This is a post-peer-review, pre-copyedit version of an article published in Journal of Systems and Software. The final authenticated version is available online at: https://doi.org/10.1016/j.jss.2020.110885

A Systematic Mapping Study on Architectural Smells Detection

Haris Mumtaz, Paramvir Singh, and Kelly Blincoe

Abstract—The recognition of the need for high-quality software architecture is evident from the increasing trend in investigating architectural smells. Detection of architectural smells is paramount because they can seep through to design and implementation stages if left unidentified. Many architectural smells detection techniques and tools are proposed in the literature. The diversity in the detection techniques and tools suggests the need for their collective analysis to identify interesting aspects for practice and open research areas. To fulfill this, in this paper, we unify the knowledge about the detection of architectural smells through a systematic mapping study. We report on the existing detection techniques and tools for architectural smells to identify their limitations. We find there has been limited investigation of some architectural smells (e.g., micro-service smells); many architectural smells are not detected by tools yet; and there are limited empirical validations of techniques and tools. Based on our findings, we suggest several open research problems, including the need to 1) investigate undetected architectural smells (e.g., Java package smells), 2) improve the coverage of architectural smell detection across architecture styles (e.g., service-oriented and cloud), and 3) perform empirical validations of techniques and tools in industry across different languages and project domains.

Index Terms—Architectural smells, architectural debt, antipatterns, smell detection techniques, systematic mapping study

1 Introduction

Software architecture forms the foundation and defines the structural and semantic composition of a software system [8]. A software system's architecture directly influences the software artifacts created in the subsequent development stages (e.g., design and implementation) [32]. Therefore, it is paramount that the quality of software architecture is not compromised. Still, it is common for the software architecture quality to be degraded because of the adoption of certain design decisions. For instance, a decision to create a centralized component that handles the majority of the responsibilities can lead to modularization problems in the architecture [46]. Such inappropriate decisions introduce "bad smells" in the architecture, negatively impacting the architecture quality, mainly maintainability [10,41]. The term "smell" refers to a structural problem in a software artifact. For instance, "code smells" refer to structural problems in source code [25]. In the literature, the term "smell" is interchangeably used with other terms, such as antipattern, flaw, anomaly, etc. [P38,P39,P62,P73]. In the rest of this paper, we use the term "smells" to refer the structural design problems in software architecture. However, while describing the techniques and tools, for consistency reason, we use the same terms (e.g., antipattern, flaw, etc.) as mentioned in the papers.

Based on the scope and impact of bad smells, they are often classified into three levels of granularity in the order: architecture (high-level), design, and implementation (low-level) [42]. Smells at the implementation level refer to the structural problems in the low-level constructs (functions or methods) such as Long Method, Long Parameter List, etc. [25]. At the design-level, structural problems in the classes, such as Missing Abstraction and Insufficient Modularization, are observed [46]. In contrast to implementation and design smells, the architecture level smells indicate the structural problems in the components (which could be groups of classes) and their interactions with other components. Examples of architectural smells are God Component and Dense Structure [P3]. The scope of this paper is limited to the analysis of architectural smells detection techniques and tools.

- *H. Mumtaz and K. Blincoe are with the Department of Electrical, Computer, and Software Engineering at the University of Auckland, New Zealand. P. Singh is with the School of Computer Science at the University of Auckland, New Zealand. E-mail: {hmum126}@aucklanduni.ac.nz, {p.singh,k.blincoe}@auckland.ac.nz.*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

The timely detection and eradication of architectural smells are essential because, otherwise, the cost of fixing the repercussions later in the software development life cycle is exceptionally high, especially in large scale industrial projects [P58]. Over the past decade, several techniques and tools have been developed to detect architectural smells. These techniques and tools focus on various architectural styles (e.g., service, layered, etc.), architectural smells (e.g., dependency, performance, etc.), and types of software (e.g., web, middleware, etc.). As a specific example, Ouni et al. [P49] identified service-oriented smells that impact the maintainability of web systems. Likewise, Sanctis et al. [P13] detected performance smells in software architecture. The techniques and tools are validated, mostly, in two ways (through empirical studies and case studies) using a variety of data (from open source projects and commercial projects). For instance, Velasco-Elizondo et al. [P74] conducted an empirical validation of their detection technique for model-view-controller (MVC) architectural smells on open source web systems.

This demonstrates that the detection techniques and tools are diverse, and a systematic and comprehensive analysis of the available techniques and tools would help the research community learn about the interesting aspects and limitations. Currently, the information about architectural smells detection is spread across multiple literature databases, and to the best of our knowledge, there is no literary evidence of systematic gathering and discussion of such information.

To fill this gap, in this paper, we present a systematic mapping study on the detection of architectural smells. The objective is to unify the scattered knowledge about architectural smells detection techniques and tools into one literary source to analyze what has been accomplished in this area, highlight useful findings, and reflect on the limitations of what is available. As expected from a mapping study, we also report on publication trends of the architectural smells detection. To fulfill these objectives, we formulate the following research questions:

- RQ1: What are the demographics of the published articles?
  Goal – This RQ identifies **when** the articles are published; the venues **where** research related to architectural smells detection is published; and the **origin** type (academic or industry) of these articles.

- RQ2: What detection techniques for architectural smells are proposed in literature?
  Goal – This RQ identifies and analyzes the architectural smells detection techniques that have been proposed in the literature. We investigate what types of architectural smells can currently be detected; what architecture styles are handled by the techniques; and how the techniques are evaluated.

© 2020. This manuscript version is made available under the CC-BY-NC-ND 4.0 license http://creativecommons.org/licenses/by-nc-nd/4.0/

- RQ3: What detection tools for architectural smells are proposed and evaluated in literature?
  Goal – This RQ identifies and analyzes the detection tools that have been reported in the literature. The purpose of RQ3 is to highlight the architecture smell detection tools that are available to the software development community. Software tools make detection techniques more accessible to software practitioners, but not all detection techniques have supporting tools. Therefore, we also analyze detection tools separate to detection techniques. Similar to RQ2, we report on the architectural smells that tools can detect. We also report on the validation, availability, and language support of the detection tools.

- RQ4: What are the limitations of the reported detection techniques and tools?
  Goal – This RQ identifies limitations reported in the proposed detection techniques and tools. The limitations are comprised of the ones found in the literature and identified through the collective analysis of the detection techniques and tools. Through this RQ, we identify undetected architectural smells, understudied architecture styles, and unexplored quality characteristics. This RQ also reflects on the limitations in the evaluation processes of the techniques and tools.

We present the systematic mapping process that defines our search strategy, inclusion criteria, and exclusion criteria using the guidelines of Petersen et al. [38, 39]. We developed a search string to retrieve the literature relevant to our research questions. We applied the search string to seven digital databases: Scopus, Web of Science, INSPEC, ACM Digital Library, IEEEXplore, SpringerLink, and DBLP. As a result, we retrieved 529 unique articles, which were reduced to 76 after applying the inclusion and exclusion criteria. We also performed snowballing on the selected articles using the guidelines of Wohlin [51]. After snowballing, the final list of articles (85) was selected for analysis. To analyze the gathered literature, we formalized an analysis framework comprising of several factors related to techniques, tools, and their validation. In addition, for an in-depth analysis, we matched the architectural smells that are detected by techniques and tools with a list of all architectural smells obtained from recent papers which catalog currently known architectural smells [5, 13, 17, 26, 28, 31, 45, 47]. This enabled us to identify known architectural smells that are not currently detected by any techniques or tools.

The main contribution of this paper is the systematic unification of the information about architectural smells detection (techniques and tools) that is scattered across various digital databases and libraries. We present a comprehensive analysis of architectural smells detection techniques and tools with a focus on highlighting the interesting findings, gaps, and limitations. We also discuss open research areas that need exploration and investigation.

2 Related Work

Before describing our study, we provide an overview of related systematic reviews and mapping studies.

De Paulo Sobrinho et al. [18] conducted a literature review to cover five W's: which, when, what, who, and where on bad smells in different software artifacts (code, design, and architecture). They found that some kinds of smells are studied more frequently than others (which); that research in the area of bad smells is spread across time (when); that findings, claims, and experimental setups vary among different detection approaches (what); who the authors are that publish (continuously or sporadically) in the research area related to bad smells (who); and finally, which venues mostly publish the research about bad smells (where). They reported on the investigation of various bad smells in different software artifacts and at different granularity levels. Through their intensive analysis, they presented future directions for research in the area of bad smells. Compared to this study, they did not present details on architectural smells techniques and tools.

There are some literature reviews and mapping studies on architectural technical debt (ATD). Besker et al. [11, 12] performed a systematic literature review on ATD. Their focus was on the debt, interest, and principle related to the effort and cost required to deal with ATD. They also proposed a model to categorize the characteristics of ATD better. Alves et al. [3] identified several indicators for detecting technical debt in software architecture through a systematic mapping study. Ampatzoglou et al. [4] performed a systematic literature review to cover the financial aspects of technical debt management. Li et al. [30] published a mapping study on the management of technical debt. Verdecchia et al. [50] applied a systematic mapping study to identify, classify, and evaluate studies that identify ATD. They focused on analyzing the publication trends, characteristics, and industry involvement in ATD identification. Tom et al. [48] conducted a literature review to uncover the nature of technical debt and its implications on the software development process. Finally, the goal of Fernández-Sánchez et al. [22] was to identify and analyze the key elements (related to the cost estimation and decision making) for managing technical debt. Generally, ATD is closely related to architectural smells, however, none of these studies investigated architecture smell detection specifically.

In a systematic literature review, Sabir et al. [41] evaluated smell detection techniques and their evolution within the scope of object-oriented and service-oriented systems. They identified various commonalities and differences in the detection of bad smells in object-oriented and service-oriented systems. They reported several key findings, including some smells that receive less attention and a catalog of service-oriented smells that require more investigation. Vale et al. [49] presented a systematic literature review on the existence of bad smells in the software product line (SPL). They concluded that research on SPL-specific smells is an open area to explore, considering many limitations and challenges discussed in the literature. They also provided a catalog of code smells, architectural smells, and hybrid smells. The systematic mapping study performed by Bandi et al. [6] studied the empirical evaluations of techniques that investigate the effects of code decay on software quality. They also included design and architectural violations related to code decay and software quality. Although these reviews reported interesting findings and limitations about architectural smells detection, the scopes are limited to specific architectural styles or domains.

There are also some reviews on the detection of design-level smells—sub-optimal patterns in software design [43]. Mumtaz et al. [35] conducted a literature survey to identify gaps in the detection techniques for design smells. They analyzed the proposed techniques and experimental designs to locate limitations related to unexplored design smells and experimental evaluations. They found a scarcity of detection approaches for UML sequence diagrams and use cases. Similarly, Misbhauddin and Alshayeb [34] provided a systematic review of existing research in UML model refactoring. Alkharabsheh et al. [2] analyzed the detection approaches for design smells using several comparison factors. The comparison factors include smell type, detection techniques, detection tools, validation, artifact type, and language support. They also explored the relationship between the detected design smells and quality attributes. There is also a systematic review that compared bad smells detection tools in terms of bad smells coverage, language support, and usability issues [21]. The scope of these reviews is limited to design smells. However, the comparison factors are relevant for our analysis.

Azadi et al. [P3] presented a review of architectural smells detection tools. They identified the architectural smells detected by various tools and presented a catalog of architectural smells. In their review, they also reported three quality principles (modularity, hierarchy, and health dependency structure) violated by the architectural smells. Furthermore, they discussed the differences between the detection strategies used by the tools to detect each smell. From a tool's perspective, our work is different from Azadi et al. [P3] in that we mainly focus on highlighting the limitations by analyzing the detection tools. From our analysis, we identified many architectural smells (e.g., performance-related smells) that are not detected by currently available tools. We also provided details of the validations of the detection tools and presented the limitations related to the validation. In addition, we reflected on the language support provided by the tools and discussed some issues related to

the language scalability. Furthermore, we discussed various ways the limitations in the detection tools could be addressed. In addition to detection tools, we also analyze other detection techniques (and their limitations) which do not have associated tools.

To summarize, from the architecture point of view, the systematic reviews related to software architecture either focus on financial aspects or their scope is limited to a specific domain. The literature reviews and mapping studies that focused on the financial aspects of technical debt had research questions aimed at measuring and managing the maintenance cost of the changes due to the existence of the technical debt. Some literature reviews had scope limited to a specific domain or paradigm, such as the reviews on the bad smells detection approaches for object-oriented and service-oriented systems. There is also a systematic literature review that reported on the studies investigating only those smells that belong to the software product line. To the best of our knowledge, there is no systematic review or mapping study that collectively covers both detection techniques and tools for architectural smells. We fill this gap with a systematic mapping study, where we analyze the detection techniques and tools with a broader scope of architectural smells detection. The systematic reviews and mapping studies at the design level used a variety of comparison factors (e.g., smell type, artifact type, domain, etc.) to evaluate the detection approaches. In our systematic mapping study, we also use similar analysis factors to compare the architectural smells detection techniques and tools.

3 Systematic Mapping Process

In this paper, we followed the systematic mapping guidelines provided by Petersen et al. [38, 39]. The first author executed the systematic mapping process, however, the results were iteratively discussed between the authors. Our mapping process contained three steps: planning, execution, and analysis. At the planning stage, we identified research questions and a mapping study protocol. The rationale of the mapping protocol was to formulate a method to execute the mapping process. In the execution phase, the searched literature was exposed to the inclusion and exclusion criteria to identify a set of articles for in-depth analysis. Lastly, at the results analysis stage, the selected literature was subjected to analysis to address the research questions. The systematic mapping process is also depicted as a flow chart in Figure 1.

3.1 Mapping Protocol Overview

We used the following mapping protocol that was discussed and finalized between the authors:

- Search the databases and digital libraries from 1999 to 2019 inclusive using the search string (see Section 3.2)—retrieved 837 articles.
- Remove duplicates (automatically and manually) to obtain unique articles (see Section 3.2)—resulted in 529 articles.
- Apply inclusion and exclusion criteria (see Section 3.3)—resulted in 76 articles.
- Apply snowballing to 76 articles (see Section 3.4)—identified 27 new articles.
- Apply inclusion and exclusion criteria to 27 articles identified through snowballing (see Section 3.4)—removed 18 articles.
- Extract data from the selected articles (85) that was needed to answer our research questions.

3.2 Search Strategy

Searched databases – We searched the following databases and digital libraries because they are commonly used to extract computer science and software engineering publications: Scopus, Web of Science, INSPEC, ACM Digital Library, IEEEXplore, SpringerLink, and DBLP [14]. We searched these databases and libraries to obtain information related to the detection of architectural smells in October 2019.

Search string – We formulated the following search string to locate the related work: ***"software architectur*" AND ("smell*" OR "antipattern*" OR "debt" OR "flaw*" OR "anomal*")***. The included search terms were derived based on our research questions. First, the term "software architectur*" was included to get the literature confined to software architecture. Then, to narrow the software architecture literature to the papers discussing architectural smells, we included the term "smell*". We also considered different terms that are used in the same context as smells in the literature. For instance, Sharma and Spinellis [43] reported the use of the terms "smell" and "antipattern" interchangeably by software engineering researchers and practitioners. Similarly, other terms ("debt", "flaw*", and "anomal*") were also used in the same context as smells [P60, P73]. Therefore, we included these terms in our search string. Moreover, our search string used conjunction and disjunction operators to combine multiple search terms into a single search string. The specific search string used in each database is shown in Table 1.

Search string validation – To see the effectiveness of our search string, we performed a focused search analysis of the papers published in the *International Conference on Software Engineering (ICSE)* from 2016 to 2019. The reason for performing this exercise was to gain confidence in the search string. We chose ICSE because it is the flagship software engineering conference. We selected the four ICSE editions (2016–2019) because they are the most recent editions and represent a reasonably sized set of papers for manual validation. We manually looked at all the papers in these editions to shortlist the relevant ones (for comparison later). By reading through the abstracts and introductions of all papers from the proceedings of ICSE 2016–2019, we identified 15 papers of interest. By applying the search string within ICSE 2016–2019, we found 12 of the identified relevant papers (80%). The search string missed three papers (20%), two from ICSE '19 and one from ICSE '18. Subsequently, we executed a snowballing strategy on the 12 retrieved articles. As a result, we found the remaining papers that were previously missed by our search string. To conclude, 80% (12 out of 15) of the articles were found through our search string, and the remaining three papers were located through snowballing. Thus, we were confident that the search string combined with a snowballing strategy has good accuracy in finding the papers of interest.

Retrieved articles – We received 837 hits in total by applying the search string in the databases mentioned earlier. The number of retrieved articles respective to each electronic database is shown in Table 1. After removing the duplicated articles, the number of unique articles was reduced to 529. These 529 articles were exposed to our inclusion and exclusion criteria described in the next section.

3.3 Inclusion and Exclusion Criteria

We defined the inclusion and exclusion criteria to filter the set of articles—from the unique retrieved articles (529)—that were relevant to our research objectives and questions. The rationale of inclusion criteria was to consider articles that substantially discuss architectural smells detection. In contrast, the purpose of defining the exclusion criteria was to discard articles containing information that is insufficient for answering our research questions.

We used the following inclusion criteria:

- Articles that discuss the detection of architectural smells.
- Articles that describe techniques or tools aiming to detect architectural smells.
- Articles that are written in English.
- Articles that are published 1999 onwards because the term "bad smell" was introduced in this year.
- We consider literature published in journals, conferences, books, and workshops.

The exclusion criterion was iteratively modified during the exclusion process. For instance, "briefings" was added to our exclusion criteria when a briefing report was found during the snowballing process. The list of final exclusion criteria is:

Table 1. Distribution of articles per electronic database

Database	Search string	Retrieved Articles
Scopus	“software architectur*” AND (“smell*” OR “antipattern*” OR “debt” OR “flaw*” OR “anomal*”)	115
Web of Science	“software architectur*” AND (“smell*” OR “antipattern*” OR “debt” OR “flaw*” OR “anomal*”)	78
INSPEC	“software architecture” AND (“smell*” OR “antipattern*” OR “debt” OR “flaw*” OR “anomal*”)	161
ACM Digital Library	“software architecture” AND (“smell*” OR “antipattern*” OR “debt” OR “flaw*” OR “anomal*”)	78
IEEEXplore	“software architectur*” AND (“smell*” OR “antipattern*” OR “debt” OR “flaw*” OR “anomal*”)	136
SpringerLink	“software architectur*” AND (“smell*” OR “antipattern*” OR “debt” OR “flaw*” OR “anomal*”)	244
DBLP	“software architectur*” (“smell*” | “antipattern*” | “debt” | “flaw*” | “anomal*”)	25
Total (includes duplication)		**837**
Total (unique articles)		**529**
Total (after inclusion and exclusion criteria)		**76**
Total (after snowballing)		**85**

- Articles that discuss software architecture and architectural smells (not detection) in general.
- Articles that define architectural smells or present only a catalog of architectural smells. Although such papers [5, 13, 17, 26, 28, 31, 45, 47] were excluded, we did use them to identify the architectural smells (including their aliases) that are detected by the techniques and tools and those smells that are not detected yet.
- Articles that only consider architectural refactoring, not architecture smell detection.
- Literature available in the form of interviews, news, posters, tutorials, and lectures.
- Articles that present proposed work, such as visionary reports, proposals, briefings, etc.
- Articles whose full text is not available.

The initial search in the digital databases was restricted to TAK (Title, Abstract, and Keywords). The titles and abstracts of the retrieved articles were reviewed to apply the inclusion and exclusion criteria. In some cases, where the relevance of the article was unclear from the titles and abstracts, we also read the introductions of the articles. After applying the inclusion and exclusion criteria to the articles (529), the number of articles was reduced to 76 (as shown in Table 1). These 76 articles became the candidates for the snowballing strategy described in the next section.

3.4 Snowballing

We used the guidelines of Wohlin [51] to perform snowball sampling to search the relevant literature that might have been missed by our search string. We performed one iteration of forward and backward snowballing. By applying this snowballing strategy to the selected 76 articles, we identified 27 new articles. Similar to the articles (retrieved by our search string), the newly found 27 articles were also subjected to our inclusion and exclusion criteria. After the inclusion and exclusion criteria were applied, an additional nine articles were added to the pool of 76 articles making the final set of papers in this study 85 articles (also shown in Table 1).

4 Analysis Framework

After obtaining the desired literature sources, we extracted the key factors related to the detection of architectural smells. In this section, we describe the factors constituting our analysis framework. The factors were extracted according to the guidelines provided by Petersen et al. [38, 39] and driven by our research questions. The first author extracted the data from the 85 primary papers. To ensure descriptive validity, the other two authors also independently extracted the data

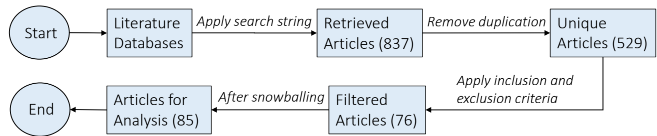

Fig. 1. Systematic Mapping Process

from just over 15% of the primary papers. The disagreements (and perceptual differences in the observations) in the data extraction were then discussed among all the authors to reach a common ground on how the data should be extracted and processed. Once all three authors agreed on a common extraction and classification scheme, the first author extracted the data from the rest (85%) of the primary papers.

The analysis framework was developed by considering both the content of the papers and our research questions. To analyze the detection techniques and tools, some key factors needed to be identified (e.g., what kind of architectural smells are detected; what types of detection techniques are proposed). To have a comprehensive set of factors, we considered factors that have been used in prior studies to analyze the detection of design smells [2, 35] and bad smells detection tools [21]. For the factors that are related to the evaluation of techniques, we also sought guidance from Jedlitschka et al. [27]. They reported some essential elements for an experimental setup in software engineering, which we also adopted with some variations (also used in [2, 21, 35]). Moreover, to see the prominent publication venues and involvement of industry in the detection of architectural smells, we also considered some demographics-related factors as a part of our analysis. Consequently, in our analysis framework, we divided the factors (with some overlap) into three categories: demographics-related (RQ1), technique-related (RQ2), and tool-related (RQ3). The analysis framework is depicted (with examples) in Figure 2. The category-wise explanation of the factors is provided in the rest of this section. Note that the “types” mentioned in each analysis factor were extracted from the primary papers. In the case of tool-related factors, where information is missing in the papers, we also extracted data from the official webpages of the tools.

Demographics-related factors – We used the following demographics-related factors to address our RQ1:

- **Publication years** – How many articles are published in each year.
  Types – 2010 to 2019.
- **Publication venue** – The venues in which the literature related to architectural smells detection is published.
  Types – Journal, conference, and workshop.

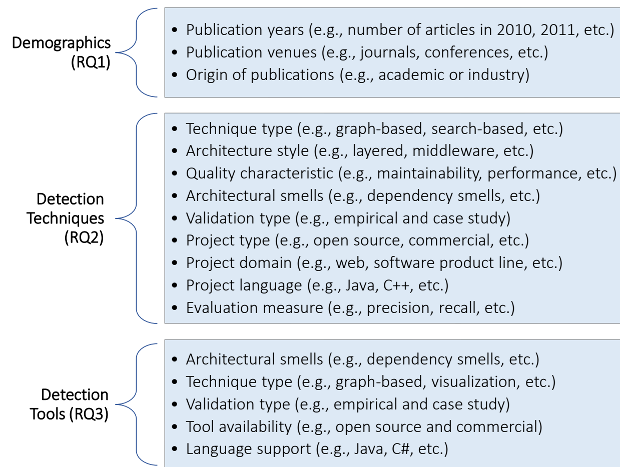

Fig. 2. Analysis framework factors with examples

- **Origin of publications** – The affiliation of the authors who published the articles related to architectural smells detection.
  Types – Academic and industry.

Technique-related factors – To answer RQ2, we formulated the following factors:

- **Technique type** – The foremost information that we looked at was the technique types presented in the related corpus. The main idea was to first identify what techniques were employed for detection and then formulate a classification to categorize the detection techniques. For instance, if a technique used a graphical representation to locate undesired dependencies between components of the architecture [P27], we classified the technique under the graph-based approach. To find such information, we read through the proposed technique sections of the articles.
  Types – Rules-based, graph-based, design structure matrix, model-driven, code smells analysis, reverse engineering and history-based, search-based, visualization, and others.

- **Architecture style** – The literature investigated different architecture styles while detecting the smells. For instance, some articles detected architectural smells for MVC architecture [P26]. To collect such information, we scanned for the architecture styles in the articles.
  Types – Service-oriented architecture, Model-View-Controller, layered, component, cloud, client-server, C-language architecture, Java EE architecture, Android architecture, and aspect-oriented architecture.

- **Quality characteristic** – The literature investigated some architectural quality characteristics. We map the articles to quality characteristics based on two criteria. First, if a primary paper specifically investigated a quality characteristic. Second, if the primary paper linked the detected architectural smells with a specific quality characteristic. For instance, many articles detected those architectural smells that impact the maintainability aspects of software systems. We used the quality characteristics chart specified in the ISO Standard (ISO25010) quality model [1].
  Types – Maintainability, performance, and security.

- **Architectural smells** – The articles focused on detecting a variety of architectural smells making smells another paramount element of our analysis framework.
  Types – Service-oriented, performance, dependency, package, MVC-related, component, and other smells. The architectural smells that fall into these categories are listed in Table 14. There is no existing study that presents a comprehensive classification of architectural smells, and that is not the goal of this study since we are analyzing only the subset of architectural smells that are currently detected by techniques or tools. Therefore, in our mapping study, we discuss the architectural smells in the same way they are presented in the primary papers. For example, if a primary paper says that they have detected dependency smells, we classify the smells detected by that technique in the "dependency" category. The description of each smell is provided in Appendix A. Mostly, we extracted the description of smells from the catalog papers, however if the definition of a smell is not provided in the catalog paper, we cited one of the primary papers where that description is available.

Since validation holds the key to see the applicability of a detection technique, we also examined if and how the techniques were validated. We noticed that, in many primary papers, "validation" and "evaluation" are used in the same context, therefore, we also use these terms interchangeably throughout this paper. Jedlitschka et al. [27] discussed several essential ingredients of an experimental setup, such as evaluation variables, subjects, data, etc. We used these ingredients as guidance to analyze the evaluation data (project type, project domain, and project language) and evaluation measure. The factors are described as follows:

- **Validation type** – We looked whether the technique was validated and, if so, the validation type. We found all studies were empirical in nature. Ideally, we would have categorized the validations using the five types of empirical studies reported by Easterbrook et al. (controlled experiments, case studies, survey, ethnographies, and action research) [19]. However, the primary papers did not report their validation type at this level of detail. Instead, the primary papers mentioned only case studies and empirical studies as the evaluation methods for the detection approaches. While a case study is a type of an empirical study [19], we report these two validation types separately since these are the two terms used by the primary papers. Usually, a case study allows an in-depth examination of a particular case (single project/system/product), whereas an empirical study gives a more generalized understanding by considering multiple software systems at the same time in the evaluation process [19]. Still, the borderline between these types of evaluation is generally blurred [40], and hence we rely on the terminologies used in the primary papers to classify them.

  Types – Empirical study and case study.

- **Project type** – The validations were performed with different types of projects.
  Types – Open source, commercial, and student project.

- **Project domain** – The projects also belonged to different software domains. Note that the "domain" here is software specific, such as application software (e.g., web application) or system software (e.g., driver).
  Types – Software product line, web application, integration system, service-based systems, android system, distributed system, middleware, parser, and driver.

- **Project language** – The investigated software projects were developed in different programming languages.
  Types – Java, C#, C++, C, AspectJ, PHP, and Python.

- **Evaluation measure** – The articles measured variables to analyze the effectiveness of the detection techniques.
  Types – Precision, recall, correlation, causality, regression, execution (detection or computation) time, robustness, semantic similarity, ranking measure, decoupling level, propagation cost, maintenance cost, response time, throughput, utilization, debt history, and architects' and developers' feedback.

Tool-related factors – For detection tools (RQ3), we extracted the associated architectural smells, the associated technique type, the validation type, tool availability, and language support. These factors

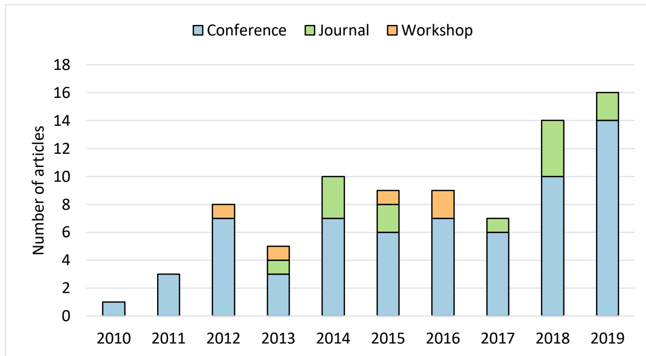

Fig. 3. Year-wise distribution of the articles

were also used in a comparative study to analyze bad smells detection tools [21]. For architectural smells, technique type, and validation type, we used the same framework as described above for detection techniques. The two additional tool-specific factors are described as follows:

- **Tool availability** – We searched in which form the architectural smells detection tools are available. If no information regarding tool availability was provided in the paper, we looked for the official webpage of the tool to obtain this information.
  Types – Open source, commercial, and not available.
- **Language support** – It was also important to note what programming languages are supported by the tool. The rationale was to see the coverage of the tool in terms of the development languages. Once again, if this information was not available in the paper, we located it from the official webpage of the tool.
  Types – Java, C++, C, C#, Python, .Net, PHP, JavaScript, TypeScript, Go, Swift, COBOL, Apex, Kotlin, Ruby, Scala, HTML, CSS, ABAP, Flex, Objective-C, SQL, VB, XML, Ada, Fortran, JOVIAL, Assembly, F#, JSP, R, Erlang, Unix Scripts, and Pascal.

5 Architectural Smells Detection

This section answers our research questions by examining the demographics of the published literature on architectural smells detection (RQ1), presenting and discussing the detection techniques (RQ2) and tools (RQ3), and reflecting on the gaps and limitations of the detection techniques and tools (RQ4). The main findings of our RQs are summarized in Figure 12.

5.1 Demographics (RQ1)

RQ1 – What are the demographics of the published articles?

When – Figure 3 shows the distribution of publications each year (until 2019). There could be more articles published in 2019 that were not indexed because we executed our search in October 2019. Figure 3 shows an overall trend of an increasing number of publications over the years except in 2013 and 2017. A subtle spike was observed in 2014, which resulted from one research group releasing multiple papers in that year.

Where – In Table 2, we list the venues that published the articles related to architectural smells detection. It can be seen that the articles are scattered across many venues; however, some venues tend to publish more on architectural smells detection. For instance, the dedicated conferences on software architecture and technical debt recorded the most publications. The conference with the most number of articles (6) is the *European Conference on Software Architecture*. In terms of journals, *Information and Software Technology* indexed the most articles (3).

Origin – We also looked at the affiliations of the researchers who are active in studying architectural smells detection. We looked at the affiliations of the authors and whether they are from academia or industry. We saw that all of the active researchers have academic affiliations; however, some of them have collaborated with software companies. For instance, Reimanis et al. [P58] applied a detection technique in a company's software to analyze its architectural quality. We did not find any article that was solely produced by industry.

5.2 Detection Techniques (RQ2)

RQ2 – What detection techniques for architectural smells are proposed in literature?

Several architectural smells detection techniques have been proposed in the literature. The goal of RQ2 is to analyze the detection techniques, therefore, we present and discuss the techniques in terms of the analysis framework (formulated in Section 4).

To address RQ2, we present the architectural smells detection techniques in terms of nine categories, identified from the "technique type" factor of our analysis framework during the data extraction and classification process. The nine categories are: rules-based, graph-based, design structure matrix, model-driven, code smells analysis, reverse engineering and history-based, search-based, visualization, and others. Analysing the detection techniques based on their technique type allows identification of the key gaps, limitations, and possible future research directions. For instance, knowing which technique types have limited evaluation would open areas for future work.

In this section, we describe all the detection techniques in terms of these nine technique types. Inside each technique type, we also discuss the other technique-related factors (architecture style, smells detected, quality characteristic, architectural smells, and validation-related factors) to collectively analyze all the related techniques. The articles that belong to the nine categories are listed in Figure 4. Note that, in Figure 4, some primary references appear in two categories, for instance, [P49] appears in both rules-based and search-based categories. In such hybrid approaches, we found that the main techniques were using rules (metrics and thresholds) only to supplement them. Therefore, we describe the hybrid approaches in their main technique section. For instance, in [P49], rules were only supplementing the search-based technique, therefore, it is described in the search-based section. In all hybrid approaches, we found one dominating technique type and one supplementing (i.e., rules with metrics and thresholds). In one case, we also noticed that a technique visualized a dependency graph to identify smells, but since the technique allowed the interactive exploration of smells, we consider it in the "visualization" category rather than the "graph-based" category [P4]. Detailed supplementary material[1] is available, which shows the mapping of each paper to the different categories. The supplemental material also shows how each paper in our dataset was categorized according to all factors in our analysis framework.

5.2.1 Rules-based

The rules-based approach is the most commonly applied strategy to detect smells in software architecture. 13 of the 85 (15.29%) papers proposed a rule-based approach. Rules-based approaches take advantage of: *1)* metrics and their thresholds (rules), and *2)* pre-defined frameworks, heuristics, or guidelines to detect structural problems in the software artifacts. In this section, we describe rules-based approaches that focused on detecting architectural smells.

Description of Existing Rules-based Techniques – Many rules-based techniques applied pre-defined frameworks and patterns to detect architectural smells [P24, P37, P42, P50, P53–P55, P74]. These techniques used architectural guidelines and compliance checking to create frameworks for specifying and detecting architectural smells. In addition, these techniques studied a variety of software architectures, for instance, one technique detected smells in MVC [P74] and some in service architecture [P42, P50, P55].

We also found a couple of techniques that used product metrics to detect architectural smells [P33, P64]. Both of these techniques analyzed the possibility of whether modularity metrics could be used as indicators of architectural technical debt. One of these two techniques measured modularity metrics to identify modularity violations at the package-level [P64]. The other technique detected dependency

[1] http://doi.org/10.5281/zenodo.4013146

Table 2. Publication venues

Venue	Number of articles
European Conference on Software Architecture (ECSA)	6
International Conference on Technical Debt (ICTD)	5
International Conference on Software Engineering (ICSE)	5
International Conference on Service-Oriented Computing (ICSOC)	5
EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)	4
International Workshop on Managing Technical Debt (MTD)	4
Information and Software Technology (IST)	3
International Conference on Software Architecture (ICSA)	3
Working IEEE/IFIP Conference on Software Architecture (WICSA)	3
ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)	2
Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)	2
IEEE Transactions on Software Engineering (TSE)	2
International Conference on Automated Software Engineering	2
Science of Computer Programming (SCP)	2
International Conference on Web Services (ICWS)	2
Journal of Systems and Software (JSS)	1
IEEE Transactions on Services Computing (TSC)	1
IEEE Working Conference on Reverse Engineering (WCRE)	1
IEEE Working Conference on Software Visualization (VISSOFT)	1
International Conference on Software Architecture Workshops (2017 ICSAW—Tool papers)	1
International Conference on Software Architecture Companion (2019 ICSA-C—QUDOS/CSE)	1
International Workshop on Bringing Architectural Design Thinking into Developers Daily Activities (BRIDGE)	1
International Conference on Engineering of Complex Computer Systems (ICECCS)	1
Software and Systems Modeling (SoSyM)	1
International Working Conference on Source Code Analysis and Manipulation (SCAM)	1
Australasian Computer Science Conference (ACSC)	1
International Conference on Software Maintenance and Evolution (ICSME)	1
ACM Symposium on Applied Computing (SAC)	1
IEICE Transaction on Information and Systems (TIS)	1
European Conference on Software Maintenance and Reengineering (CSMR)	1
Journal of Software Engineering Research and Development (JSERD)	1
Brazilian Symposium on Software Engineering (SBES)	1
Conference on Genetic and Evolutionary Computation (GECCO)	1
International Journal of Cooperative Information Systems (IJCIS)	1
International Symposium on Empirical Software Engineering and Measurement	1
International Conference on Software Engineering and Formal Methods (SEFM)	1
International Conference on Cloud Computing Technology and Science (CloudCom)	1
International Conference on Software Analysis, Evolution and Re-engineering (SANER)	1
Hawaii International Conference on System Sciences (HICSS)	1
Computer Software and Applications Conference (COMPSAC)	1
International Journal of Computer Theory and Engineering (IJCTE)	1
ACM SIGSOFT Software Engineering Notes (SEN)	1
International Conference on Software Process Improvement (CIMPS)	1
International Conference on Mobile Software Engineering and Systems (MOBILESoft)	1
Institute for System Programming (ISP)	1
Asia-Pacific Symposium on Internetware (INTERNETWARE)	1
Belgium-Netherlands Software Evolution Workshop (BENEVOL)	1

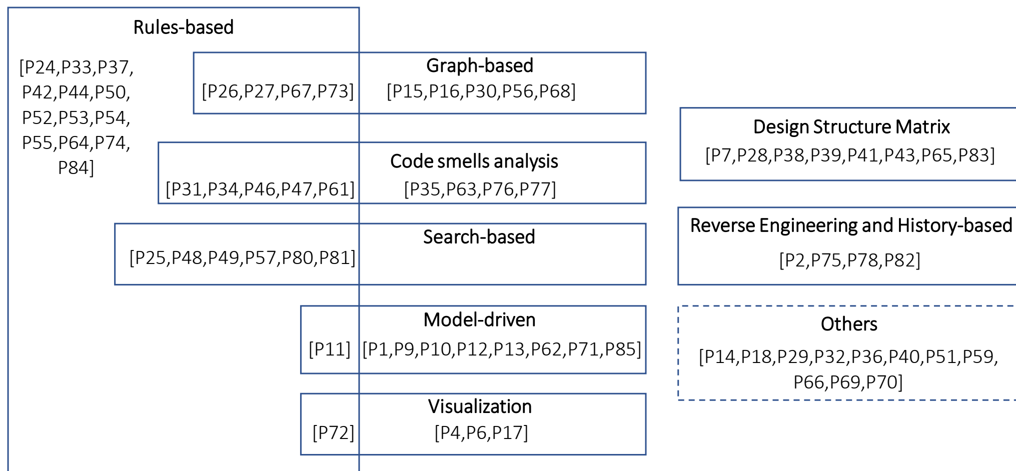

Fig. 4. Categorization of the architectural smells detection techniques

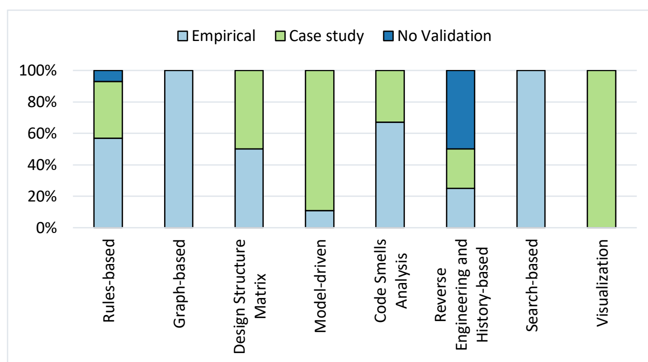

Fig. 5. Validation type used in the detection techniques

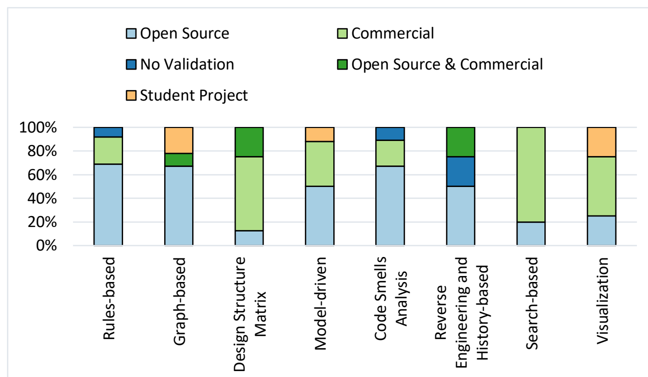

Fig. 6. Project type used in the validation of detection techniques

smells that were involved in increasing the complexity of the Java packages [P33].

The findings from the rules-based techniques in terms of the analysis factors are described as follows:

- **Architecture style** – We observed that approximately 60% of the rules-based techniques focused on service-oriented architecture [P42, P44, P50, P52–P55, P84]. We found only one technique that focused on MVC archiecture [P74].
- **Quality characteristic** – The rules-based techniques focused on a variety of quality aspects, such as modularity [P37, P64], maintainability [P33, P42, P44, P50, P53–P55, P74, P84], complexity [P33, P52], and evolvability [P42, P44, P50, P52–P55, P84].
- **Smells detected** – The architectural smells detected by rules-based approaches are presented in Table 3. Because of the attention on the service-oriented architecture, the detected smells were service-based, prominently, Multi-service, Chatty Service, Nano-service, Data Service, and Ambiguous Service. The technique that studied MVC architecture investigated MVC-related smells [P74]. A few rules-based approaches also detected dependency and modularity smells in the architecture [P33, P37, P64]. These techniques focused on package-level smells, however, the specific smells that can be detected were not described in the papers. Note that the package-level smells refer to smells in Java packages, which is semantically similar to namespaces in other languages (e.g., C#).
- **Validation** – All but one of the techniques were validated. Approximately 60% of the techniques with validation were evaluated empirically, while the rest were validated through case studies (see Figure 5). Open source systems were mainly used in the validations of the rules-based techniques (see Figure 6), particularly, in the cases, where service-oriented smells were detected. The validation studies mainly measured precision and recall of the proposed approaches as indicators of the effectiveness of the techniques. For the techniques that used product metrics, one measured correlation [P33] and the other technique measured both correlation and regression [P64]. In many techniques, in addition to measuring recall or precision, performance of their detection was also measured in terms of execution/computation time [P24, P44, P53, P55, P74]. In another technique, robustness, accuracy (recall and precision), execution time, and detection time were computed as performance indicators [P52].

Rules common in other technique types – We also noticed that rules-based approaches (specifically metrics and thresholds) were com-

monly used in combination with other technique types [P25–P27, P48, P49, P57, P72, P80, P81]. For instance, graph-based techniques (e.g. [P26, P27]), search-based techniques (e.g. [P25, P48, P49, P57, P80, P81]), a model-driven technique ([P11]), and a visualization technique ([P72]) incorporate some rules in the detection techniques. For all of these, rules are used only to supplement the main technique type, so these are described and analyzed later in the relevant technique type.

Table 3. Architectural smells detected by rules-based approaches

ID	Architectural smells
[P24]	Cyclic Dependencies, Factory Pattern Violations, Layering Violations, other user-defined pattern violation
[P33]	Dependency smells* (package-level)
[P37]	Dependency smells* with respect to complexity and modularity
[P42,P55]	Multi-service, Tiny Service, Sand Pile, Chatty Service, Knot Service, Nobody Home, Duplicated Service, Bottleneck Service, Service Chain, Data Service
[P44]	Multi-service, Tiny Service, Chatty Service, Knot Service, Bottleneck Service, Service Chain
[P50]	Multi-service, Tiny Service, Duplicated Service, Bottleneck Service
[P52]	Breaking Self-descriptiveness, Forgetting Hypermedia, Ignoring Caching, Ignoring MIME Types, Ignoring Status Code, Misusing Cookies, Tunneling through GET, Tunneling through POST
[P53]	Ambiguous Name, Bloated Service, Multi-service, Tiny Service, Nobody Home, Crudy Interface, Crudy URI, Sand Pile, Chatty Web Service, Forgetting Hypermedia, Ignoring MIME Types
[P54]	Ambiguous Name, Chatty Web Service, Crudy Interface, Data Web Service, Duplicated Web Service, Fine-grained Web Service, God Object Web Service, Low Cohesive Operations (in the same portType), Maybe it's not RPC, Redundant PortTypes
[P64]	Modularity violations* (package-level)
[P74]	Model, view, and controller using each other's data/functionalities
[P84]	God Object

*specific smells not mentioned in the paper

5.2.2 Graph-based

Graph-based approaches depict entities or components of a system as nodes, and the relationships are represented through edges. Graph-based techniques are extensively used to detect architectural smells because it is intuitive to represent problematic relationships between the entities of the software architecture in graphical form. In this section, we present graph-based approaches (9 of 85—10.58%) that were employed to detect architectural smells.

Description of Existing Graph-based Techniques – Graph-based techniques have been used to predict architectural smells [P15,P56,P68]. Two of these techniques used semantic information to predict the architectural smells [P15, P56]. For instance, Diaz-Pace et al. [P15] explored the use of social network analysis to extract architecture information useful to predict the undesired dependencies in the future versions of Java projects. The predictions were mainly based on identifying problematic links (that represent dependency smells) between the components (represented as nodes in the graph) of the architecture. For instance, a node (component) involved in many links (many dependencies) with other nodes might create modularization issues in the architecture. Such links between nodes of a graph represent undesired dependencies that should be removed to have a better maintainable architecture. Another technique analyzed relationships between architectural smells using a dependency graph to see if a smell can indicate the presence of other smells [P30]. We also observed a graph-based approach that not only detects architectural smells, but also suggests appropriate refactorings for the detected smells [P16].

Some of the graph-based techniques used a combination of graph-based and rule-based techniques to identify architectural smells [P26, P27, P67, P73]. Some apply rules or constraints to the graphical representation of the software architecture [P26, P27, P73]. Others combine product metrics with dependency graphs to detect architectural smells [P67].

The findings from the graph-based techniques in terms of the analysis factors are described as follows:

- **Architecture style** – Mostly, the architecture style is generic for the graph-based techniques, however, a couple of techniques focused on MVC architecture [P26, P27], and one focused on the software architecture in C-language environment [P67].

- **Quality characteristic** – The focus of graph-based techniques (approximately 67%) was mainly on maintainability quality [P26, P27, P30, P56, P67, P68]. One technique focused on modularization aspects—a sub-characteristic of maintainability—of the software architecture [P16] and another on security aspects [P73].

- **Smells detected** – A variety of architectural smells were detected through graph-based approaches, for instance, dependency-related smells (e.g., Cyclic, Unstable, Hub-like, etc.) [P15, P26, P27, P30, P67, P68], responsibility violations (e.g., Scattered Functionality, Feature Concentration, etc.) [P26, P27, P56], and security flaws (e.g. Information Disclosure and Information Tampering) [P73]. One technique detected the architectural smells at three abstraction levels: module (package), file, and function using graphs [P67]. Architectural smells detected by graph-based approaches are listed in Table 4.

- **Validation** – The techniques were evaluated empirically mainly with open source projects (see Figure 5 and Figure 6). All evaluations used projects developed in Java, except in one case [P67], where the evaluation was performed using projects written in C. The effectiveness of the proposed graph-based approaches was measured using recall and precision, except in four papers, where two computed semantic similarity [P15,P56]; one measured different constraints (e.g., provenance, transitivity, reachability, etc.) [P73]; and one computed multiple correlations [P67].

Table 4. Architectural smells detected by graph-based approaches

ID	Architectural smells
[P15]	Cyclic Dependencies, Hub-like Dependencies
[P16]	Abstraction without Decoupling, Subtype Knowledge, Degenerated Inheritance, Cycles between Namespaces
[P26,P27]	Responsibility violations (Separation of Concerns), Dependency violations (Unauthorized dependency)
[P30]	Concern Overload, Cyclic Dependency, Link Overload, Unused Interface, Sloppy Delegation, Co-change Coupling
[P56]	Scattered Functionality, Feature Concentration
[P67]	Large File, Hub-like Dependencies, Message Chain, Shotgun Surgery, Cyclic Dependencies, Inappropriate Intimacy, Feature Envy, Long Parameter List
[P68]	Cyclic Dependencies, Unstable Dependencies, Hub-like Dependencies
[P73]	Security flaws (Information Disclosure, Information Tampering)

5.2.3 Design Structure Matrix (DSM)

A design structure matrix is a mechanism commonly used to design, develop, understand, and manage complex system [20]. It is a two-dimensional matrix that represents the structural relationships in the software. DSMs are widely employed to detect architectural smells because the matrix can represent complex architectural components and their relationships.

Description of Existing DSM Techniques – Many approaches (8 of 85—9.41%) have been developed that use a DSM to represent software architecture data to locate clusters and patterns that represent

architectural smells [P7, P28, P38, P39, P41, P43, P65, P83]. For instance, one technique created a coupling probability matrix based on the design structure matrix idea to identify four types of architectural flaws [P83]. Undesired coupling between components of the architecture was linked to dependency-related smells. Similarly, in two DSM techniques [P28, P65], authors extracted data from software architecture and depicted it using DSM to identify file clusters that participated in architectural smells. Similarly, some approaches defined patterns and antipatterns based on design rule theory using architectural knowledge [P7, P38, P39]. In these techniques, the complex software architecture was split into smaller components, patterns, etc. and depicted using the design rule space method.

The findings from the DSM techniques in terms of the analysis factors are described as follows:

- **Architecture style** – None of the DSM techniques studied a specific architecture style, except one [P83] in which component architecture was studied. The rest were all applicable to generic architecture styles.
- **Quality characteristic** – We noticed that only the maintainability of the software architecture was assessed through design structure matrix.
- **Smells detected** – Architectural smells, such as Unstable Interface, Package Cycles, Unhealthy Inheritance, Clique, and Module Dependency, were mostly identified by structuring the architectural components and entities into a design structure matrix. All the DSM techniques supported the detection of Unstable Interface. In one technique, component-level smells (Hub, Anchor Submission, and Anchor Dominant) were identified [P83]. The architectural smells detected by DSM approaches are listed in Table 5.
- **Validation** – Both empirical and case studies were conducted with mainly commercial projects (see Figure 5 and Figure 6). The projects used to evaluate the techniques were implemented in different development languages. Nearly 38% of the approaches used Java [P28, P38, P39] or C# [P7, P41, P65], while 25% ([P7, P41]) selected C and C++ projects for validation. In several techniques, the maintenance cost of the architecture was evaluated by considering change frequency, change churn, bug frequency, and bug churn [P7, P38, P39, P41, P65, P83]. The effectiveness of some techniques was also measured using recall [P7, P28, P38, P65] and one through precision [P28]. In one study, the evaluation measured the decoupling level and propagation cost of a system [P43].

Table 5. Architectural smells detected by DSM approaches

ID	Architectural smells
[P7]	Unstable Interface, Modularity Violation, Improper Inheritance, Cross-module Cycle, Cross-package Cycle
[P28]	Unstable Interface, Implicit Cross-module Dependency, Unhealthy Inheritance Hierarchy
[P38]	Unstable Interface, Implicit Cross-module Dependency, Unhealthy Inheritance Hierarchy, Cross-module Cycle, Cross-package Cycle
[P39]	Unstable Interface, Modularity Violations, Unhealthy Inheritance Hierarchy, Crossing, Clique, Package Cycle
[P41]	Unstable Interface, Modularity Violation, Unhealthy Inheritance, Cyclic Dependency, Package Cycle, Crossing
[P43]	Clique, Package Cycle, Improper Inheritance, Modularity Violation, Crossing, Unstable Interface
[P65]	Unstable Interface, Implicit Cross-module Dependency, Unhealthy Inheritance Hierarchy, Clique, Package Cycle
[P83]	Hub, Anchor Submission, Anchor Dominant, Modularity Violation

5.2.4 Model-driven

In model-driven methods, the structure and behavior of the systems are represented using abstractions and modeling [33]. Model-driven approaches are common to detect architectural smells because, through modeling, abstractions can be created that generate structures to analyze software architecture. 9 of the 85 (10.58%) papers implemented model-driven approaches.

Description of Existing Model-driven Techniques – Many model-driven approaches took advantage of model transformation mechanisms to detect architectural smells [P1, P9–P13, P71]. For instance, in two approaches the model transformation was achieved using intermediate XML representations [P10, P11]. The detection of smells was accomplished by assigning logical predicates in the XML representations. Similarly, a couple of techniques used a stochastic process to model performance-aware components in software architecture [P9, P13]. A model-driven technique detected anomalies related to communication within and among the components of the architecture [P62]. In another approach, a meta-model for a Java EE application was created. In the meta-model, they used Query/View/Transformation language to state the process of antipatterns detection [P85].

The findings from the model-driven techniques in terms of the analysis factors are described as follows:

- **Architecture style** – Mostly, model-driven approaches did not focus on specific architectural styles, except for a few techniques where the techniques focused on micro-services [P1], Java EE [P85], and component [P62] architectures.
- **Quality characteristic** – Approximately 67% of the approaches measured the impact of architectural smells on the performance of software architecture [P1, P9–P11, P13, P71]. There were also two methods that looked into maintainability aspects of software architecture [P12, P85].
- **Smells detected** – Most of the techniques measured and detected performance smells [P1, P9–P11, P13, P71]. The prominently detected smells were Blob, Unbalanced Processing, One-lane Bridge, Traffic Jam, and Ramp. The architectural smells detected by model-driven techniques are presented in Table 6.
- **Validation** – From the validation perspective, case studies were performed with commercial and open source systems (see Figure 5 and Figure 6). The project domains in half of the techniques were not specified [P9, P13, P71], whereas the other half used web-based systems [P1, P10, P11]. In the case where a technique studied service architecture, the investigated projects were developed in REST [P1]. For Java EE architecture, Java projects were used [P85]. Where the performance of the architecture was evaluated, response time, throughput, and utilization were measured. In some instances, recall was computed as an evaluation measure for the proposed techniques [P12, P62, P85].

5.2.5 Code Smells Analysis

Some of the proposed techniques (9 of 85—10.58%) analyzed code smells to identify architectural smells. The main idea of these methods is to see how source code smells can be used to detect smells in the software architecture.

Description of Existing Code Smells Analysis Techniques – Several detection techniques used the knowledge of code smells to locate architectural smells [P31, P34, P35, P46, P47, P61, P63, P76, P77]. These techniques mainly looked at the correlation between code smells and architectural smells. In other words, the proposed techniques analyzed whether code smells in a software system could indicate smells in the corresponding architecture of that system. For example, a class shows the lack of modularization if it has a greater number of lines of code—represents a large class smell. This large class could indicate the modularization issues (e.g., Blob or God Object, etc.) in the related architecture of the system. Note that identifying the correlation between code and architectural smells may not be exactly the detection

Table 6. Architectural smells detected by model-driven approaches

ID	Architectural smells
[P1]	Pipe and Filter, One-lane Bridge
[P9]	Blob, Unbalanced Processing, One-lane Bridge, Excessive Dynamic Allocation, Traffic Jam, Ramp
[P10]	Unbalanced Processing (Concurrent Processing Systems), Circuitous Treasure Hunt, Ramp
[P11]	Blob, Unbalanced Processing, Circuitous Treasure Hunt, Empty Semi-trucks, Tower of Babel, One-lane Bridge, Traffic Jam, Excessive Dynamic Allocation, Ramp, More is Less
[P12]	Unhealthy Inheritance, Cross-module Cycles, Package Cycles
[P13]	Blob, Extensive Processing, One-lane Bridge, Excessive Dynamic Allocation, Traffic Jam, Ramp, Pipe and Filter
[P62]	Anomalies within and among components (communication-related)
[P71]	Blob, Unbalanced Processing, Circuitous Treasure Hunt, Empty Semi-trucks, One-lane Bridge, Traffic Jam
[P85]	Fine-grained Web Service, Multi-service, Tiny Service, Ignoring Reality, Too Much Code, Embedded Navigation Information, Accessing Entity Directly, Fine-grained Remote Calls, Transparent Facade, Session A-plenty, Stifle, Database Connection Hog, Performance Afterthoughts

of architectural smells, but the correlation, if identified, can be used to predict the presence of architectural smells based on the analysis of code smells. As a specific instance, a technique used static code analysis and cloud information to predict the appearance of performance smells if a system is migrated to the cloud architecture [P61]. Approximately half of the code smells analysis techniques also employed code smells metrics which were relevant to the identification of architectural smells [P31, P34, P46, P47, P61]. For instance, in [P61], the authors used five code-level metrics (related to size and complexity) to detect "Blob" smell, which corresponds to the lack of modularization in the architecture. One paper also investigated the self-admitted technical debt left by the developers in the source code to identify some architectural divergences related to dependency smells [P63]. This paper did not directly use the code smells to detect architectural smells but used the technical debt information stated (by the developers) in the comments of the source code.

The findings from the code smells analysis techniques in terms of the analysis factors are described as follows:

- **Architecture style** – Code smells analysis techniques investigated different architectural styles. Approximately, 78% ([P34, P35, P46, P47, P63, P76, P77]) and 67% ([P34, P35, P46, P47, P76, P77]) of the studies examined layered and MVC architectures, respectively. We also found one technique for each of the following architectures: cloud [P61], aspect-oriented (aspectual) [P35], and client-server [P47].

- **Quality characteristic** – The techniques measured different quality characteristics (e.g., maintainability [P34, P35, P46, P47, P76, P77], consistency [P31], and performance [P61]).

- **Smells detected** – Since the techniques mainly studied MVC and layered architectures, the architectural smells were mostly specific to these two architectures. In cloud architecture, performance smells (Empty Semi-trucks, Circuitous Treasure Hunt, and Blob) were detected [P61]. In one technique, inconsistent classes and components in the architecture were explored [P31]. The architectural smells detected through code smells analysis are listed in Table 7.

- **Validation** – It can be seen from Figure 5 that the techniques were validated using empirical studies and case studies. Most (67%—see Figure 6) were conducted with open source Java projects [P31, P34, P46, P63, P76, P77]. Only one study used projects developed in C++, C#, and AspectJ languages for the validation [P34]. Most studies (67%) were validated using projects from the software product line, web, and middleware domains [P34, P35, P46, P47, P76, P77]. The effectiveness of the approaches was mostly measured using correlation, precision, and recall.

Table 7. Architectural smells detected by code smells analysis approaches

ID	Architectural smells
[P31]	Inconsistent Classes/Components
[P34, P35]	Ambiguous Interface, Extraneous Connector, Connector Envy, Scattered Functionality, Concern Overload
[P46]	Ambiguous Interface, Concern Overload, Connector Envy, Cyclic Dependency, Scattered Functionality, Unused Interface
[P47]	Ambiguous Interface, Connector Envy, Concern Overload, Cyclic Dependency, Extraneous Connector, Scattered Functionality, Unused Interface, Architectural Violation
[P61]	Empty Semi-trucks, Circuitous Treasure Hunt, Blob
[P63]	Dependency smells (specific smells not mentioned)
[P76, P77]	Ambiguous Interface, Concern Overload, Connector Envy, Cyclic Dependency, Scattered Functionality, Unused Interface, Package-level Feature Envy, Package-level Dispersed Coupling

5.2.6 Reverse Engineering and History-based

Reverse engineering is a process to deduce design and architectural features from the developed software systems to learn about the production procedures involved in their initial development [15]. History data analysis is also a way to understand the patterns and changes in the software systems. In this section, we describe approaches (4 of 85—4.7%) that explored historical data and reverse-engineered the artifacts to detect architectural smells.

Description of Existing Reverse Engineering and History-based Techniques – In the literature, we found a few techniques based on historical data or reverse engineering of the software artifacts [P2, P75, P78, P82]. Some techniques used reverse engineering mechanisms and historical data analysis to cluster the data elements to identify architectural smells [P75, P78, P82]. These techniques used architectural guidelines and compliance checking to identify clusters with problematic dependencies and connections. These problematic clusters were identified as architectural smells. One techniques used only historical data to predict architectural smells [P2]. In this approach, the authors used a link-prediction strategy where dependency information from the previous versions of the software was used to predict architectural issues in the proceeding versions.

The findings from the reverse engineering and history-based techniques in terms of the analysis factors are described as follows:

- **Architecture style** – These techniques focused on generalized architectural style, with an exception, where Verdecchia et al. [P75] focused on the Android architecture.

- **Quality characteristic** – All the techniques focused specifically on maintainability [P2, P75, P78, P82].

- **Smells detected** – Mostly, dependency-related architectural smells (e.g., Cyclic Dependency, Unstable Dependency, Hub-like Dependency, etc.) were identified by either reverse-engineering or extracting historical data from the systems [P2, P82]. However, in one technique, communication-related smells were identified [P78]. The detected smells by each approach are presented in Table 8.

- **Validation** – Half of the approaches were not validated. The two approaches where validations were performed; one [P82] performed an empirical study (using open source and commercial systems) and the other [P78] conducted a case study (with open

source projects) (see Figure 5). Both validations were performed using open source systems (see Figure 6). Precision [P78] and debt history [P82] were used as evaluation measures.

Table 8. Architectural smells detected by reverse engineering and history-based approaches

ID	Architectural smells
[P2]	Unstable Dependency, Cyclic Dependency, Hub-like Dependency, Implicit Cross-package Dependency
[P75]	Android Architecture Violations
[P78]	Interface Violation, Undercover Transfer Object, Non-transfer Objects Communication, Unauthorized Call
[P82]	Dependency smells (specific smells not mentioned)

5.2.7 Search-based

Search-based methods in software engineering formulate problems as computational search problems that can be solved with a metaheuristic approach [16]. A few approaches (6 of 85—7.05%) employed search-based techniques to detect architectural smells.

Description of Existing Search-based Techniques – We found that all the search-based approaches were developed in combination with rules [P25, P48, P49, P57, P80, P81]. The approaches mainly employed genetic programming, for instance, in [P49], the approach implemented an algorithm to optimize the detection solution by using the crossover and mutation operations of the genetic programming. Their solution was composed of detection rules based on the real examples of smells in web services. The rules were formulated using product metrics and thresholds, and each rule represented an instance of a smell in the architecture.

The findings from the search-based techniques in terms of the analysis factors are described as follows:

- **Architecture style and quality characteristic** – We observed that 100% of the search-based techniques focused on service-oriented architecture and maintainability quality characteristic.
- **Smells detected** – The techniques investigated a variety of service-oriented smells, such as Multi-service, Chatty Service, Data Service, etc. The architectural smells detected by each search-based technique are listed in Table 9.
- **Validation** – All of the search-based techniques were empirically validated using web-based systems (see Figure 5). In addition, in Figure 6, it can be seen that four out of the five (80%) techniques were evaluated with commercial projects. One [P49] used open source projects in the evaluation. The effectiveness of all of the search-based techniques was measured using precision and recall.

Table 9. Architectural smells detected by search-based approaches

ID	Architectural smells
[P25, P49, P80, P81]	Chatty Web Service, Crudy Interface, Data Web Service, Fine-grained Web Service, God Object Web Service, Maybe it's not RPC, Redundant PortTypes, Ambiguous Web Service
[P48, P57]	Mutli-service, Nano-service, Chatty Service, Data Service, Ambiguous Service

5.2.8 Visualization

Only a few methods (4 of 85—4.7%) employed visualization strategies to detect architectural smells. Visualization techniques can aid in the understanding of large software systems with multivariate and multidimensional data [36, 37].

Description of Existing Visualization Techniques – Visualization techniques were not the commonly used techniques to detect architectural smells. Only four techniques used visualization [P4, P6, P17, P72]. In one of the studies [P72], a visualization technique was used in combination with detection rules (product metrics and thresholds) to identify architectural smells automatically. Others propose visualizations that enable developers or architects to manually identify architectural smells as they can use the interactive functionalities to explore the architecture [P4, P6, P17].

The findings from the visualization techniques in terms of the analysis factors are described as follows:

- **Architecture style and quality characteristic** – All the visualization techniques focused on generalized architectural style. In terms of quality characteristic, half ([P17, P72]) of the approaches focused on performance, while one considered maintainability [P4].
- **Smells detected** – Half of the techniques detected dependency-related architectural smells [P4, P6], and the other half identified performance smells [P17, P72]. The architectural smells detected by the visualization techniques are presented in Table 10.
- **Validation** – From the validation perspective, the visualization techniques were evaluated only through case studies (see Figure 5) and mostly (50%) with commercial projects (see Figure 6). Evaluation measures related to performance (utilization, throughput, and response time) were computed in [P72]. In the other performance-oriented approach, efficiency and maintenance cost were measured [P17]. In the remaining two techniques [P4, P6], the authors measured recall to express the effectiveness.

Table 10. Architectural smells detected by visualization approaches

ID	Architectural smells
[P4]	Cyclic Dependencies, Subtype Knowledge
[P6]	Dependency smells (specific smells not mentioned)
[P17]	Misplaced Component
[P72]	Blob, Unbalanced Processing, Circuitous Treasure Hunt, Empty Semi-trucks, One-lane Bridge, Traffic Jam

5.2.9 Others

In this section, we briefly describe the techniques (11 of 85—12.94%) that do not fit in the above categories. In other words, a technique whose foundation is not based on any of the categories described above is placed into this "Others" category.

Fontana et al. [P18] proposed an approach to identify architectural technical debt using architecture decisions and change scenarios. Tamburri et al. [P66] found the correlation of community smells with architectural smells in sub-optimal modularization structures. The main idea was to use community smells as indicators of smells in the architecture. Martini et al. [P36] used questionnaire and interviews to collect the detection procedure adopted by the software practitioners. They found that practitioners used the combination of tool and their knowledge of architecture to identify architectural technical debt. De Toledo et al. [P14] performed a qualitative analysis of documents and interviews to identify technical debt in the communication layer of a micro-service system. For instance, they identified too many point-to-point connections between micro-services (i.e., Chatty Micro-service), creating a high volume of connections passing through the communication layer.

Sanchez et al. [P59] presented an approach for specifying and identifying architectural smells as constraints using the Archery architectural description language. Mo et al. [P40] transformed architectural models into an augmented constraint network to identify dependency relations related to architectural decay. Trubiani et al. [P70] executed load testing using a profiler tool to obtain performance hotspots, which are related to the specifications of the performance antipatterns. Palma et al. [P51]

quantified the impact of service antipatterns on the maintenance effort in service-based systems. They performed multiple statistical tests to identify relationships between change-proneness/code churn and antipatterns. Tripathi et al. [P69] identified and presented wrong practices in the antipattern template of service-oriented architecture. Le et al. [P29] proposed a framework to detect architectural smells based on their symptoms with the help of architecture recovery and analysis.

Since the approaches described in the "Others" category are distinctive in nature, we are not describing their commonalities and differences as we did in the technique sections above. However, we found some commonalities between these techniques, for instance, some techniques detected dependency smells [P18, P29, P32, P36]; a few MVC smells [P29, P40, P59]; and some identified smells in service-oriented architecture [P14, P51, P69].

5.3 Detection Tools (RQ3)

RQ3 – What detection tools for architectural smells are proposed and evaluated in literature?

This section addresses RQ3 by describing the architectural smell detection tools reported in the literature (12 of 85—14.11%). The descriptions of tools are presented in a similar manner using a subset of the categories developed for detection techniques. We used only a subset because we adopted only those categories employed by the tools. Similar to detection techniques, the detailed supplementary material[1] is also available for detection tools.

Graph-based – Fontana et al. [P20] introduced an open source tool named Arcan that detects architectural smells in Java projects. The tool uses abstraction knowledge of the project to identify dependencies—including problematic architectural dependencies—between the project's elements. Later, Biaggi et al. [P5] improved the scalability of Arcan by adding the functionality to handle projects compiled in C and C++ languages. Fontana et al. [P19] evaluated the Arcan tool to see if architectural smells detected by Arcan are actually perceived as architectural issues. Fontana et al. [P23] also reported their experience of employing two detection tools (Sonargraph and Structure101) to identify architectural erosion in the open source systems. In another report, Fontana et al. [P22] presented their experience report on three architectural smells detection tools (Sonargraph, SonarQube, and inFusion). Von Zitzewitz [P79] described the architecture, working, and application of the Sonargraph tool. The tool allowed software architects to describe the architectural blueprint using a customized domain specific language (DSL). Once the DSL was defined, the architecture quality was automatically checked and enforced during the development process.

Azadi et al. [P3] illustrated the key differences in the detection techniques exploited by the existing detection tools (AI Reviewer, ARCADE, Arcan, Designite, Hotspot Detector (no longer available as a standalone tool—now integrated into DV8 suite), Massey Architecture Explorer (no longer available), Sonargraph, STAN, and Structure101). They showed which and how the architectural smells were detected by these tools. For instance, in the case of Hub-like Dependency, the supported tools (ARCADE, AI Reviewer, Arcan, and Designite) employed graphs in different ways. AI Reviewer detected Hub-like Dependencies by focusing on the ingoing and outgoing dependencies of concrete classes (non-abstract) in the dependency graph; Arcan focused on abstraction and unbalanced (ingoing and outgoing) dependencies in the graph; Designite relied on fan-in and fan-out metrics to identify Hub-like Dependencies using the dependency graph; and finally, ARCADE detected this smell by comparing the ingoing and outgoing dependencies with the aggregated dependencies in the graph. Similarly, for each supported architectural smell, the authors demonstrated the different detection strategies of the tools. In a similar manner, Fontana et al. [P21] explained how the technical debt indexes are computed in five different detection tools (CAST, inFusion, Sonargraph, SonarQube, and Structure101). They also demonstrated the different detection mechanisms used by these detection tools. We also observed that the most commonly employed detection tools were Arcan, Sonargraph, SonarQube, and Structure101.

Visualization – Sharma [P60] presented Designite—a quality assessment tool—that identifies several architectural smells. Designite also provides information about the root cause of the smells to assist the developers in refactoring. Cai and Kazman [P8] presented a tool suite, named DV8, that provides maintainability assessment using two metrics (decoupling level and propagation cost); detection of six architectural smells (Unstable Interface, Modularity Violations, Unhealthy Inheritance Hierarchy, Crossing, Clique, and Package Cycle); and quantification of maintenance cost. They showed that, by using DV8 suite metrics and visualization, they were able to automatically detect architectural smells and express the maintenance difficulties in the projects. The suite also calculated the maintenance cost of the files affected by the architectural smells.

History-based – Reimanis et al. [P58] measured and predicted the architecture quality of a system by using the structural information of that system. The structural information was in the form of product metrics about historical changes in the structure of the system's architecture. The structural information (metrics) were fed to CLIO (an architectural degradation detection tool) [52] to automatically identify dependencies between the components of a commercial system. The main contribution of CLIO was the automated detection of smells in the commercial system developed in, previously not supported, C++ language.

Rules-based – Nayrolles et al. [P45] presented an open source service-oriented antipatterns detection tool, SODA, to automatically detect antipatterns in service-based systems. They identified several service-related architectural smells.

The findings from the tool papers in terms of the related analysis factors are described as follows:

- **Smells detected** – The detected architectural smells reported in the tool papers are summarized in Table 11. Note that the architectural smells that can be detected by each tool is not mentioned on the official webpages of the tools; therefore, we relied on the smells reported in the tool papers. We noticed that identifying dependency smells was common among the detection tools. For instance, problematic dependencies, such as Cyclic, Unstable, Hub-like, Implicit Cross-module, etc. were common among the majority of the tools. On the other hand, only one tool (SODA) detects service-oriented smells.

- **Technique type** – Since it is easier to depict and analyze dependencies using graphs, the most common technique type employed by the tools was graph-based (almost 67%) [P3,P5,P19–P23,P79]. One tool, SODA [P45], handled the service-oriented architectural smells by implementing rules. Designite [P60] and DV8 [P8] used the combination of visualization and rules to identify architectural smells automatically. Another tool, CLIO [P58], took advantage of historical information to detect architectural smells.

- **Validation** – Only Arcan was empirically validated (see Figure 7). In the rest of the tool papers, the evaluations were performed using case studies (4 out of 12 papers) or no validation was conducted (5 out of 12 papers).

- **Tool availability** – We found that the detection tools are available as a mix of open source and commercial products. For instance, Arcan is fully open source; Structure101 is commercial; Sonargraph and SonarQube are available as open source (limited functionality) and commercial. In Table 12 and Figure 8, we show the distribution of tools available as open source, commercial, both, and not available. Out of 15, three are open source, six are commercial, and three are available both as open source and commercial. Three tools (inFusion, Hotspot Detector, and Massey Architecture Explorer) are no longer available.

- **Language support** – Language coverage is also spread to have maximum language support. Some tools, like DV8, SonarQube, and CAST, manage a long list of development languages, including C, C++, Java, C#, Python, PHP, .Net, and many more (see

Table 11. Mapping of detected architectural smells to tools as reported in the primary papers.

Architectural smell	Arcan	Sonargraph	Structure101	CLIO	Designite	DV8	SonarQube	inFusion	AI Reviewer	ARCADE	Hotspot Detector	Massey Architecture Explorer	STAN	CAST	SODA
Multi-service	–	–	–	–	–	–	–	–	–	–	–	–	–	–	✓
Tiny Service	–	–	–	–	–	–	–	–	–	–	–	–	–	–	✓
Sand Pile	–	–	–	–	–	–	–	–	–	–	–	–	–	–	✓
Chatty Service	–	–	–	–	–	–	–	–	–	–	–	–	–	–	✓
Knot Service	–	–	–	–	–	–	–	–	–	–	–	–	–	–	✓
Nobody Home	–	–	–	–	–	–	–	–	–	–	–	–	–	–	✓
Duplicated Service	–	–	–	–	–	–	–	–	–	–	–	–	–	–	✓
Bottleneck Service	–	–	–	–	–	–	–	–	–	–	–	–	–	–	✓
Service Chain	–	–	–	–	–	–	–	–	–	–	–	–	–	–	✓
Data Service	–	–	–	–	–	–	–	–	–	–	–	–	–	–	✓
Bloated Service	–	–	–	–	–	–	–	–	–	–	–	–	–	–	✓
Unstable Dependency	✓	✓	✓	–	✓	–	–	✓	✓	✓	✓	✓	✓	–	–
Hub-like Dependency	✓	✓	✓	–	–	–	–	–	✓	✓	✓	✓	✓	–	–
Cyclic Dependency	✓	✓	✓	–	✓	–	✓	✓	✓	✓	✓	✓	✓	–	–
Implicit Cross-module Dependency	–	–	–	–	–	–	–	–	✓	✓	✓	✓	✓	–	–
Package Cycles	–	✓	✓	–	–	✓	–	–	–	–	–	–	–	–	–
Biggest Package Cycle Group	–	✓	✓	–	–	–	–	–	–	–	–	–	–	–	–
Ambiguous Interface	–	–	–	–	✓	–	–	–	✓	✓	✓	✓	✓	–	–
Unstable Interface	–	–	–	–	–	✓	–	–	–	–	–	–	–	–	–
Unused Interface	–	–	–	–	–	–	–	–	–	✓	–	–	–	–	–
Unhealthy Inheritance Hierarchy	–	–	–	–	–	✓	–	–	–	–	–	–	–	–	–
Cyclic Hierarchy	–	–	–	–	–	–	–	–	✓	✓	✓	✓	✓	–	–
Multipath Hierarchy	–	–	–	–	–	–	–	–	✓	✓	✓	✓	✓	–	–
Abstraction without Decoupling	–	–	–	–	–	–	–	–	✓	✓	✓	✓	✓	–	–
Unutilized Abstraction	–	–	–	–	–	–	–	–	✓	✓	✓	✓	✓	–	–
Modularity Violations	–	–	–	✓	–	✓	–	–	–	–	–	–	–	–	–
God Component	–	–	–	–	✓	–	–	–	✓	✓	✓	✓	✓	–	–
Feature Concentration	–	–	–	–	✓	–	–	–	–	–	–	–	–	–	–
Scattered Functionality	–	–	–	–	✓	–	–	–	✓	✓	✓	✓	✓	–	–
Dense Structure	–	–	–	–	✓	–	–	–	–	–	–	–	–	–	–
Crossing	–	–	–	–	–	✓	–	–	–	–	–	–	–	–	–
Clique	–	–	–	–	–	✓	–	–	–	–	–	–	–	–	–
SAP Breaker	–	–	–	–	–	–	–	✓	–	–	–	–	–	–	–
Multiple Architecture Violations	✓	–	–	–	–	–	–	–	✓	✓	✓	✓	✓	–	–
Specification–implementation Violation	✓	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Sloppy Delegation	–	–	–	–	–	–	–	–	–	✓	–	–	–	–	–
Co-change Coupling	–	–	–	–	–	–	–	–	–	✓	–	–	–	–	–
Separation of Concerns	–	–	–	–	–	–	–	–	–	✓	–	–	–	–	–
Concern Overload	–	–	–	–	–	–	–	–	–	✓	–	–	–	–	–
Link Overload	–	–	–	–	–	–	–	–	–	✓	–	–	–	–	–

Table 13), whereas a few (e.g., Designite) provide only limited language support. Most of the supported languages are the compiled languages because there is no overhead of translating the programs into native code at run-time. Only a few interpreted languages (e.g., JavaScript, Python, etc.) are supported by the detection tools. The distribution of language support in terms of development languages is shown in Table 13. It is evident that Java has the most support, while other popular languages (e.g., C++ and C) also have favorable inclusion. However, some development languages (e.g., .Net, Python, and PHP) receive a little attention.

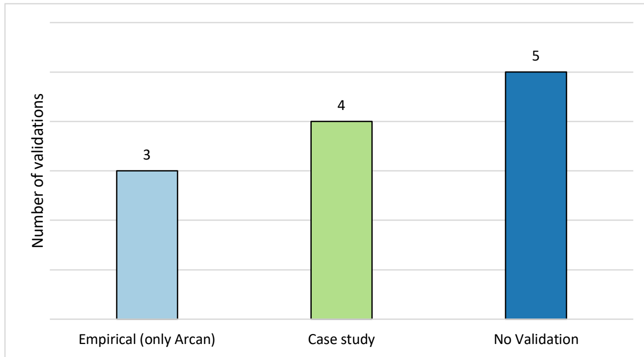

Fig. 7. Validation type in detection tools

Table 12. Tool availability

Tool	Open source	Commercial
Arcan	✓	–
Sonargraph	✓	✓
Structure101	–	✓
CLIO	–	✓
Designite	✓	✓
DV8	–	✓
SonarQube	✓	✓
AI Reviewer	–	✓
ARCADE	✓	–
STAN	–	✓
CAST	–	✓
SODA	✓	–

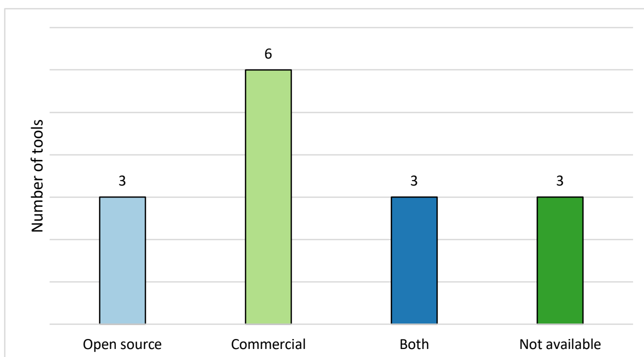

Fig. 8. Availability of detection tools

5.4 Limitations of Detection Techniques and Tools (RQ4)

This section presents a discussion of the limitations of architectural smells detection techniques and tools reported in the literature and identified through the cumulative analysis of the detection techniques (RQ2) and tools (RQ3).

In Table 14, we show, for each architecture smell, whether existing techniques and tools can detect it. The number (in bracket in Table 14) represents the frequency of research articles in which a detection technique each smell is described. We observed some interesting patterns from Table 14. We found that across all technique categories, service-oriented and performance-related architectural smells were the most widely supported. It can also be seen that service-oriented smells are mainly detected by employing rules-based and search-based approaches, whereas performance-related smells were mostly identified through model-driven techniques. We also observed that the architectural smells that encapsulate dependency issues can be identified through almost all of the technique types. For instance, Cyclic Dependency is detected through rules-based, graph-based, design structure matrix, code smells analysis, history-based, and visualization techniques.

In the rest of this section, we discuss the limitations of architecture smell detection techniques and tools, as identified through this mapping study. These include limitations that are derived through the collective mapping of data extracted from the studies, as well as, the ones that are directly reported by study authors.

Undetected Smells – Table 14 shows that there are many smells that are not detected by existing techniques and tools. The undetected smells belong to package, services (including micro-services), MVC, and other architectural-level smells. Although service-oriented smells (e.g., Multi, Tiny, Knot, Data, etc.) are widely studied, there are still many service-specific smells, such as Golden Hammer, Silver Bullet, Brain Controller, etc. that are not detected yet. Some service smells which are not currently detected at a more granular level (micro-services), for instance, Shared Libraries and Shared Persistency. We also saw that the package-related smells received limited attention, and there are several package smells (e.g., Package Instability, Package abstractness, etc.) that need detection. Furthermore, we found a few undetected smells that belong to MVC architecture, for instance, Brain Controller and Brain Repository. Also, some of the abstraction (e.g., Missing and Incomplete), encapsulation (e.g., Leaky and Missing), and hierarchy (e.g., Unnecessary and Speculative) specific smells are not subjected to any investigation. System-level smells (Too Many Subsystems, No Subsystems, and Overgeneralized Subsystems), in Table 14, represent problems with size and generalization in the system's architecture.

We also observed the limited coverage of architectural smells in detection tools. As shown in Table 14, the detection tools mostly provide support for dependency-related and service-specific architectural smells. In the case of other types of architectural smells, tool support is still lacking. The primary papers also pointed out the need to integrate the detection approaches with tools [P49] and improve the scalability of tools in terms of detected smells [P38].

Increasing support for the currently unsupported architectural smells will be challenging. Some of the primary papers reported that not all architectural smells are equally easy to detect because some smells required additional information (e.g., evolution history data [P38]). Therefore, when such information is not available for specific smells in certain datasets, it is challenging to detect them [P38]. Moreover, we noticed that static analysis techniques are commonly applied to detect the architectural smells, but the semantic analysis is rarely employed. We argue that semantic analysis techniques could pave ways to detect those architectural smells that encapsulate semantic information. Future research can continue to study ways to improve support across difficult to detect architectural smells. In addition, increased industry collaboration can also be beneficial to increase the data availability.

Lack of Adequate Quantification of Architectural Smells – We saw that many rules-based approaches were employed to detect architectural smells. However, identifying an appropriate set of metrics and their thresholds is a challenge [P30, P38, P48, P49]. For instance, Ouni et al. [P49] identified thresholds by conducting experiments based on

Table 13. Language support provided by detection tools

Tool	Java	C++	C	C#	Python	.Net	PHP	JavaScript	TypeScript	Go	Swift	COBOL	Apex	Kotlin	Ruby	Scala	HTML	CSS	ABAP	Flex	Objective-C	SQL	VB	XML	Ada	Fortran	JOVIAL	Assembly	F#	JSP	R	Erlang	Unix Scripts	Pascal
Arcan	✓	✓	✓	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Sonargraph	✓	✓	✓	✓	✓	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Structure101	✓	✓	✓	–	✓	✓	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	✓
CLIO	✓	✓	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Designite	✓	–	–	✓	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
DV8	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	✓	–	–	–	–	✓	✓	–	–	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–	–	–	✓
SonarQube	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–	–	–	–	–	–	–	–
AI Reviewer	–	✓	✓	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
ARCADE	✓	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
STAN	✓	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
CAST	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	✓	✓	✓	✓	✓	✓	–	✓	✓	✓	✓	✓	✓	✓
SODA	✓	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–

trial-and-error, which meant that the proper configuration of thresholds was required. This implies there is a lack of adequate quantification of architectural smells.

Lack of Broader Focus on Quality Characteristics – Identifying the maintainability issues of the software systems was the main focus of many of the existing techniques (see Figure 9). Note that this figure excludes articles where the focus was only on architecture quality in general and a specific quality characteristic is not mentioned. The second most investigated quality characteristic was performance. Furthermore, we found only one technique that supported the security smells of the software architecture (see Figure 9). Other quality characteristics, such as usability, reliability, portability, etc. are not the focus of any detection technique or tool, urging the need to broaden the focus to detect architectural smells that impact a wider range of quality characteristics.

Lack of Inclusiveness of Architecture Styles – We observed that some architectural styles undergo more examination, for instance, service-oriented architecture was studied the most—19 research articles (see Figure 10). Note that this figure excludes the articles that did not mention a specific architecture style. It can also be seen that MVC and layered architectures received reasonable investigation, but some architectural styles, such as component and cloud, have limited detection support. It is known that smells can reside in software architectures regardless of the architectural style, meaning that all styles of architecture should receive a fair analysis.

Insufficiency of Empirical Validations – In terms of validation, we observed an even blend of empirical and case studies to evaluate the effectiveness of the techniques. The detection tools (except Arcan) have not been empirically validated—the majority of the tools are evaluated through case studies or not validated (also shown in Figure 7). We found that the validations were performed using a limited number of software projects, which also corresponds to the limitation reported by the primary papers [P30, P38]. This suggests a need to have more empirical validations.

While the use of commercial and open source systems was also evenly distributed across the validations, software projects included in the validations are biased towards a small set of programming languages, such as Java, C#, and C++, which reduces the practical applicability of the techniques and tools in projects that employ other programming languages. Figure 11 illustrates the languages used to validate the techniques. The primary papers also urged the need to include different architectures, domains, and languages to improve generalization [P30, P38].

Limited Language Support – Following on from this, in addition to have limited languages in the projects involved in the validations of the techniques and tools, the tools themselves support only a limited set of languages. Table 13 shows the languages supported by each tool. Java is supported by the most tools, while only a few tools support .Net and PHP. In recent years, Python has become a popular development language among the software community, but we see only a few tools that support Python projects. It is important to note that language scalability can impact the usefulness and adoption of the tools, particularly as languages evolve, the detection tools will need to be updated to accommodate these changes.

Limited Involvement of Industry – We also noticed a scarcity of empirical studies that involve architects and developers from the software industry. Only a small percentage of studies performed a qualitative analysis of feedback obtained from developers using interviews and questionnaires [P14, P36]. To encourage industry adoption of the detection techniques and tools, more studies should look to include software practitioners in the validation.

Additionally, in RQ1, we found no academic and industrial collaborations in authorship of the detection technique and tool papers. However, some of the studies were validated using large scale industrial systems. Still, there is room to increase collaboration with industry in this research area. The limitations of some of the primary papers indicated that the lack of industry collaboration impact construct validity. These papers expressed concerns that developers might disagree with the selected examples of the architectural smells as the best candidates for architectural problems because of their understanding and expertise [P48, P49]. Further validation with industry can alleviate these threats.

Other Limitations – We also found some limitations (reported in the primary papers) that were not identified by our collective analysis of the techniques and tools. Some papers reported potential threats to conclusion validity based on the statistical analysis. For instance, in [P30], the authors noted that most of their results had statistical significance, but that some exceptional cases required further investigation. From an internal validity point of view, the most common threat reported in the primary paper was related to the appropriate selection of evaluation measures (e.g., [P38, P49]).

6 Discussion

In this section, first, we discuss the implications of the findings from the analysis of the detection techniques and tools. Later, we reflect on the significance of the open challenges (discussed in Section 5.4) and highlight possible directions for future research.

6.1 Implications of Findings

Focus on dependency smells – We observed that detecting dependency smells is common among the majority of the techniques. One potential reason for this could be that it is relatively easy to depict dependencies using graph structures or visualizations [P4]. In addition, usually, dependency smells are connected with coupling issues

Table 14. Technique and tool detection as per architectural smell (undetected are in bold). The number represents the frequency of research articles

Smell category	Architectural smell	Rules-based	Graph-based	Design structure matrix	Model-driven	Code smells analysis	Reverse engg. & history	Search-based	Visualization	Others	Tool	Architectural smell	Rules-based	Graph-based	Design structure matrix	Model-driven	Code smells analysis	Reverse engg. & history	Search-based	Visualization	Others	Tool
Service	Low Cohesive Operations	(1)	–	–	–	–	–	–	–	–	–	Multi-service	(5)	–	–	(1)	–	–	–	–	(1)	✓
	Tiny/Nano/Fine-grained Service	(7)	–	–	(1)	–	–	(5)	–	(1)	✓	Chatty Service	(6)	–	–	–	–	–	(5)	–	(1)	✓
	Knot Service	(3)	–	–	–	–	–	–	–	(1)	✓	Bottleneck Service	(4)	–	–	–	–	–	–	–	(1)	✓
	Service Chain	(3)	–	–	–	–	–	–	–	(1)	✓	Sand Pile	(3)	–	–	–	–	–	–	–	–	✓
	Nobody Home	(3)	–	–	–	–	–	–	–	(1)	✓	Duplicated Service	(3)	–	–	–	–	–	–	–	–	✓
	Data Service	(2)	–	–	–	–	–	(2)	–	–	✓	Ambiguous Name	(2)	–	–	–	–	–	–	–	–	–
	Maybe it's not RPC	(2)	–	–	–	–	–	(3)	–	–	–	Redundant PortTypes	(2)	–	–	–	–	–	(3)	–	–	–
	Bloated Service	(1)	–	–	–	–	–	–	–	(1)	✓	Forgetting Hypermedia	(2)	–	–	–	–	–	–	–	–	–
	Crudy Interface	(3)	–	–	–	–	–	(3)	–	–	–	Ignoring MIME Types	(2)	–	–	–	–	–	–	–	–	–
	Crudy URI	(1)	–	–	–	–	–	–	–	–	–	Ignoring Caching	(1)	–	–	–	–	–	–	–	–	–
	Pipe and Filter	–	–	–	(1)	–	–	–	–	–	–	**Golden Hammer**	–	–	–	–	–	–	–	–	–	–
	API Versioning	–	–	–	–	–	–	–	–	–	–	**Silver Bullet**	–	–	–	–	–	–	–	–	–	–
	ESB Usage	–	–	–	–	–	–	–	–	–	–	**Hard-coded Endpoints**	–	–	–	–	–	–	–	–	–	–
	Micro-service Greedy	–	–	–	–	–	–	–	–	–	–	**Not Having an API Gateway**	–	–	–	–	–	–	–	–	–	–
	Shared Libraries	–	–	–	–	–	–	–	–	–	–	**Shared Persistency**	–	–	–	–	–	–	–	–	–	–
	Too Many Standards	–	–	–	–	–	–	–	–	–	–	**Wrong Cuts**	–	–	–	–	–	–	–	–	–	–
	Overstandardized SOA	–	–	–	–	–	–	–	–	–	–	**Nothing New**	–	–	–	–	–	–	–	–	–	–
	Big Bang	–	–	–	–	–	–	–	–	–	–	**Incomplete Service**	–	–	–	–	–	–	–	–	–	–
	No Legacy	–	–	–	–	–	–	–	–	–	–	**Shiny Nickel**	–	–	–	–	–	–	–	–	–	–
Performance	Excessive Dynamic Allocation	–	–	–	(3)	–	–	–	–	–	–	Ramp	–	–	–	(4)	–	–	–	–	–	–
	Tower of Babel	–	–	–	(1)	–	–	–	–	(1)	–	More is Less	–	–	–	(1)	–	–	–	–	–	–
	One-lane Bridge	–	–	–	(5)	–	–	–	(1)	–	–	Unbalanced Processing	–	–	–	(4)	–	–	–	(1)	–	–
	Traffic Jam	–	–	–	(5)	–	–	–	(1)	–	–	Circuitous Treasure Hunt	–	–	–	(3)	(1)	–	–	(1)	(1)	–
	Empty Semi-trucks	–	–	–	(2)	(1)	–	–	(1)	–	–	Extensive Processing	–	–	–	(1)	–	–	–	–	(1)	–
Dependency	Unauthorized Dependency	–	(2)	–	–	–	–	–	–	–	–	Unstable Dependency	–	–	–	–	–	(1)	–	–	(2)	✓
	Hub-like Dependency	–	(3)	–	–	–	(1)	–	–	(2)	✓	Cyclic Dependency	(1)	(3)	(1)	–	(4)	(1)	–	(1)	(3)	✓
	Implicit Cross-module Dependency	–	–	(3)	–	–	–	–	–	(1)	✓	Cycles between Namespaces	–	(1)	–	–	–	–	–	–	–	–
Package	Package Cycle	–	–	(6)	(1)	–	–	–	–	–	✓	Clique	–	–	(3)	–	–	–	–	–	–	✓
	Too Small Package	–	–	–	–	–	–	–	–	–	–	**Unused Package**	–	–	–	–	–	–	–	–	–	–
	Unclear Package Name	–	–	–	–	–	–	–	–	–	–	**Package Instability**	–	–	–	–	–	–	–	–	–	–
	Package Abstractness	–	–	–	–	–	–	–	–	–	–	**Unbalanced Package Hierarchy**	–	–	–	–	–	–	–	–	–	–
MVC	Sloppy Delegation	–	(1)	–	–	–	–	–	–	(1)	✓	Co-change Coupling	–	(1)	–	–	–	–	–	–	(1)	✓
	Separation of Concerns	–	(2)	–	–	–	–	–	–	–	✓	Concern Overload	–	(1)	–	–	(6)	–	–	–	(3)	✓
	Scattered Functionality	–	(1)	–	–	(4)	–	–	–	(3)	✓	Feature Concentration	–	(1)	–	–	–	–	–	–	–	–
	Link Overload	–	(1)	–	–	–	–	–	–	(1)	✓	Connector Envy	–	–	–	–	(6)	–	–	–	(2)	–
	Fat Repository	–	–	–	–	–	–	–	–	–	–	**Promiscuous Controller**	–	–	–	–	–	–	–	–	–	–
	Brain Controller	–	–	–	–	–	–	–	–	–	–	**Meddling Service**	–	–	–	–	–	–	–	–	–	–
	Brain Repository	–	–	–	–	–	–	–	–	–	–	**Laborious Repository Method**	–	–	–	–	–	–	–	–	–	–
Component	Anchor Submission	–	–	(1)	–	–	–	–	–	–	–	Anchor Dominant	–	–	(1)	–	–	–	–	–	–	–
	Non-transfer Communication	–	–	–	–	–	(1)	–	–	–	–	Unauthorized Call	–	–	–	–	–	(1)	–	–	–	–
	Undercover Transfer Object	–	–	–	–	–	(1)	–	–	–	–	Subtype Knowledge	–	(1)	–	–	–	–	–	✓	–	–
	Abstraction without Decoupling	–	(1)	–	–	–	–	–	–	–	✓	Blob/God Object	(3)	–	–	(4)	(1)	–	(3)	(1)	(1)	–
Other smells	Unhealthy Inheritance	–	–	(5)	(1)	–	–	–	–	–	✓	Improper Inheritance	–	–	(2)	–	–	–	–	–	–	–
	Unstable Interface	–	–	(7)	–	–	–	–	–	–	✓	Unused Interface	–	(1)	–	–	(4)	–	–	–	(1)	✓
	Ambiguous Interface	–	–	–	–	(6)	–	–	–	(1)	✓	Interface Violation	–	–	–	–	–	(1)	–	–	–	–
	Degenerated Inheritance	–	(1)	–	–	–	–	–	–	–	–	Modularity Violations	(1)	–	(5)	–	–	–	–	–	(1)	✓
	Misplaced Component	–	–	–	–	–	–	–	(1)	–	–	Security Flaws	–	(1)	–	–	–	–	–	–	–	–
	Incomplete Abstraction	–	–	–	–	–	–	–	–	–	–	**Missing Abstraction**	–	–	–	–	–	–	–	–	–	–
	Missing Encapsulation	–	–	–	–	–	–	–	–	–	–	**Leaky Encapsulation**	–	–	–	–	–	–	–	–	–	–
	Speculative Hierarchy	–	–	–	–	–	–	–	–	–	–	**Unnecessary Hierarchy**	–	–	–	–	–	–	–	–	–	–
	Too Many Subsystems	–	–	–	–	–	–	–	–	–	–	**Overgeneralized Subsystems**	–	–	–	–	–	–	–	–	–	–
	No Subsystems	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–

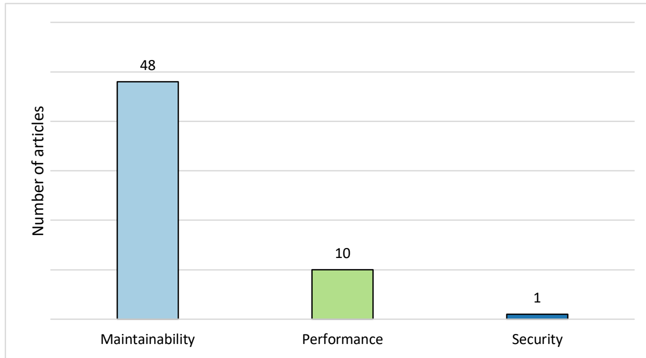

Fig. 9. Frequency of the quality characteristics investigated by the detection techniques

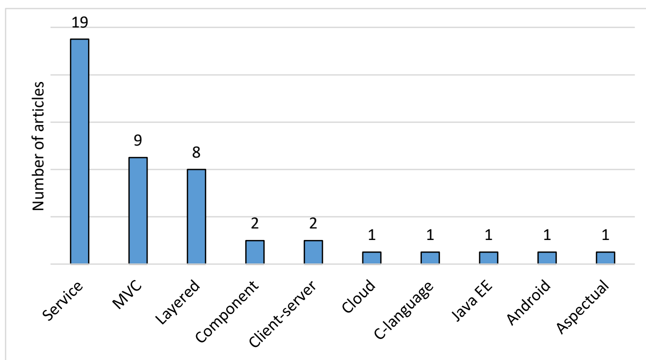

Fig. 10. Frequency of the architecture styles investigated by the detection techniques

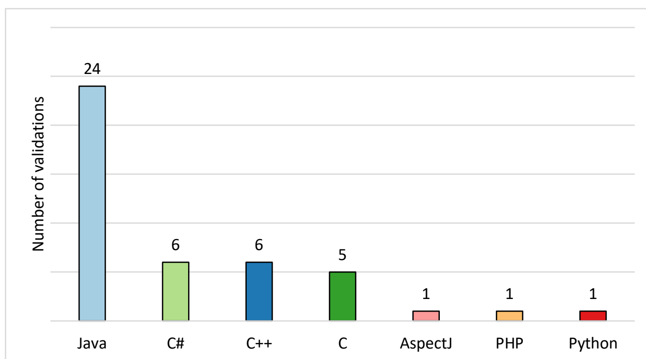

Fig. 11. Development languages of the projects used for validations

in the architecture, which is related to maintainability. Since we have seen that, in most of the primary papers, researchers are interested in maintainability of software systems (see Figure 9), this could also be a reason for detecting dependency smells, since these can have serious impacts on maintainability. Finally, in terms of practical relevance, the software industry is mostly interested in reducing the maintenance cost [7]. We know that dependency smells hinder the maintainability characteristic of a system [P2], therefore, it is more relevant for the software industry to detect them. This argument is also supported by the smell coverage provided by detection tools. Most of the detection tools can detect common dependency smells. This could be because of industry demand, which would lead to higher tool adoption.

Many search-based techniques for service-oriented smells – We also noticed that most of the service-oriented smells are detected through search-based techniques. This could be driven by the scale and diversity of the data, as search-based techniques are suitable for large and diverse datasets [P49]. For example, search-based techniques use the architectural smells data (metrics and their thresholds) in many software systems to optimize the detection [P48, P49]. It is also noteworthy that large-scale web services were used in the evaluations of the search-based techniques, further illustrating their ability to work with large-scale and diverse data.

Focus on maintainability quality – In terms of quality characteristics, maintainability is the focus in the majority of the detection techniques. This could be because of the innate relationship between smells and the maintainability of a software [9]. Another reason could be the desire to reduce the development costs associated with software maintenance. However, other software qualities like performance and security can be impacted by architectural smells, and these qualities have received much less attention in this research area.

Diversity in the detection techniques – One important finding is the lack of diversity in the detection techniques. Comparably, code smells analysis techniques are more diverse in terms of the investigated architecture styles (MVC, layered, and cloud). Additionally, they are also more diverse in the project domains (SPL, web, and middleware). These diversities in code smells analysis techniques make them more applicable to a wider range of projects. We observed that some techniques (e.g., DSM, search-based, and visualization) employed commercial systems and used multiple languages in their evaluations, which make them more scalable and generalizable. However, many architectural smells detection tools and methods (e.g., rules-based, graph-based, and code smells analysis) have less evidence of scalability and generalizability in the literature.

6.2 Open Challenges and Future Research Directions

Many smells still need detection – From the limitations, we found several architectural smells that are not detected by existing techniques or tools. The coverage of architectural smells in tools is even more limited, which can be a road-blocker for them to be employed in software industry. The undetected smells vary in terms of their granularity level. For instance, in object-oriented architecture, the undetectable smells exist at the class-level, package-level, interface-level, etc. The smells at these different granularity levels are inter-related. For example, Package Instability (a package-level smell) can raise coupling issues that affect communication between classes (a class-level smell). This suggests the need for equal attention regardless of their level because a smell at one level can affect (i.e., introducing another smell) the design structure of the other level.

We have seen a variety of undetected architectural smells , but how they should be prioritized depends on various aspects, such as impact on quality characteristics, complexity of refactoring, etc. Software organizations may have different quality goals; for instance, some focus on maintainability and some on security. Therefore, they prioritize the smells according to their quality goals. In the literature, the papers that describe the currently undetected architectural smells have reflected on the software quality problems which can appear if the smells are left undetected. For instance, Taibi and Lenarduzzi [47] explained that the "Hard-coded Endpoints" smell can introduce issues in a micro-services environment. If micro-services are connected with Hard-coded End-

RQ1 Demographics	RQ2 Detection Techniques	RQ3 Detection Tools	RQ4 Limitations
• Trend of investigating architectural smells started in 2010. • Maximum number of studies appears in 2018 and 2019. • Most articles are published in conferences. • Conferences on software architecture, technical debt, and ICSE index many articles on architectural smells detection. • All the active SE researchers have academic affiliations.	• Rules-based techniques are mostly employed. • Almost all the technique types investigate dependency-related smells. • Maintainability is the main quality attribute studied across all technique types. • Mostly the Java-based projects are used for evaluations.	• Tools mostly detect dependency-related smells. • Graph-based techniques are mainly employed in tools. • Only one tool (SODA) detects service-oriented smells. • Only one tool (Arcan) is empirically validated. • Mainly Java projects are supported by tools, followed by C++ and C.	• Many architectural smells still need investigation. • Lack of broader focus on quality attributes and architecture styles. • Limited involvement of developers in the evaluation process. • Academic and industrial collaboration needs to be increased. • Limited language support provided by detection tools. • Open source tools with full functionalities are limited.

Fig. 12. Main findings of our RQs

points, it is problematic to change their locations, resulting in slowing down the maintainability process of software applications. As another instance, Package Instability can also impact maintainability because of high dependency between packages—the changes in one package for changes in another package. Similarly, in many papers, researchers have identified several quality characteristics (e.g., modularity, evolvability, modifiability, etc.) that could be impacted by the architectural smells [5,13,17,26,28,31,45,47]. Therefore, in most cases, the priority of detecting a particular smell lies in its severity and degree of impact on different software quality characteristics, and how important those quality characteristics are to software organizations. Another important aspect to prioritize the smells is based on the level of complexity involved in their refactorings. The complexity of the required refactoring is considered because if refactoring involves cumbersome tasks, more resources would be required to remove the smell. Future research should consider these impacts when prioritizing the smells to study for new detection methods and tools.

Conduct experiments to quantify undetected smells – We found that it was a challenge to adequately quantify and measure architectural smells. Some studies employed the metrics for code smells to determine their relations with architectural technical debt [24,29]. A major drawback of entirely relying on code metrics is losing the semantics of architectural smells. To avoid losing the semantic information of architecture, other methods, such as statistical analysis of software data (e.g., size, complexity, etc.) [23] and ROC (Receiver Operating Characteristic) curves [44] can be adopted for identifying thresholds for architecture data. Therefore, we believe future studies should focus on identifying appropriate thresholds and metrics that can be used to quantify architectural smells better.

Measure the impact on a broader set of quality characteristics – Maintainability was the most commonly studied quality characteristic when considering architecture smell detection. It can give insight into the effort and resources required to fix the problems resulting from the smells. The second most investigated quality characteristic was performance. However, the detection techniques that consider other quality characteristics (e.g., security) are limited. We believe that it is important to understand the impact of architectural smells on other quality characteristics, since all are important for producing high quality software. For example, even if software is maintainable, if it has large security problems, it would not be considered high-quality software.

In addition, it is paramount to individually investigate the sub-characteristics of a quality characteristic. For instance, in maintainability, we observed that modularity and modifiability were focused during detection but sub-characteristics like testability or reusability were not given any attention. Future work could investigate ways to measure these quality sub-characteristics in the detection methods. This would require examining the relationship between different smells and sub-characteristics (e.g., reusability). The relationship could be established by performing empirical analysis of the impact of architectural smells on a particular quality characteristic. The advantage of such analysis is that it would allow the tool developers to focus on a particular set of architectural smells that impact a specific type of quality characteristic.

Investigate the quality of other popular architectural styles – We observed that some architectural styles undergo more examination, for instance, service-oriented architecture. It is likely due to the popularity of using service-oriented architecture in recent years. However, architectural styles, such as component and cloud, are barely explored. The current spread of cloud technology and the shift of many technologies to the cloud environment suggest the significance of investigating the quality of cloud architecture. Future work could focus on understudied architectural styles to improve detection techniques across a broader range of architectures.

Enhance the diversity in empirical validations – Validation is the key to show the applicability of an approach, and the data is a vital ingredient in the validation process. In the existing approaches and tools we first noticed the lack of diversity in the empirical evaluations, for instance, less validations with a broader set of programming languages. This can bring many challenges because of the different styles and practices offered by the development languages, thus, not all detection techniques and tools will be able to be easily applied to additional programming languages. However, to ensure high-quality architecture across a wide range of software systems, future research must investigate ways to expand the detection techniques and tools (and their evaluation) across a wider range of programming languages. We suggest to include those development languages that are also common in software development. For instance, although .Net and PHP are popular development languages, we noticed that only a few tools support .Net and PHP projects. Similarly, the Python language is also supported by only a handful of tools. Future work can focus on improving the existing tools with an enhanced set of supported languages.

Need more scalable techniques and tools – In addition to supporting a wider range of programming languages, detection techniques and tools need to be empirically evaluated using realistic data to ensure scalability. For instance, in our results, we noticed that tools (except one) were either validated with case studies (33%) or not evaluated at all (42%). It is interesting to see that some tools (DV8, sonarQube, and CAST) can work with many programming languages, but there is scarcity of empirical evidence in the literature regarding their effectiveness. This shows the need to improve the scalability of the detection methods with more empirical evaluations and with industrial projects. Future studies should look to use industrial-scale commercial projects in the evaluation process of detection techniques and tools.

Involve software developers in evaluations - In the evaluation of detection techniques and tools, we found only a few studies that performed a qualitative analysis of feedback obtained from developers. We argue that there is no replacement of including software developers

in the process of evaluating the detection techniques. The input of developers in the evaluation processes of architectural smells would enhance the applicability in real-world scenarios.

Need a shift to academic–industrial collaborations – Related to the need to evaluate techniques and tools with industrial-scale data and a need for greater involvement of software practitioners in evaluations, we noticed that all the detection techniques and tools originated from academics in terms of authorship. Increasing academic and industrial collaboration in the creation of architecture techniques and tools can help to minimize the gap between research and practice and encourage better adoption of the proposed techniques and tools by software practitioners.

7 Threats to Validity

In this section, we present threats to validity based on the guidelines provided by Petersen et al. [39].

Descriptive Validity – Descriptive validity refers to the accurate and objective description of observations. To reduce this threat, we recorded the data (article ID, article title, publication year, publication venue, and analysis factors) in a tabular representation. Article ID, article title, publication year, source (digital libraries), and publication venue were automatically recorded from the searched databases, whereas analysis factors were extracted by objectively analyzing the selected articles. In addition, the second and third authors of this paper also independently performed the data extraction process on a subset (just over 15%) of the primary papers. Any disagreements were resolved through iterative discussions between all authors. After these discussions and agreements were reached on how data should be extracted and categorised, the data was extracted from the remaining 85% of the primary papers. We also distinguished the technique and tool articles to support the recording of the data and extraction process. All of the detailed data and classifications for each paper are provided in our supplementary material[1].

Theoretical Validity – Theoretical validity refers to our ability to capture intended data, keeping into consideration the biases and selection of subjects. The theoretical validity threats in our systematic mapping study were mainly originating from the incompleteness of the searched literature domain, inaccuracy in the gathered data, and inaccuracy in the data synthesis process.

The completeness of the literature domain depends on the electronic databases, search string, and inclusion and exclusion criteria. Due to a high number of electronic databases, it is not feasible to search through every available electronic database. Therefore, in this mapping study, we search seven well-known databases (Scopus, Web of Science, INSPEC, ACM Digital Library, IEEEXplore, SpringerLink, and DBLP) of software engineering and computer science literature.

Another limitation comes from the search string (used to retrieve the related articles) because relevant publications could be omitted by it. We confined our search string to the keywords that are the most relevant to architectural smells detection. The number of keywords with the use of appropriate logical operators in the search string could increase the number of retrieved articles, but then this might also result in a substantial number of irrelevant publications. To reduce the threat that our search string was missing relevant articles, we validated our search string by performing a focused search analysis of the papers published in *International Conference on Software Engineering* (ICSE) from 2016 to 2019. Using our search string and snowballing, we were able to find all articles from this small validation set. While we cannot guarantee that we identified every relevant paper, this gives us confidence that our dataset includes many articles of interest.

Some limitations could also be imposed by our inclusion and exclusion criteria because ill-designed or incomplete inclusion and exclusion criteria could result in an incorrect selection of articles. To mitigate this threat, we iteratively modified the exclusion criteria during the selection process to ensure that no relevant articles were discarded and irrelevant articles were not included.

The analysis of the selected articles was performed in terms of various factors in the analysis framework. We designed the analysis framework to cover the main aspects of the detection techniques and tools to answer our research questions. We do not claim this analysis framework to be complete. Additional factors could be introduced to improve the comprehensiveness of the analysis process. However, the data we extracted has provided answers to our research questions and helped us identify many interesting findings when considering the research literature as a whole.

Finally, a threat is connected to the biases of the author, who was executing the extraction and classification process. This threat was mitigated by regular discussions about the data extraction and classification performed by the author with other authors of this mapping study.

Interpretive validity – This refers to the conclusions drawn based on the given data. A researcher's biases could be a threat in interpreting the data. For instance, a researcher with experience in conducting case studies might misinterpret the results from other types of validations. This threat was mitigated by multiple rounds of discussions between authors on the conclusions drawn from our mapping study.

Repeatability – This refers to providing sufficient information about the research process to ensure the experiments could be repeated. We strived to ensure repeatability by providing detailed reporting of our mapping process and providing all of the detailed classifications for each article as supplementary material. We also used existing guidelines for conducting systemic mapping studies in software engineering to strengthen the repeatability.

8 Conclusion

In this paper, we performed a systematic mapping study of architectural smells detection techniques and tools. From the analysis of the related literature, we highlighted some key findings. We observed that service-oriented, performance-related, and dependency-originated architectural smells are widely detected. We also observed that there are still many architectural smells related to packages and services that are not detected by existing techniques and tools. Moreover, some of the architectural smells that are related to quality principles (such as abstraction, encapsulation, and hierarchy) lack detection techniques and tools. We also noticed the scarcity of empirical evaluations of detection techniques and tools, especially with large scale industrial projects. The selection of software projects for evaluation is also leaned towards programming languages, such as Java, C, C++, and C#, suggesting the need to include other development languages as well.

Based on our findings, we suggested some future research directions. We emphasize having a more in-depth analysis of architectural smells at different granularity levels, such as investigating the package instability and its impact on the communication between classes. Similarly, we suggest looking at the abstraction properties of the interfaces to understand the impact on the overall architecture of the software. Another promising future research direction is the identification of software metrics and their thresholds for the currently undetected architectural smells. Furthermore, accurate mapping of metrics to quality characteristics can open many ways in which quality characteristics can be measured and evaluated. We also suggest the inclusion of architectural styles that are currently adopted in industrial projects, such as cloud, micro-services, etc. Another future work could be conducting empirical validations with real-world projects covering a broader set of projects belonging to different domains and development languages. Lastly, for all the potential research directions mentioned in this study, we recommend focusing on the applicability and usefulness for the viewpoint of the software development industry.

References

[1] Software product quality model ISO/IEC 25010. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010. Last Accessed: 2020-04-22.

[2] K. Alkharabsheh, Y. Crespo, E. Manso, and J. A. Taboada. Software design smell detection: A systematic mapping study. *Software Quality Journal*, 27(3):1069–1148, 2019. doi: 10.1007/s11219-018-9424-8

[3] N. S. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman. Identification and management of technical debt: A systematic mapping study. *Information and Software Technology*, 70:100–121, 2016. doi: 10.1016/j.infsof.2015.10.008

[4] A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou. The financial aspect of managing technical debt: A systematic literature review. *Information and Software Technology*, 64:52–73, 2015. doi: 10.1016/j.infsof.2015.04.001

[5] M. Aniche, G. Bavota, C. Treude, M. A. Gerosa, and A. van Deursen. Code smells for Model-View-Controller architectures. *Empirical Software Engineering*, 23(4):2121–2157, 2018. doi: 10.1007/s10664-017-9540-2

[6] A. Bandi, B. J. Williams, and E. B. Allen. Empirical evidence of code decay: A systematic mapping study. In *20th Working Conference on Reverse Engineering*, pp. 341–350. IEEE, 2013. doi: 10.1109/WCRE.2013.6671309

[7] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig. Software complexity and maintenance costs. *Communications of the ACM*, 36(11):81–95, 1993. doi: 10.1145/163359.163375

[8] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 2003.

[9] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, 107:1–14, 2015. doi: 10.1016/j.jss.2015.05.024

[10] I. M. Bertran. Detecting architecturally-relevant code smells in evolving software systems. In *Proceedings of the 33rd International Conference on Software Engineering*, pp. 1090–1093. IEEE, 2011. doi: 10.1145/1985793.1986003

[11] T. Besker, A. Martini, and J. Bosch. A systematic literature review and a unified model of ATD. In *42th EUROMICRO Conference on Software Engineering and Advanced Applications*, pp. 189–197. IEEE, 2016. doi: 10.1109/SEAA.2016.42

[12] T. Besker, A. Martini, and J. Bosch. Managing architectural technical debt: A unified model and systematic literature review. *Journal of Systems and Software*, 135:1–16, 2018. doi: 10.1016/j.jss.2017.09.025

[13] J. Bogner, T. Boceck, M. Popp, D. Tschechlov, S. Wagner, and A. Zimmermann. Towards a collaborative repository for the documentation of service-based antipatterns and bad smells. In *Proceedings of the IEEE International Conference on Software Architecture Companion*, pp. 95–101. IEEE, 2019. doi: 10.1109/ICSA-C.2019.00025

[14] A. Cavacini. What is the best database for computer science journal articles? *Scientometrics*, 102(3):2059–2071, 2015. doi: 10.1007/s11192-014-1506-1

[15] E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE software*, 7(1):13–17, 1990. doi: 10.1109/52.43044

[16] J. Clarke, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, et al. Reformulating software engineering as a search problem. *IEE Proceedings - Software*, 150(3):161–175, 2003. doi: 10.1049/ip-sen:20030559

[17] H. S. de Andrade, E. Almeida, and I. Crnkovic. Architectural bad smells in software product lines: An exploratory study. In *Proceeding of the 11th Working IEEE/IFIP Conference on Software Architecture*, pp. 1–6. ACM, 2014. doi: 10.1145/2578128.2578237

[18] E. V. de Paulo Sobrinho, A. De Lucia, and M. de Almeida Maia. A systematic literature review on bad smells 5W's: Which, When, What, Who, Where. *IEEE Transactions on Software Engineering*, 2018. doi: 10.1109/TSE.2018.2880977

[19] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian. Selecting empirical methods for software engineering research. In *Guide to Advanced Empirical Software Engineering*, pp. 285–311. Springer, 2008. doi: 10.1007/978-1-84800-044-5_11

[20] S. D. Eppinger and T. R. Browning. *Design Structure Matrix Methods and Applications*. MIT press, 2012.

[21] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo. A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, pp. 1–12, 2016. doi: 10.1145/2915970.2915984

[22] C. Fernández-Sánchez, J. Garbajosa, A. Yagüe, and J. Perez. Identification and analysis of the elements required to manage technical debt by means of a systematic mapping study. *Journal of Systems and Software*, 124:22–38, 2017. doi: 10.1016/j.jss.2016.10.018

[23] K. A. Ferreira, M. A. Bigonha, R. S. Bigonha, L. F. Mendes, and H. C. Almeida. Identifying thresholds for object-oriented software metrics. *Journal of Systems and Software*, 85(2):244–257, 2012. doi: 10.1016/j.jss.2011.05.044

[24] F. A. Fontana, V. Ferme, and M. Zanoni. Towards assessing software architecture quality by exploiting code smell relations. In *Proceedings of the 2nd International Workshop on Software Architecture and Metrics*, pp. 1–7. IEEE Press, 2015. doi: 10.1109/SAM.2015.8

[25] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2018.

[26] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Toward a catalogue of architectural bad smells. In *Proceeding of the International Conference on the Quality of Software Architectures*, pp. 146–162. Springer, 2009. doi: 10.1007/978-3-642-02351-4_10

[27] A. Jedlitschka, M. Ciolkowski, and D. Pfahl. Reporting experiments in software engineering. In *Guide to advanced empirical software engineering*, pp. 201–228. Springer, 2008. doi: 10.1007/978-1-84800-044-5_8

[28] J. Král and M. Žemlicka. Popular SOA antipatterns. In *Proceeding of the Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*, pp. 271–276. IEEE, 2009. doi: 10.1109/ComputationWorld.2009.80

[29] J. Lenhard, M. Blom, and S. Herold. Exploring the suitability of source code metrics for indicating architectural inconsistencies. *Software Quality Journal*, 27(1):241–274, 2019. doi: 10.1007/s11219-018-9404-z

[30] Z. Li, P. Avgeriou, and P. Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220, 2015. doi: 10.1016/j.jss.2014.12.027

[31] M. Lippert and S. Roock. *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. John Wiley & Sons, 2006.

[32] F. Losavio, L. Chirinos, N. Lévy, and A. Ramdane-Cherif. Quality characteristics for software architecture. *Journal of Object Technology*, 2(2):133–150, 2003. doi: 10.5381/jot.2003.2.2.a2

[33] F. Miranda and C. Abreu. *Handbook of Research on Computational Simulation and Modeling in Engineering*. Engineering Science Reference, 2016.

[34] M. Misbhauddin and M. Alshayeb. UML model refactoring: A systematic literature review. *Empirical Software Engineering*, 20(1):206–251, 2015. doi: 10.1007/s10664-013-9283-7

[35] H. Mumtaz, M. Alshayeb, S. Mahmood, and M. Niazi. A survey on UML model smells detection techniques for software refactoring. *Journal of Software: Evolution and Process*, 31(3):e2154, 2019. doi: 10.1002/smr.2154

[36] H. Mumtaz, F. Beck, and D. Weiskopf. Detecting bad smells in software systems with linked multivariate visualizations. In *Proceedings of the IEEE Working Conference on Software Visualization*, pp. 12–20. IEEE, 2018. doi: 10.1109/VISSOFT.2018.00010

[37] H. Mumtaz, S. Latif, F. Beck, and D. Weiskopf. Exploranative code quality documents. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):1129–1139, 2019. doi: 10.1109/TVCG.2019.2934669

[38] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson. Systematic mapping studies in software engineering. In *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*, pp. 1–10, 2008. doi: 10.14236/ewic/EASE2008.8

[39] K. Petersen, S. Vakkalanka, and L. Kuzniarz. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, 64:1–18, 2015. doi: 10.1016/j.infsof.2015.03.007

[40] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131, 2009. doi: 10.1007/s10664-008-9102-8

[41] F. Sabir, F. Palma, G. Rasool, Y.-G. Guéhéneuc, and N. Moha. A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems. *Software: Practice and Experience*, 49(1):3–39, 2019. doi: 10.1002/spe.2639

[42] T. Sharma, P. Singh, and D. Spinellis. An empirical investigation on the relationship between design and architecture smells. *Empirical Software Engineering*, 2020. doi: 10.1007/s10664-020-09847-2

[43] T. Sharma and D. Spinellis. A survey on software smells. *Journal of Systems and Software*, 138:158–173, 2018. doi: 10.1016/j.jss.2017.12.034

[44] R. Shatnawi, W. Li, J. Swain, and T. Newman. Finding software metrics threshold values using ROC curves. *Journal of Software Maintenance and Evolution: Research and Practice*, 22(1):1–16, 2010. doi: 10.1002/smr.404

[45] C. U. Smith and L. G. Williams. Software performance antipatterns. In *Proceedings of the 2nd International Workshop on Software and Performance*, pp. 127–136, 2000. doi: 10.1145/350391.350420

[46] G. Suryanarayana, G. Samarthyam, and T. Sharma. Refactoring for software design smells. *ACM SIGSOFT Software Engineering Notes*, 40, 2015. doi: 10.1016/C2013-0-23413-9

[47] D. Taibi and V. Lenarduzzi. On the definition of microservice bad smells.

IEEE Software, 35(3):56–62, 2018. doi: 10.1109/MS.2018.2141031

[48] E. Tom, A. Aurum, and R. Vidgen. An exploration of technical debt. *Journal of Systems and Software*, 86(6):1498–1516, 2013. doi: 10.1016/j.jss.2012.12.052

[49] G. Vale, E. Figueiredo, R. Abílio, and H. Costa. Bad smells in software product lines: A systematic review. In *8th Brazilian Symposium on Software Components, Architectures and Reuse*, pp. 84–94. IEEE, 2014. doi: 10.1109/SBCARS.2014.21

[50] R. Verdecchia, I. Malavolta, and P. Lago. Architectural technical debt identification: The research landscape. In *Proceedings of the IEEE/ACM International Conference on Technical Debt*, pp. 11–20. IEEE/ACM, 2018. doi: 10.1145/3194164.3194176

[51] C. Wohlin. 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*, p. 38. Citeseer, 2014. doi: 10.1145/2601248.2601268

[52] S. Wong, Y. Cai, M. Kim, and M. Dalton. Detecting software modularity violations. In *Proceedings of the 33rd International Conference on Software Engineering*, pp. 411–420, 2011. doi: 10.1145/1985793.1985850

PRIMARY REFERENCES

[P1] D. Arcelli, V. Cortellessa, D. Di Pompeo, R. Eramo, and M. Tucci. Exploiting architecture/runtime model-driven traceability for performance improvement. In *Proceedings of the IEEE International Conference on Software Architecture*, pp. 81–90. IEEE, 2019. doi: 10.1109/ICSA.2019.00017

[P2] F. Arcelli Fontana, P. Avgeriou, I. Pigazzini, and R. Roveda. A study on architectural smells prediction. In *EUROMICRO Conference Series on Software Engineering and Advanced Applications*, 2019. doi: 10.1109/SEAA.2019.00057

[P3] U. Azadi, F. A. Fontana, and D. Taibi. Architectural smells detected by tools: A catalogue proposal. In *Proceedings of the International Conference on Technical Debt*. IEEE, 2019. doi: 10.1109/TechDebt.2019.00027

[P4] D. Baum, J. Dietrich, C. Anslow, and R. Müller. Visualizing design erosion: How big balls of mud are made. In *Proceedings of the IEEE Working Conference on Software Visualization*, pp. 122–126. IEEE, 2018. doi: 10.1109/VISSOFT.2018.00022

[P5] A. Biaggi, F. A. Fontana, and R. Roveda. An architectural smells detection tool for C and C++ projects. In *Proceedings of the 44th EUROMICRO Conference on Software Engineering and Advanced Applications*, pp. 417–420. IEEE, 2018. doi: 10.1109/SEAA.2018.00074

[P6] J. Brondum and L. Zhu. Visualising architectural dependencies. In *Proceedings of the 3rd International Workshop on Managing Technical Debt*, pp. 7–14. IEEE Press, 2012. doi: 10.1109/MTD.2012.6226003

[P7] Y. Cai and R. Kazman. Software architecture health monitor. In *Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities*, pp. 18–21. ACM, New York, NY, USA, 2016. doi: 10.1145/2896935.2896940

[P8] Y. Cai and R. Kazman. DV8: Automated architecture analysis tool suites. In *Proceedings of the 2nd International Conference on Technical Debt*, pp. 53–54. IEEE Press, 2019. doi: 10.1109/TechDebt.2019.00015

[P9] V. Cortellessa, M. De Sanctis, A. Di Marco, and C. Trubiani. Enabling performance antipatterns to arise from an ADL-based software architecture. In *Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, pp. 310–314. IEEE, 2012. doi: 10.1109/WICSA-ECSA.212.51

[P10] V. Cortellessa, A. Di Marco, and C. Trubiani. Performance antipatterns as logical predicates. In *Proceedings of the 15th IEEE International Conference on Engineering of Complex Computer Systems*, pp. 146–156. IEEE, 2010. doi: 10.1109/ICECCS.2010.44

[P11] V. Cortellessa, A. Di Marco, and C. Trubiani. An approach for modeling and detecting software performance antipatterns based on first-order logics. *Software & Systems Modeling*, 13(1):391–432, 2014. doi: 10.1007/s10270-012-0246-z

[P12] W. Czabanski, M. Bruntink, and P. Martin. Actionable measurements-improving the actionability of architecture level software quality violations. In *Proceedings of the 17th Belgium-Netherlands Software Evolution Workshop*, pp. 51–55, 2018.

[P13] M. De Sanctis, C. Trubiani, V. Cortellessa, A. Di Marco, and M. Flamminj. A model-driven approach to catch performance antipatterns in ADL specifications. *Information and Software Technology*, 83:35–54, 2017. doi: 10.1016/j.infsof.2016.11.008

[P14] S. S. de Toledo, A. Martini, A. Przybyszewska, and D. I. Sjøberg. Architectural technical debt in microservices: A case study in a large company. In *Proceedings of the IEEE/ACM International Conference on Technical Debt*, pp. 78–87. IEEE, 2019. doi: 10.1109/TechDebt.2019.00026

[P15] J. A. Díaz-Pace, A. Tommasel, and D. Godoy. Towards anticipation of architectural smells using link prediction techniques. In *Proceedings of the IEEE 18th International Working Conference on Source Code Analysis and Manipulation*, pp. 62–71. IEEE, 2018. doi: 10.1109/SCAM.2018.00015

[P16] J. Dietrich, C. McCartin, E. Tempero, and S. M. A. Shah. On the existence of high-impact refactoring opportunities in programs. In *Proceedings of the 35th Australasian Computer Science Conference*, vol. 122, pp. 37–48. Australian Computer Society, Inc., 2012.

[P17] U. Eliasson, A. Martini, R. Kaufmann, and S. Odeh. Identifying and visualizing architectural debt and its efficiency interest in the automotive domain: A case study. In *Proceedings of the IEEE 7th International Workshop on Managing Technical Debt*, pp. 33–40. IEEE, 2015. doi: 10.1109/MTD.2015.7332622

[P18] F. A. Fontana, I. Pigazzini, C. Raibulet, S. Basciano, and R. Roveda. PageRank and criticality of architectural smells. In *Proceedings of the 13th European Conference on Software Architecture*, vol. 2, pp. 197–204. ACM, 2019. doi: 10.1145/3344948.3344982

[P19] F. A. Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, and E. Di Nitto. Arcan: A tool for architectural smells detection. In *Proceedings of the IEEE International Conference on Software Architecture Workshops*, pp. 282–285. IEEE, 2017. doi: 10.1109/ICSAW.2017.16

[P20] F. A. Fontana, I. Pigazzini, R. Roveda, and M. Zanoni. Automatic detection of instability architectural smells. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, pp. 433–437. IEEE, 2016. doi: 10.1109/ICSME.2016.33

[P21] F. A. Fontana, R. Roveda, and M. Zanoni. Technical debt indexes provided by tools: A preliminary discussion. In *Proceedings of the IEEE 8th International Workshop on Managing Technical Debt*, pp. 28–31. IEEE, 2016. doi: 10.1109/MTD.2016.11

[P22] F. A. Fontana, R. Roveda, and M. Zanoni. Tool support for evaluating architectural debt of an existing system: An experience report. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pp. 1347–1349. ACM, 2016. doi: 10.1145/2851613.2851963

[P23] F. A. Fontana, R. Roveda, M. Zanoni, C. Raibulet, and R. Capilla. An experience report on detecting and repairing software architecture erosion. In *Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture*, pp. 21–30. IEEE, 2016. doi: 10.1109/WICSA.2016.37

[P24] M. Goldstein and I. Segall. Automatic and continuous software architecture validation. In *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, pp. 59–68. IEEE, 2015. doi: 10.1109/ICSE.2015.135

[P25] T. Hassouna. Detection of web service refactoring opportunities. Master's thesis, University of Michigan-Dearborn, 2017. doi: 2027.42/136611

[P26] S. Hayashi, F. Minami, and M. Saeki. Inference-based detection of architectural violations in MVC2. In *Proceedings of the 12th International Conference on Software Technologies*, pp. 394–401, 2017.

[P27] S. Hayashi, F. Minami, and M. Saeki. Detecting architectural violations using responsibility and dependency constraints of components. *IEICE Transaction on Information and Systems*, 101(7):1780–1789, 2018. doi: 10.1587/transinf.2017KBP0023

[P28] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, and A. Shapochka. A case study in locating the architectural roots of technical debt. In *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, pp. 179–188. IEEE, 2015. doi: 10.1109/ICSE.2015.146

[P29] D. Le, D. Link, Y. Zhao, A. Shahbazian, C. Mattmann, and N. Medvidovic. Toward a classification framework for software architectural smells. *Technical Report*, 2017.

[P30] D. M. Le, D. Link, A. Shahbazian, and N. Medvidovic. An empirical study of architectural decay in open-source software. In *Proceedings of the IEEE International Conference on Software Architecture*, pp. 176–17609. IEEE, 2018. doi: 10.1109/ICSA.2018.00027

[P31] J. Lenhard, M. M. Hassan, M. Blom, and S. Herold. Are code smell detection tools suitable for detecting architecture degradation? In *Proceedings of the 11th European Conference on Software Architecture*, pp. 138–144. ACM, 2017. doi: 10.1145/3129790.3129808

[P32] Z. Li, P. Liang, and P. Avgeriou. Architectural technical debt identification based on architecture decisions and change scenarios. In *Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture*, pp. 65–74. IEEE, 2015. doi: 10.1109/WICSA.2015.19

[P33] Z. Li, P. Liang, P. Avgeriou, N. Guelfi, and A. Ampatzoglou. An empirical investigation of modularity metrics for indicating architectural technical debt. In *Proceedings of the 10th International ACM SIGSOFT Conference on Quality of Software Architectures*, pp. 119–128. ACM, 2014. doi: 10.1145/2602576.2602581

[P34] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. von Staa. On the relevance of code anomalies for identifying architecture degradation symptoms. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*, pp. 277–286. IEEE, 2012. doi: 10.1109/CSMR.2012.35

[P35] I. Macia, A. Garcia, A. von Staa, J. Garcia, and N. Medvidovic. On the impact of aspect-oriented code smells on architecture modularity: An exploratory study. In *Proceedings of the 5th Brazilian Symposium on Software Components, Architectures and Reuse*, pp. 41–50. IEEE, 2011. doi: 10.1109/SBCARS.2011.18

[P36] A. Martini, F. A. Fontana, A. Biaggi, and R. Roveda. Identifying and prioritizing architectural debt through architectural smells: A case study in a large software company. In *Proceedings of the European Conference on Software Architecture*, pp. 320–335. Springer, 2018. doi: 10.1007/978-3-030-00761-4_21

[P37] A. Martini, E. Sikander, and N. Madlani. A semi-automated framework for the identification and estimation of architectural technical debt: A comparative case-study on the modularization of a software component. *Information and Software Technology*, 93:264–279, 2018. doi: 10.1016/j.infsof.2017.08.005

[P38] R. Mo, Y. Cai, R. Kazman, and L. Xiao. Hotspot patterns: The formal definition and automatic detection of architecture smells. In *Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture*, pp. 51–60. IEEE, 2015. doi: 10.1109/WICSA.2015.12

[P39] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng. Architecture anti-patterns: Automatically detectable violations of design principles. *IEEE Transactions on Software Engineering*, 2019. doi: 10.1109/TSE.2019.2910856

[P40] R. Mo, J. Garcia, Y. Cai, and N. Medvidovic. Mapping architectural decay instances to dependency models. In *Proceedings of the 4th International Workshop on Managing Technical Debt*, pp. 39–46. IEEE Press, 2013. doi: 10.1109/MTD.2013.6608677

[P41] R. Mo, W. Snipes, Y. Cai, S. Ramaswamy, R. Kazman, and M. Naedele. Experiences applying automated architecture analysis tool suites. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 779–789. ACM, 2018. doi: 10.1145/3238147.3240467

[P42] N. Moha, F. Palma, M. Nayrolles, B. J. Conseil, Y.-G. Guéhéneuc, B. Baudry, and J.-M. Jézéquel. Specification and detection of SOA antipatterns. In *Proceedings of the International Conference on Service-Oriented Computing*, pp. 1–16. Springer, 2012. doi: 10.1007/978-3-642-34321-6_1

[P43] M. Nayebi, Y. Cai, R. Kazman, G. Ruhe, Q. Feng, C. Carlson, and F. Chew. A longitudinal study of identifying and paying down architecture debt. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, pp. 171–180. IEEE Press, 2019. doi: 10.1109/ICSE-SEIP.2019.00026

[P44] M. Nayrolles, N. Moha, and P. Valtchev. Improving SOA antipatterns detection in service based systems by mining execution traces. In *Proceedings of the 20th Working Conference on Reverse Engineering*, pp. 321–330. IEEE, 2013. doi: 10.1109/WCRE.2013.6671307

[P45] M. Nayrolles, F. Palma, N. Moha, and Y.-G. Guéhéneuc. SODA: A tool support for the detection of SOA antipatterns. In *International Conference on Service-Oriented Computing*, pp. 451–455. Springer, 2012. doi: 10.1007/978-3-642-37804-1_51

[P46] W. N. Oizumi, A. F. Garcia, T. E. Colanzi, M. Ferreira, and A. V. Staa. On the relationship of code-anomaly agglomerations and architectural problems. *Journal of Software Engineering Research and Development*, 3(1):11, 2015. doi: 10.1186/s40411-015-0025-y

[P47] W. N. Oizumi, A. F. Garcia, T. E. Colanzi, M. Ferreira, and A. von Staa. When code-anomaly agglomerations represent architectural problems? An exploratory study. In *Proceedings of the Brazilian Symposium on Software Engineering*, pp. 91–100. IEEE, 2014. doi: 10.1109/SBES.2014.18

[P48] A. Ouni, R. Gaikovina Kula, M. Kessentini, and K. Inoue. Web service antipatterns detection using genetic programming. In *Proceedings of the Annual Conference on Genetic and Evolutionary Computation*, pp. 1351–1358. ACM, 2015. doi: 10.1145/2739480.2754724

[P49] A. Ouni, M. Kessentini, K. Inoue, and M. O. Cinnéide. Search-based web service antipatterns detection. *IEEE Transactions on Services Computing*, 10(4):603–617, 2015. doi: 10.1109/TSC.2015.2502595

[P50] F. Palma. Detection of SOA antipatterns. In *Proceedings of the International Conference on Service-Oriented Computing*, pp. 412–418. Springer, 2012. doi: 10.1007/978-3-642-37804-1_43

[P51] F. Palma, L. An, F. Khomh, N. Moha, and Y.-G. Guéhéneuc. Investigating the change-proneness of service patterns and antipatterns. In *Proceedings of the IEEE 7th International Conference on Service-Oriented Computing and Applications*, pp. 1–8. IEEE, 2014. doi: 10.1109/SOCA.2014.43

[P52] F. Palma, J. Dubois, N. Moha, and Y.-G. Guéhéneuc. Detection of REST patterns and antipatterns: A heuristics-based approach. In *Proceedings of the International Conference on Service-Oriented Computing*, pp. 230–244. Springer, 2014. doi: 10.1007/978-3-662-45391-9_16

[P53] F. Palma, N. Moha, and Y.-G. Guéhéneuc. UniDoSA: The unified specification and detection of service antipatterns. *IEEE Transactions on Software Engineering*, 2018. doi: 10.1109/TSE.2018.2819180

[P54] F. Palma, N. Moha, G. Tremblay, and Y.-G. Guéhéneuc. Specification and detection of SOA antipatterns in web services. In *Proceedings of the European Conference on Software Architecture*, pp. 58–73. Springer, 2014. doi: 10.1007/978-3-319-09970-5_6

[P55] F. Palma, M. Nayrolles, N. Moha, Y.-G. Guéhéneuc, B. Baudry, and J.-M. Jézéquel. SOA antipatterns: An approach for their specification and detection. *International Journal of Cooperative Information Systems*, 22(04):1341004, 2013. doi: 10.1142/S0218843013410049

[P56] I. Pigazzini. Automatic detection of architectural bad smells through semantic representation of code. In *Proceedings of the 13th European Conference on Software Architecture*, vol. 2, pp. 59–62. ACM, 2019. doi: 10.1145/3344948.3344951

[P57] V. Pismag and J. Kelly. Prediction of web service antipatterns using machine learning. Master's thesis, University of Michigan-Dearborn, 2017. doi: 2027.42/136193

[P58] D. Reimanis, C. Izurieta, R. Luhr, L. Xiao, Y. Cai, and G. Rudy. A replication case study to measure the architectural quality of a commercial system. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, p. 31. ACM, 2014. doi: 10.1145/2652524.2652581

[P59] A. Sanchez, L. S. Barbosa, and A. Madeira. Modelling and verifying smell-free architectures with the archery language. In *Proceedings of the International Conference on Software Engineering and Formal Methods*, pp. 147–163. Springer, 2014. doi: 10.1007/978-3-319-15201-1_10

[P60] T. Sharma. How deep is the mud: Fathoming architecture technical debt using designite. In *Proceedings of the IEEE/ACM International Conference on Technical Debt*, pp. 59–60. IEEE, 2019. doi: 10.1109/TechDebt.2019.00018

[P61] V. S. Sharma and S. Anwer. Detecting performance antipatterns before migrating to the cloud. In *Proceedings of the IEEE 5th International Conference on Cloud Computing Technology and Science*, vol. 1, pp. 148–151. IEEE, 2013. doi: 10.1109/CloudCom.2013.166

[P62] M. E. Shin, Y. Xu, F. Paniagua, and J. H. An. Detection of anomalies in software architecture with connectors. *Science of Computer Programming*, 61(1):16–26, 2006. doi: 10.1016/j.scico.2005.11.002

[P63] G. Sierra, A. Tahmid, E. Shihab, and N. Tsantalis. Is self-admitted technical debt a good indicator of architectural divergences? In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Re-engineering*, pp. 534–543. IEEE, 2019. doi: 10.1109/SANER.2019.8667999

[P64] P. Skiada, A. Ampatzoglou, E.-M. Arvanitou, A. Chatzigeorgiou, and I. Stamelos. Exploring the relationship between software modularity and technical debt. In *Proceedings of the 44th EUROMICRO Conference on Software Engineering and Advanced Applications*, pp. 404–407. IEEE, 2018. doi: 10.1109/SEAA.2018.00072

[P65] W. Snipes, S. Karlekar, and R. Mo. A case study of the effects of architecture debt on software evolution effort. In *Proceedings of the 44th EUROMICRO Conference on Software Engineering and Advanced Applications*, pp. 400–403. IEEE, 2018. doi: 10.1109/SEAA.2018.00071

[P66] D. Tamburri, R. Kazman, and W.-J. Van Den Heuvel. Splicing community and software architecture smells in agile teams: An industrial study. In *Proceedings of the 52nd Hawaii International Conference on System*

Sciences, 2019. doi: 10.24251/HICSS.2019.843

[P67] D. Tiwari, H. Washizaki, Y. Fukazawa, T. Fukuoka, J. Tamaki, N. Hosotani, and M. Kohama. Metrics driven architectural analysis using dependency graphs for C language projects. In *Proceedings of the IEEE 43rd Annual Computer Software and Applications Conference*, vol. 1, pp. 117–122. IEEE, 2019. doi: 10.1109/COMPSAC.2019.00025

[P68] A. Tommasel. Applying social network analysis techniques to architectural smell prediction. In *Proceedings of the International Conference on Software Architecture Companion*, pp. 254–261. IEEE, 2019. doi: 10.1109/ICSA-C.2019.00053

[P69] D. Tripathi, U. Suman, M. Ingle, and S. Tanwani. Towards introducing and implementation of SOA design antipatterns. *International Journal of Computer Theory and Engineering*, 6(1):20, 2014. doi: 10.7763/IJCTE.2014.V6.829

[P70] C. Trubiani, A. Bran, A. van Hoorn, A. Avritzer, and H. Knoche. Exploiting load testing and profiling for performance antipattern detection. *Information and Software Technology*, 95:329–345, 2018. doi: 10.1016/j.infsof.2017.11.016

[P71] C. Trubiani and A. Koziolek. Detection and solution of software performance antipatterns in palladio architectural models. *ACM SIGSOFT Software Engineering Notes*, 36(5):36–36, 2011. doi: 10.1145/1958746.1958755

[P72] C. Trubiani, A. Koziolek, V. Cortellessa, and R. Reussner. Guilt-based handling of software performance antipatterns in palladio architectural models. *Journal of Systems and Software*, 95:141–165, 2014. doi: 10.1016/j.jss.2014.03.081

[P73] R. Vanciu and M. Abi-Antoun. Finding architectural flaws using constraints. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pp. 334–344. IEEE, 2013. doi: 10.1109/ASE.2013.6693092

[P74] P. Velasco-Elizondo, L. Castañeda-Calvillo, A. García-Fernandez, and S. Vazquez-Reyes. Towards detecting MVC architectural smells. In *Proceedings of the International Conference on Software Process Improvement*, pp. 251–260. Springer, 2017. doi: 10.1007/978-3-319-69341-5_23

[P75] R. Verdecchia. Identifying architectural technical debt in android applications through automated compliance checking. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, pp. 35–36. ACM, 2018. doi: 10.1145/3197231.3198442

[P76] S. Vidal, E. Guimaraes, W. Oizumi, A. Garcia, A. D. Pace, and C. Marcos. Identifying architectural problems through prioritization of code smells. In *Proceedings of the 10th Brazilian Symposium on Software Components, Architectures and Reuse*, pp. 41–50. IEEE, 2016. doi: 10.1109/SBCARS.2016.11

[P77] S. Vidal, W. Oizumi, A. Garcia, A. D. Pace, and C. Marcos. Ranking architecturally critical agglomerations of code smells. *Science of Computer Programming*, 182:64–85, 2019. doi: 10.1016/j.scico.2019.07.003

[P78] M. von Detten and S. Becker. Combining clustering and pattern detection for the re-engineering of component-based software systems. In *Proceedings of the Joint ACM SIGSOFT conference–QoSA and ACM SIGSOFT symposium–ISARCS on Quality of Software Architectures–QoSA and Architecting Critical Systems–ISARCS*, pp. 23–32. ACM, 2011. doi: 10.1145/2000259.2000265

[P79] A. von Zitzewitz. Mitigating technical and architectural debt with sonargraph: Using static analysis to enforce architectural constraints. In *Proceedings of the 2nd International Conference on Technical Debt*, pp. 66–67. IEEE Press, 2019. doi: 10.1109/TechDebt.2019.00022

[P80] H. Wang, M. Kessentini, T. Hassouna, and A. Ouni. On the value of quality of service attributes for detecting bad design practices. In *Proceedings of the IEEE International Conference on Web Services*, pp. 341–348. IEEE, 2017. doi: 10.1109/ICWS.2017.126

[P81] H. Wang, A. Ouni, M. Kessentini, B. Maxim, and W. I. Grosky. Identification of web service refactoring opportunities as a multi-objective problem. In *Proceedings of the IEEE International Conference on Web Services*, pp. 586–593. IEEE, 2016. doi: 10.1109/ICWS.2016.81

[P82] L. Xiao. Quantifying architectural debts. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pp. 1030–1033. ACM, New York, NY, USA, 2015. doi: 10.1145/2786805.2803194

[P83] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng. Identifying and quantifying architectural debt. In *Proceedings of the 38th IEEE/ACM International Conference on Software Engineering*, pp. 488–498. ACM, 2016. doi: 10.1145/2884781.2884822

[P84] A. Yugov. Approach to anti-pattern detection in service-oriented software systems. *In Proceedings of the Institute for System Programming*, 28(2), 2016. doi: 10.15514/ISPRAS-2016-28(2)-5

[P85] L. Zhang, Y. Sun, H. Song, W. Wang, and G. Huang. Detecting antipatterns in java EE runtime system model. In *Proceedings of the 4th Asia-Pacific Symposium on Internetware*, p. 21. ACM, 2012. doi: 10.1145/2430475.2430496

A Description of Architectural Smells

Architectural smell	Description
Abstraction without decoupling	This smell occurs where a client class uses a service represented as an abstract type, but also a concrete implementation of this service, represented as a non-abstract subtype of the abstract type [P3].
Ambiguous interface	This smell occurs when an abstraction (interface) is over-engineered by adding methods intended to accommodate potential future requirements but never used [17].
Ambiguous name	This smell occurs when developers use ambiguous or meaningless names for interfaces [13].
Anchor submission	This smell occurs when each file structurally depends on the anchor file, but each member historically dominates the anchor [P83].
Anchor dominant	This smell occurs when each file structurally depends on the anchor file, and the anchor file historically dominates each member file [P83].
API versioning	This smell occurs when APIs are not semantically versioned [47].
Architecture violation	This smell occurs when an intended architecture is different from its actual implementation [P3].
Big bang	This smell occurs when an entire system is built at once [28].
Bottleneck service	This smell occurs when a service is highly used (high incoming and outgoing coupling) by other services [13].
Bloated service	This smell occurs when a service becomes a blob with one large interface and/or lots of parameters [P53].
Blob or God object/component	This smell occurs when a component implements an excessive number of concerns [P3].
Brain controller	This smell occurs when controllers have too much flow control [5].
Brain repository	This smell occurs when a complex logic is developed in the repository [5].
Circuitous treasure hunt	This smell occurs when an object looks in several places to find the information that it needs [P11].
Chatty service	This smell occurs when a service has a high number of connections with other services [13].
Clique	This smell occurs when a group of files are tightly coupled by dependency cycles [P65].
Co-change coupling	This smell occurs when changes to a component require changes in another component [P30].
Concern overload	This smell occurs when a component implements an excessive number of concerns [P3].
Connector envy	This smell occurs when components cover too much functionality with respect to connections [17].
Crudy interface	This smell occurs when services show an RPC-like behavior by declaring CRUD-type operations [13].
Crudy URI	This smell occurs when crudy verbs (e.g., create, read, update, or delete) are used in the APIs [P53].
Cyclic dependency	This smell occurs when two or more architecture components depend on each other directly or indirectly [P3].
Cyclic hierarchy	This smell occurs when a direct referencing of a subtype from a supertype is created [P3].
Cycles between namespaces	This smell occurs when two or more namespaces depend on each other directly or indirectly [P3].
Data service	This smell occurs when a service has only accessor operations (getters and setters) [13].
Degenerated inheritance	This smell occurs when there are multiple inheritance paths connecting subtypes with their supertypes or a concrete class with their abstractions (abstract classes or interfaces) [P3].
Dense structure	This smell occurs when an abstraction or a concrete class has (outgoing and ingoing) dependencies with a large number of other abstractions or concrete classes [P3].
Duplicated service	This smell occurs when a set of highly similar services exists [13].
Empty semi-trucks	This smell occurs when an excessive number of requests is required to perform a task [P11].
ESB usage	This smell occurs when micro-services communicate via an ESB (enterprise service bus)—it adds complexities for registering and de-registering services on it [47].
Excessive dynamic allocation	This smell occurs when an application unnecessarily creates and destroys large numbers of objects during its execution [P11].
Extensive processing	This smell occurs when extensive processing impedes overall response time [P11].
Fat repository	This smell occurs when a repository is managing too many entities [5].
Feature concentration	This smell occurs when different functionalities are implemented in a single design construct [17].
Forgetting hypermedia	This smell occurs when there is a lack of hypermedia (i.e., not linking resources) [P53].
Golden hammer	This smell occurs when familiar technologies are used as solutions to every problem [47].
Hard-coded endpoints	This smell occurs when micro-services are connected with hard-coded endpoints, making the change in their locations problematic [47].
Hub-like dependency	This smell occurs when an abstraction or a concrete class has (outgoing and ingoing) dependencies with a large number of other abstractions or concrete classes [P3].
Ignoring MIME types	This smell occurs when resources do not support multiple formats (e.g., XML, JSON, etc.) [P53].
Ignoring Caching	This smell occurs when developers avoid to implement the caching capability in the web applications [P52].
Implicit cross-module dependency	This smell occurs when two or more architecture components depend on each other directly or indirectly [P3].
Improper inheritance	This smell occurs when a parent class depends on its derived class or where a client depends on both the parent and derived classes [P7].
Incomplete service	This smell occurs when the client is given the responsibility to complete the service [47].
Incomplete abstraction	This smell occurs when an abstraction does not support interrelated methods completely [46].

Continued on next page

Continued from previous page

Architectural smell	Description
Interface violation	This smell occurs when components in an architecture communicate without their interfaces [P78].
Knot service	This smell occurs when a set of very low cohesive services are tightly coupled [13].
Laborious repository method	This smell occurs when a repository method has multiple database actions [5].
Leaky encapsulation	This smell occurs when a class leaks implementation details because of its public implementation [46].
Link overload	This smell occurs when an abstraction or a concrete class has (outgoing and ingoing) dependencies with a large number of other abstractions or concrete classes [P3].
Low cohesive operations	This smell occurs when developers place very low cohesive operations (not semantically related) in a single portType [47].
Maybe its not RPC	This smell occurs when a service mainly provides CRUD-type (create, read, update, and delete) operations [13].
Meddling service	This smell occurs when services directly query the database [5].
Micro-service greedy	This smell generates an explosion of the number of micro-services composing a system [47].
Missing abstraction	This smell occurs when clumps of data are used instead of creating classes or interfaces [46].
Missing encapsulation	This smell occurs when classes are not encapsulated [46].
Misplaced component	This smell occurs when an architecture component is placed somewhere else other than the one it was intended for, resulting in undesired dependencies [P17].
More is less	This smell occurs when a system spends more time thrashing than accomplishing real work because there are too many processes relative to available resources [P11].
Modularity violation	This smell occurs when an architecture violates the modularity principles [P83].
Multi-service	This smell occurs when a service implements a multitude of methods related to different abstractions [13].
Multipath hierarchy	This smell occurs when there are multiple inheritance paths connecting subtypes with their supertypes or a concrete class with their abstractions [P3].
Not having an API gateway	This smell occurs when service-consumers communicate directly with each micro-service [47].
Nobody home	This smell occurs when a service is defined but never used [13].
Non-transfer communication	This smell occurs when communication between components is not accomplished using transfer objects [P83].
Nothing new	This smell occurs when inappropriate practices in object-oriented practices are attempted to apply in service-oriented [28].
No legacy	This smell occurs when a service provides limited standardized support of data types and interactions [47].
No subsystems	This smell occurs when a system has no subsystems [31].
One-lane bridge	This smell occurs when only one or a few processes can be executed concurrently [P11].
Overgeneralized subsystems	This smell occurs when the generalization of the subsystems is overdone [31].
Overstandardized SOA	This smell occurs when all aspects and dimensions of SOA are overstandardized [28].
Package cycle	This smell occurs when two or more packages depend on each other directly or indirectly [P3].
Package instability	This smell occurs when a package has many dependencies that frequently changes with other packages [P38].
Package abstractness	This smell occurs when a package has unnecessary or missing abstraction [46].
Pipe and filter	This smell occurs when the slowest filter in the architecture results in low throughput [P11].
Promiscuous controller	This smell occurs when controllers are offering too many actions [5].
Redundant portTypes	This smell occurs when multiple portTypes are duplicated with a similar set of operations [13].
Sand pile	This smell occurs when a service is composed of multiple smaller services sharing common data [13].
Scattered functionality	This smell occurs when a high-level concern is realized across multiple components [17].
Security flaws	This smell occurs when critical information is disclosed or tampered, when confidentiality and integrity are not ensured in the architecture [P73].
Separation of concerns	This smell occurs when the responsibilities of the components of an architecture are not appropriately separated [P27].
Service Chain	This smell occurs when consecutive service invocations happen [13].
Shared libraries	This smell occurs when shared libraries between different micro-services are used [47].
Shared persistency	This smell occurs when different micro-services access the same relational database, reducing the service independence [47].
Shiny nickel	This smell occurs due to inflexibility to incorporate new technologies within service architecture [47].
Silver bullet	This smell occurs when unknown technologies are implemented where they are not required [47].
Sloppy delegation	This smell occurs when a component delegates the functionality to other components, which should be performed internally by that component [P30].
Speculative hierarchy	This smell occurs when a hierarchy is created speculatively [46].
Subtype knowledge	This smell occurs when a direct referencing of a subtype from a supertype is created [P3].
Tiny/nano/fine-grained service	This smell occurs when a service has only a few operations [13].
Ramp	This smell occurs when processing time increases as the system is used [P11].

Continued on next page

Continued from previous page

Architectural smell	Description
Too many standards	This smell occurs when different development languages, protocols, frameworks are used in micro-services [47].
Too small package	This smell occurs when a package has only one or two classes [31].
Too many subsystems	This smell occurs when a system consists of many subsystems [31].
Tower of babel	This smell occurs when processes excessively convert, parse, and translate internal data into a common exchange format [P11].
Traffic jam	This smell occurs when one problem causes a backlog of jobs [P11].
Unbalanced processing	This smell occurs when processing cannot make use of available processors [P11].
Unauthorized dependency	This smell occurs when an unauthorized dependency exists between the components [P27].
Unstable dependency	This smell occurs when a component depends on other components that are less stable than itself [P3].
Unused package	This smell occurs when a package is no longer in use [31].
Unclear package name	This smell occurs when developers use ambiguous or meaningless names for packages [31].
Unbalanced package hierarchy	This smell occurs when the package structure is unbalanced [31].
Unauthorized call	This smell occurs when a calling component is not connected to the called component [P83].
Undercover transfer object	This smell occurs when transfer objects serve as data containers for the communication between components [P78].
Unhealthy inheritance hierarchy	This smell occurs when a direct referencing of a subtype from a supertype is created [P3].
Unstable interface	This smell occurs when an interface depends on other interfaces that are less stable than itself [P3].
Unused interface	This smell occurs when an abstraction (interface) is over-engineered by adding methods intended to accommodate potential future requirements but never used [17].
Unutilized abstraction	This smell occurs when a direct referencing of a concrete class is created, instead of referencing one of its supertypes, from an abstract class [P3].
Unnecessary hierarchy	This smell occurs when the inheritance hierarchy is unnecessarily created [46].
Wrong cuts	This smell occurs when micro-services are split based on technical layers instead of business capabilities [47].