, 2016. Security and microservices. URL https://samnewman.io/talks/appsec-and-microservices/.

Nkomo, P., Coetzee, M., 2019. Software Development Activities for Secure Microservices. In: Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 11623 LNCS, pp. 573–585. http://dx.doi.org/10.1007/978-3-030-24308-1_46.

Nygard, Michael, 2018. Release it!: design and deploy production-ready software. Release It! 1–376.

Oizumi, Willian N., Garcia, Alessandro F., Colanzi, Thelma E., Ferreira, Manuele, Staa, Arndt V., 2015. On the relationship of code-anomaly agglomerations and architectural problems. J. Softw. Eng. Res. Dev. 3 (1), 11. http://dx.doi.org/10.1186/s40411-015-0025-y.

O'Neill, Leon, 2020. Microservice security - what you need to know. URL https://crashtest-security.com/microservice-security-what-you-need-to-know/.

Ordiales Coscia, J.L., Mateos, C., Crasso, M., Zunino, A., 2013. Anti-pattern free code-first web services for state-of-the-art java WSDL generation tools. Int. J. Web Grid Serv. 9 (2), 107–126. http://dx.doi.org/10.1504/IJWGS.2013.054108.

Ordiales Coscia, J.L., Mateos, C., Crasso, M., Zunino, A., 2014. Refactoring code-first web services for early avoiding WSDL anti-patterns: Approach and comprehensive assessment. Sci. Comput. Program. 89 (PART C), 374–407. http://dx.doi.org/10.1016/j.scico.2014.03.015.

Ouni, Ali, Gaikovina Kula, Raula, Kessentini, Marouane, Inoue, Katsuro, 2015. Web Service Antipatterns Detection Using Genetic Programming. In: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation. GECCO '15, ACM, Madrid, Spain, pp. 1351–1358.

Ouni, A., Kessentini, M., Inoue, K., Cinnéide, M.Ó., 2017. Search-Based Web Service Antipatterns Detection. IEEE Trans. Serv. Comput. 10 (4), 603–617.

Pacheco, Vinicius Feitosa, 2018. Microservice Patterns and Best Practices: Explore Patterns Like CQRS and Event Sourcing To Create Scalable, Maintainable, and Testable Microservices. Packt Publishing Ltd.

Palma, F., An, L., Khomh, F., Moha, N., Gueheneuc, Y.-G., 2014a. Investigating the change-proneness of service patterns and antipatterns. In: Proceedings - IEEE 7th International Conference on Service-Oriented Computing and Applications. SOCA 2014, pp. 1–8. http://dx.doi.org/10.1109/SOCA.2014.43.

Palma, F., Moha, N., Gueheneuc, Y.-G., 2019. UniDoSA: The unified specification and detection of service antipatterns. IEEE Trans. Softw. Eng. 45 (10), 1024–1053. http://dx.doi.org/10.1109/TSE.2018.2819180.

Palma, Francis, Moha, Naouel, Tremblay, Guy, Guéhéneuc, Yann-Gaël, 2014b. Specification and detection of SOA antipatterns in web services. In: Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 8627 LNCS, pp. 58–73.

Palma, Francis, Moha, Naouel, Tremblay, Guy, Guéhéneuc, Yann-Gaël, 2014c. Specification and detection of SOA antipatterns in web services. In: Avgeriou, Paris, Zdun, Uwe (Eds.), Software Architecture. Springer International Publishing, Cham, ISBN: 978-3-319-09970-5, pp. 58–73.

Palma, Francis, Mohay, Naouel, 2015. A study on the taxonomy of service antipatterns. In: Patterns Promotion and Anti-Patterns Prevention (PPAP), 2015 IEEE 2nd Workshop on. pp. 5–8.
Palma, F., Nayrolles, M., Moha, N., Guéhéneuc, Y.-G., Baudry, B., Jézéquel, J.-M., 2013. SOA antipatterns: An approach for their specification and detection. Int. J. Coop. Inf. Syst. 22 (4), http://dx.doi.org/10.1142/S0218843013410049.
Parecki, A., 2019. Oauth: When things go wrong. URL https://www.okta.com/blog/2019/04/oauth-when-things-go-wrong/.
Perera, Srinath, 2016. Walking the wire: Mastering the four decisions in microservices architecture. URL https://softwareengineeringdaily.com/2018/12/17/walking-the-wire-mastering-the-four-decisions-in-microservices-architecture/.
Pigazzini, Ilaria, Fontana, Francesca Arcelli, Lenarduzzi, Valentina, Taibi, Davide, 2020. Toward microservice smells detection. In: Proceedings of the 3rd International Conference on Technical Debt. pp. 92–97.
Raible, Matt, 2020. Security patterns for microservice architectures. https://springone.io/2020/sessions/security-patterns-for-microservice-architectures.
Rajasekharaiah, Chandra, 2020. Cloud-Based Microservices: Techniques, Challenges, and Solutions. Springer.
Richards, Mark, 2016. Microservices AntiPatterns and Pitfalls. p. 66.
Richardson, Chris, 2014. Microservices: Decomposing applications for deployability and scalability. InfoQ 25, 15–16.
Richardson, C., 2018. Microservices Patterns. Manning Publications, New York.
Richter, D., Neumann, T., Polze, A., 2018. Security considerations for microservice architectures. In: CLOSER 2018 - Proceedings of the 8th International Conference on Cloud Computing and Services Science, Vol. 2018-January. pp. 608–615. http://dx.doi.org/10.5220/0006791006080615.
Rotem-Gal-Oz, Arnon, Bruno, Eric, Dahan, Udi, 2012. SOA Patterns. Manning.
Ruecker, B., 2019. 3 common pitfalls of microservices integration and how to avoid them. InfoWorld. URL https://www.infoworld.com/article/3254777/3-common-pitfalls-of-microservices-integrationand-how-to-avoid-them.html.
Sahni, V., 2020. Best practices for building a microservice architecture. URL https://www.vinaysahni.com/best-practices-for-building-a-microservice-architecture.
Saleh, T., 2016. Microservices antipatterns. URL https://www.infoq.com/presentations/cloud-anti-patterns.
Sass, R., 2017. Security in the world of microservices. URL https://www.itpro.com/technology/artificial-intelligence-ai/368107/what-good-ai-cyber-security-software-looks-like/.
Savchenko, Dmitry I., Radchenko, Gleb I., Taipale, Ossi, 2015. Microservices validation: Mjolnirr platform case study. In: 2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics. (MIPRO), IEEE, pp. 235–240.
Schirgi, Thomas, 2021. Architectural quality attributes for the microservices of care. URL https://courses.isds.tugraz.at/rkern/courses/sa/supplemental/microservices-of-care.pdf.
Sharma, Sourabh, 2016. Mastering Microservices with Java. Packt Publishing Ltd.
Siriwardena, P., 2019. Microservices security landscape. URL https://medium.facilelogin.com/microservices-security-landscape-7b396b3b03ea.
Siriwardena, Prabath, 2020. Challenges of securing microservices. URL https://www.styra.com/blog/security-challenges-in-microservices/.
Smith, Tom, 2017. How do you secure microservices?. URL https://dzone.com/articles/how-do-you-secure-microservices.
Smith, Tom, 2019. How to secure APIs. URL https://dzone.com/articles/how-to-secure-apis.
Soldani, J., Tamburri, D.A., Heuvel, W.J., 2018. The pains and gains of microservices: A systematic gray literature review. J. Syst. Softw. 146, http://dx.doi.org/10.1016/j.jss.2018.09.082.
Taibi, Davide, Lenarduzzi, Valentina, 2018. On the Definition of Microservice Bad Smells. IEEE Softw. 35 (3), 56–62.
Taibi, D., Lenarduzzi, V., Pahl, C., 2017. Processes, motivations, and issues for migrating to microservices architectures: an empirical investigation. IEEE Cloud Comput. 4, http://dx.doi.org/10.1109/MCC.2017.4250931.
Taibi, Davide, Lenarduzzi, Valentina, Pahl, Claus, 2018. Architectural patterns for microservices: A systematic mapping study. In: CLOSER 2018: Proceedings of the 8th International Conference on Cloud Computing and Services Science; Funchal, Madeira, Portugal, 19-21 March 2018. SciTePress.
Taibi, Davide, Lenarduzzi, Valentina, Pahl, Claus, 2020b. Microservices anti-patterns: A taxonomy. In: Microservices. Springer, pp. 111–128.
Toledo, Saulo S. de, Martini, Antonio, Sjøberg, Dag I.K., 2020. Improving agility by managing shared libraries in microservices. In: International Conference on Agile Software Development. Springer, pp. 195–202.
Troisi, Marco, 2017. 8 best practices for microservices app sec. URL https://techbeacon.com/app-dev-testing/8-best-practices-microservices-app-sec.
Vidal, Santiago, Guimaraes, Everton, Oizumi, Willian, Garcia, Alessandro, Pace, Andrés Díaz, Marcos, Claudia, 2016. Identifying architectural problems through prioritization of code smells. In: 2016 X Brazilian Symposium on Software Components, Architectures and Reuse. (SBCARS), pp. 41–50. http://dx.doi.org/10.1109/SBCARS.2016.11.
Wolff, E., 2016. Microservices: Flexible Software Architecture. Addison-Wesley, Reading.
Yarygina, T., Bagge, A.H., 2018. Overcoming security challenges in microservice architectures. In: Proceedings - 12th IEEE International Symposium on Service-Oriented System Engineering, SOSE 2018 and 9th International Workshop on Joint Cloud Computing. JCC 2018, pp. 11–20. http://dx.doi.org/10.1109/SOSE.2018.00011.
Ziade, Tarek, 2017. Python Microservices Development: Build, Test, Deploy, and Scale Microservices in Python. Packt Publishing Ltd.
Zimmermann, O., 2017. Microservices tenets. Comput. Sci. Res. Dev. 32, http://dx.doi.org/10.1007/s00450-016-0337-0.

Tomas Cerny is a Professor of Computer Science at Baylor University. His area of research is software engineering, cloud systems, and code analysis. He received his Master's, and Ph.D. degrees from the Faculty of Electrical Engineering at the Czech Technical University in Prague, and an M.S. degree from Baylor University.

He started his academic career in 2009 at the Czech Technical University, FEE, from where he transferred to Baylor University in 2017. Dr. Cerny served 10+ years as the lead developer of the International Collegiate Programming Contest Management System. He authored over 100 publications, mostly related to code analysis and enterprise systems. Among his awards are best papers at Microservices 2022, IEEE SOSE 2022, Closer 2022, LXNLP 2022, the Outstanding Service Award ACM SIGAPP 2018 and 2015, or the 2011 ICPC Joseph S. DeBlasi Outstanding Contribution Award. He served on the committee of multiple conferences in the past few years, including program or conference chairs at ACM SAC, ACM RACS, and ICITCS.

Andrea Janes senior lecturer at FHV Vorarlberg University of Applied Sciences (Austria) and adjunct professor at University of Oulu (Finland). His research activity is related to the area of software maintenance and development. In particular, his research involves the identification of cost-efficient software production techniques, quality assurance methodologies, as well as the application of foundational aspects of software engineering methods such as testing and software process improvement. He received the master's in computer science from the Technical University of Vienna, Austria and the doctorate in computer science (with distinction) from the University of Klagenfurt (Austria). He was a visiting researcher at the Research Center Hagenberg (Austria) and the Tampere University (Finland). He was an assistant professor at the Free University of Bolzano-Bozen (Italy). He served as a program committee member of various international conferences and as a reviewer for various international journals (e.g., TSE, EMSE, JSS, and IST) in the field of software engineering. He has been Doctoral Symposium co-chair of PROFES 2022, short papers and poster co-chair of EASE 2023, program co-chair of PROFES 2023, Journal First and Special Issue chair of QUATIC2023, and industrial papers co-chair of SANER 2024. He organized several workshops and events for practitioners focused on the application of research in industry. Since 2017, he is also involved in technology transfer within Smart Data Factory, a group within the NOI technology park with the goal of technology transfer within the local industry.

Davide Taibi is full Professor at the University of Oulu (Finland) where he head the M3S Cloud research group. His research is mainly focused on Empirical Software Engineering applied to cloud-native systems, with a special focus on the migration from monolithic to cloud-native applications. He is investigating processes, and techniques for developing Cloud Native applications, identifying cloud-native specific patterns and anti-patterns. He is member of the International Software Engineering Network (ISERN) from 2018. Before moving to Finland, he has been Assistant Professor at the Free University of Bozen/Bolzano (2015–2017), post-doctoral research fellow at the Technical University of Kaiserslautern and Fraunhofer Institute for Experimental Software Engineering - IESE (2013–2014) and research fellow at the University of Insubria (2007–2011).