IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received August 18, 2020, accepted September 14, 2020, date of publication September 18, 2020, date of current version October 1, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3024671

Software Architecture Degradation in Open Source Software: A Systematic Literature Review

AHMED BAABAD[1,2], HAZURA BINTI ZULZALIL[1], SA'ADAH HASSAN[1], AND SALMI BINTI BAHAROM[1]

[1]Department of Software Engineering and Information System, Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, Serdang 43400, Malaysia

[2]Department of Management Information Systems, Administrative Sciences, Hadhramout University, Mukalla, Yemen

Corresponding author: Hazura Binti Zulzalil (hazura@upm.edu.my)

This work was supported in part by the Universiti Putra Malaysia (UPM).

ABSTRACT Software architecture (SA) has a prominent role in all stages of system development. Given the persistent evolution of software systems over time, SA tends to be eroded or degraded. Such phenomenon is called architectural degradation. In light of this phenomenon, the current study focuses on problems of architectural erosion in the open-source software (OSS). There has been a significant research activity on the OSS over the last decade. Nonetheless, the architectural degradation problems in the OSS are still scattered and disorganized. In addition, there has been no systematic attempt made on existing studies to provide evidence, insight and better understanding for researchers and practitioners. The main objective of the present study is to provide a profound understanding and to review the existing studies on the architectural erosion of the OSS. In this study, we conduct a systematic literature review (SLR) to gather, organize, classify, and analyze the architectural degradation of previous papers published until the year 2020. The data for this study were collected from eight major online databases (ACM, Springer, ScienceDirect, Taylor, IEEE Explorer, Scopus, Web of Science, and Wiley). A total of 74 primary studies were identified as the final samples of this research. The results indicated that rapid software evolution, frequent changes, and the lack of developers' awareness are the most common causes occurred in architecture degradation. Meanwhile, the prominent key indicators of architectural erosion symptoms are code smells and architectural smells. Additionally, the results indicated the most commonly used of the proposed solution for addressing architectural erosion is the metrics-based strategy. Acknowledging the limitations of the current study, more studies are needed that focus on determining other causes that are still ambiguous and improving the other solutions to provide better results in the precision and effectiveness of addressing architectural erosion.

INDEX TERMS Software architecture, architectural degradation, architectural erosion, open-source, OSS, systematic literature review.

I. INTRODUCTION

Modern societies have considerably been relying on large-scale systems, which have a huge effect on our daily living, education, finance, healthcare, communication, transportation, entertainment, commerce, security, and defense [1]. Nonetheless, there is an increasing fragility in these systems, leading to cascading failures because of the nature of operating interdependent connected ecosystems [2]. Accordingly, a NATO workshop had been conducted to identify the software crisis as early as in 1968 [3]. Consequently, the software architecture (SA) domain has been broadly adopted, however, with only mere attention [4] as a significant subfield of software engineering from the past decade, particularly in research and industry [5] as well as software development [6].

SA is considered the basic structure block for establishing any system, as it is the crucial and important factor in defining, succeeding, and developing systems design [7] as well as quality criteria [8], which makes it one of the most fundamental issues in designing and developing software today.

Furthermore, SA has a prominent role in each stage of the system development stages from understanding, analyzing, building, managing, reusing, and requirements to its

The associate editor coordinating the review of this manuscript and approving it for publication was Mario Luca Bernardi.

deployment and maintenance [9], [10]. SA deals with the framework and interactions of a system. Interestingly, the utmost fundamental construction blocks of the structure of SA are components and the interconnection among the components.

Given the continuous development of software systems over time, SA tends to be eroded or degraded as a result of the requirement change, new features [11], corrections or architectural design change decisions, leading to a considerable systems diverge between the implemented architecture and intended architecture [12]–[14]. The phenomenon of architecture degradation usually takes place when the concrete architecture of a software system deviates from its conceptual architecture. Hence, the sustainability of software architecture will be threatened by the architectural degradation that represents the erosion and drift phenomena [15].

Typically, architectural erosion appears when the design structure of architecture is not properly identical with the source code in terms of the predefined mapping and accomplishing by the architecture engineer. The architectural drift usually occurs when the design decisions of the system are not included in the intended architecture [15]. Both phenomena are generated by several problems such as the undocumented, unforeseen, unplanned, random, scattered, and confused architectural design decisions. Furthermore, the unintentional addition, elimination, and modification occur frequently. These problems will affect even more, specifically in the maintenance and development systems life cycle [16]. It is worth noting that SA is the highest level of abstraction and basic reasoning in taking the consideration for producing software, whether this software is an open source or closed source (proprietary source). Accordingly, this study focuses on the problems of eroded architecture in the open source software (OSS).

Over the last decade, research on the OSS has acquired considerable activity, as the commercial use of the OSS components keeps extending [17]. Besides, the OSS has become one of the most debatable themes among users and practitioners. Therefore, several studies have focused on evolutional aspects of the OSS development in response to long-ranged viability and sustainability concerns of software projects based on community [18].

A number of studies have presented the software architectural structure degradation and its deviation from its initial planned (intended) architecture in the OSS. These studies have investigated several possible causes of the architectural degradation occurrence of the intended architecture and the symptoms that introduce a share in the degradation of the architectural design within the OSS environment. Additionally, SA degradation reflects negatively on software quality, leading to the collapse of the entire architecture or a redesign of the system from scratch. Several studies have also addressed the evaluation of degradation by conducting many experimental studies on the OSS to identity, avoid, minimize or repair architectural degradation. Many studies have suggested the use of some tools, models or measures, which contribute to identifying the architectural degradation and divergence that deviate from the intended architecture and understanding of erosion in its first stages. Such suggestions were made in order to preserve the rest of the architecture and proper redirection towards the stability and constancy for architecture. Conclusively, these efforts yielded a set of abundant results in research, which refer to the need for more recent comprehensive overviews and literature reviews that outline and build upon past findings; a set of current knowledge and future directions for researchers and practitioners in the field.

Nevertheless, as mentioned earlier, the concept of architectural degradation in the OSS is still scattered and disorganized. To our best knowledge, no efforts have systematically been made to analyze, summarize, arrange, and structure the existing studies to further provide evidence and better comprehension for researchers and practitioners. However, it is significant to indicate that there are some systematic literature reviews (SLRs) that may be related to the area of bad smells. Sabir *et al.* [19] concentrated on smell's growth, modern approaches, and research trends in object-oriented (OO) and service-oriented systems (SO). Rattan *et al.* [20] focused on code clone and software clone detection through methods and tools in the OO. Zhang *et al.* [21] conducted an identification aim what currently known is for code bad smells. In this study, we further provide a detailed analysis of architectural decay reasons and key symptom indicators of overall architectural erosion, covering symptoms other than bad smells in open-source software. In addition, we investigate the solutions that address architectural degradation and how effective the superiority of solutions is to provide better results through several different aspects. Therefore, we conducted a systematic study with the aim of gathering, classifying, investigating, summarizing, and synthesizing information about the precision and significance of the previous papers published until 2020. The study aims to provide a comprehensive report on the actual contribution of the empirical and thorough results of the current study including studies on this topic.

Generally, the current study presents three-fold contributions to the domain. Firstly, we identified 74 primary studies that detect architectural degradation within the OSS domains, which can be used as a beginning point to widen the knowledge on the topic. Secondly, we conducted wide explanations and profound understanding to provide the knowledge about: (i) potential causes, (ii) symptoms of degradation, (iii) proposed solutions to reduce the degradation, and (iv) evaluation of the effectiveness of the solutions. Thirdly, we identified the list of existing research in architectural erosion within the OOS to understand the current research trend based on our findings, to support further exploration in this domain.

The descriptions that follow present the structure of the rest of the study. A background of the software architecture, software architecture degradation and open-source software are presented in Section 2. The research methodology applied is discussed in Section 3. The results of the study are presented in Section 4. The discussion of the key findings is explained in

Section 5. The threats to the validity of this study are clarified in Section 6. The conclusion is outlined in Section 7.

II. BACKGROUND

This section presents a concise overview and the definition of the SA term along with the software architecture degradation and the OSS to sum up the basic definitions. The details are clarified in the subordinate subsections.

A. SOFTWARE ARCHITECTURE

Over the recent decades of the last century, SA has emerged as the initial comprehension of the large-scope structures of software systems. SA is a collection of the primary design decision made throughout the period of development. Architecture is regarded as a core of software engineering that most accurately specifies the heart of software systems design and development [9]. Accomplishing non-functional and functional requirements is one of the most widely provided parts by SA since it is an integral part of the life-cycle of software evolution [22].

There are several SA definitions, but most terms and common definitions revolve around the concepts of components, modules, and interconnection among the components [23]. For instance, Bass *et al.* [24] characterized SA as the framework of the system that consists of entities, properties, and relationships that are externally visible among them. This definition remarkably indicates that the internal characteristics of the system structure for each entity have no prominent role in SA [23]. On the other hand, Perry and Wolf [15] defined SA through the characterizing of three parts of architectural aspects: processing aspect, data aspect, and connection aspect. The processing aspect is responsible for the data aspect to process the information by connection aspect, which connects the system parts to each other. In a different light, Crispen and Stuckey [25] described SA as distribution and planning strategies. The distribution strategy describes the systems partition into components or (composed of components). The planning strategy describes the interface of the system components with each other.

B. SOFTWARE ARCHITECTURE DEGRADATION

SA degrades as the system evolves [26]. Interestingly, the eroded architecture impulses the system to complexity, difficulty, and frequent changes than before [27]. The repeated architectural decay leads to a shortening lifetime of the system or rather affects the architecture entirely, resulting in redesigning the architecture of the system from scratch [28]. As a consequence, the SA of system erode over time. This phenomenon is known as architectural degeneration [23], [29], design erosion [30], architectural erosion [11], [23], [31], drift [15], [23], [32], mismatch [33], architectural decay [34], design decay [35], code decay [12], [27], software entropy [36], architecture erosion (or decay) [37], software architecture degradation [9], [15], [38] or software aging [11].

Although there are many definitions for SA degradation, most definitions and concepts revolve around the implemented architecture and the intended architecture. For example, Taylor *et al.* [9] as well as Perry and Wolf [15] described the architectural degradation as a process of the persistent inconsistency between the descriptive software architecture as implemented and the prescriptive software architecture as intended. Gurp and Bosch [30], Lindvall and Muthig [39], as well as Dong and Godfrey [40] described the architectural degradation as a gradual gap usually detected between the actual and planned architecture software, which is implemented by its source code. Macia *et al.* [41], Bertran *et al.* [42], and Macia *et al.* [43] described the architecture degradation as a direct outcome of the gradual injection of code anomalies in the low-level of the systems as the software evolves. Accordingly, the architectural degradation occurs within the systems when implemented software architecture deviates that represents the source code or low-level far from the intended software architecture that represents the high level or the conceptual model of architecture design.

C. OPEN SOURCE SOFTWARE

The term open-source software (OSS) points out something the community can change and participate in because its design is available to everyone.

With the increased interest in the OSS, several researchers and practitioners targeted to explore and study the evolution and design of the OSS and identify some risks such as sustainability [44], making the OSS of significant interest today as a viable alternative to the closed-source development [45]. The OSS is based on a methodology in establishing its projects, which completely differs from the used method in commercial systems [46] such as the source code available to the public, the price associated with the value of the system and modification of the software to individual needs. The OSS systems studies are classified into three categories according to the success factors of OSS: The first category explores the successful OSS projects Apache [47], FreeBSD [48], OpenBSD [49] and Debian GNU/Linux [50]. The second category addresses the similarities that contribute to the composition of the process used in successful OSS Apache, Mozilla [51], fifteen OSS Projects [52] as well as Arla and Mozilla Projects [53]. The third category focuses on the general public aspect of OSS projects [54], [55]. The success of OSS projects has resulted in the reasoning stabilization of many researchers and experts that OSS may extremely contribute to resolving software crises, which in turn, some advocates believe that future software will either be the OSS or not at all [56].

III. RESEARCH METHODOLOGY

The architectural degradation needs a full detail of the selected studies to understand how architectural degradation occurs within OSS projects. Therefore, it was necessary to conduct an organized systematic study that depends on a specific protocol and follows the standards and guidelines by

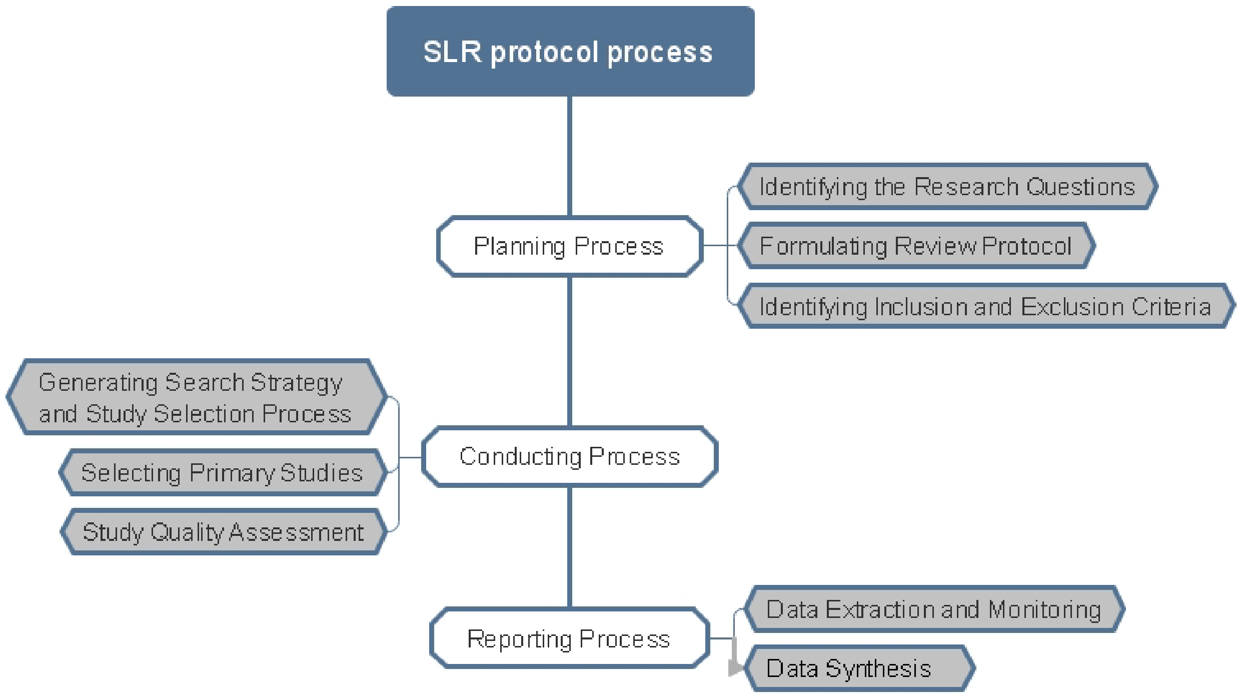

SLR protocol process
Planning Process
Identifying the Research Questions
Formulating Review Protocol
Identifying Inclusion and Exclusion Criteria
Conducting Process
Generating Search Strategy and Study Selection Process
Selecting Primary Studies
Study Quality Assessment
Reporting Process
Data Extraction and Monitoring
Data Synthesis

FIGURE 1. SLR Protocol processes and activities.

Kitchenham and Charters [57], gather all running evidence for the research question, and provide the protocol of instructions based on the evidence for practitioners [58].

Such a process enables the method to identify and aggregate the sources of primary papers, include and exclude the papers in consonance with the previously identified criteria, investigate the data and synthesize the papers in a systematic manner. Accordingly, four main questions were formulated and defined to achieve the purpose of the current study. These questions specified the planning of search strategies, which in turn, lead to the extraction of the data that answered the research questions of the current study.

The SLR protocol as used by Kitchenham and Charters [57] aims at conducting a comprehensive study and examining the profound detail in a specific part of a topic to identify the gaps and future trends in the current research, contributes to introducing a deep view that helps researchers to fill the gaps.

The SLR process consists of three main parts that are considered as indispensable principles in conducting a reliable research process: (1) planning, (2) conducting, and (3) documentation (reporting). Each part has other activities that settle down within parts that are outlined with its activities (as shown in Fig. 1). The sections that follow explain the steps in the SLR protocol.

A. PLANNING PROCESS

The most significant part of the planning process is defining the research question(s), formulating the review protocol, and identifying the inclusion and exclusion criteria that should be subject to a particular SLR Protocol. The following sub-sections describe the SLR planning process.

1) RESEARCH QUESTIONS

Essentially, the identification and construction of the research question are one of the most important steps, on which the SLR protocol processes are built. It is considered the basic idea, in which the researcher intensely realized the need to identify the details of the previously published studies. Concretely, this research question consists of four questions, in relation to the motivation for each question as demonstrated in Table 1.

2) DEVELOPING A REVIEW PROTOCOL

To conduct a review systematically, Kitchenham and Charters [57] as well as Kitchenham [65] specified a protocol method to reduce the probability of bias in the research. Without a protocol, the selection of the sample studies would be individually adapted by the researcher's

expectations. This would result in missing studies that should have been included among the sample studies that are needed in order to provide a profound investigation and broad understanding of the phenomenon.

The stages of a protocol review are clarified by the following processes: (i) identifying research questions, (ii) generating search strategy (iii) studying selection criteria and procedures, (iv) studying quality assessment procedures, (v) implementing data extraction strategy, and

TABLE 1. Research questions.

No	Research question	Motivation
RQ1	What are the possible reasons for the occurrence of architectural degradation?	This question triggers an exploration of the causes that assist in architectural degradation emergence within OSS projects.
RQ2	What are the key indicators that have a prominent role in the permanence of architectural degradation symptoms?	This question triggers the identification of the main indicators that contribute to negatively increasing the architectural degradation symptoms, which lead to generating problems such as deteriorating software performance [59, 60], increasing software maintainability costs [61, 62], and software architecture quality degradation [63, 64].
RQ3	Are there proposed solutions that contribute to identifying/addressing/minimizing/avoiding/predicting architectural degradation?	This question triggers the identification of the offered solutions and its contribution in addressing architectural degradation by contrast symptoms of the intended indicators.
RQ4	How effective are the utilization of the proposed solutions in the studies?	This question triggers to identify the effectiveness of solutions provided by the suggested studies and also determine the extent of the impact in addressing architectural decay in a negative or positive manner.

(vi) synthesizing of the extracted data. The stages of protocol review for this research are illustrated in Fig. 2.

3) INCLUSION AND EXCLUSION CRITERIA

Performing inclusion and exclusion criteria in the research process for the selected published studies constantly aims at ensuring that the primary chosen studies are reasonably related to the research questions and suitable in response to the defined questions in the SLR. Throughout the search process, several research articles were found (journals, books, chapters of books, conferences, workshops, symposiums and other research articles). Fig. 3 shows the process to conduct the initial comprehensive search process in line with the key search string in the given resources.

We selected only research articles written in the English language while research articles written in languages other than English were excluded. Another selection criterion of the articles was that the articles are related to the identification of the topic of architectural decay in the OSS community that has a direct or indirect association to answer the respective research question of the study. On the other hand, articles that were irrelevant and not clear in addressing the research questions of the study were excluded. In addition, articles which are less than four pages were also excluded because of its insufficiency to introduce a profound understanding and evaluation of the specific topic. Concerning the timing of the SLR procedure in our study, the beginning date of the search for the published articles was not specified since no efforts have been made for architectural degradation in the OSS, only the final time was specified, which was March 31, 2020. This date was the closing up for the search process and the starting time point for the data synthesis process.

Table 2 summarizes the inclusion and exclusion of the selected paper criteria in the study.

TABLE 2. Inclusion and exclusion criteria.

Inclusion	Exclusion
The article is written in English	The article is presented in languages other than English.
The article must have relevance to architectural decay in OSS directly.	The study in the article does not clearly address architectural decay in OSS.
The article has an association with the defined research questions.	The article does not correlate with the defined research questions.
The article is more than or equal to 4 pages (≥4).	The article is less than 4 pages (<4).

B. CONDUCTING PROCESS

Once the protocol has been defined in the planning process, the conducting process starts with the review proper, which involves identification of the search strategy, study selection process and selection of the primary studies as well as the quality assessment. The sub-sections that follow explain the SLR conducting process.

1) SEARCH STRATEGY

Defining an accurate and comprehensive search strategy will provide satisfactory results with a wide coverage of published studies regarding the topic, which is identified by the researchers. Kitchenham and Charters [57] presented the search mechanism that identifies the following search strategy for primary studies including search string identification and resources to be searched. Fig. 2 demonstrates the search strategy mechanism, including manual and automatic searches. The section that follows clarifies the search strategy mechanism.

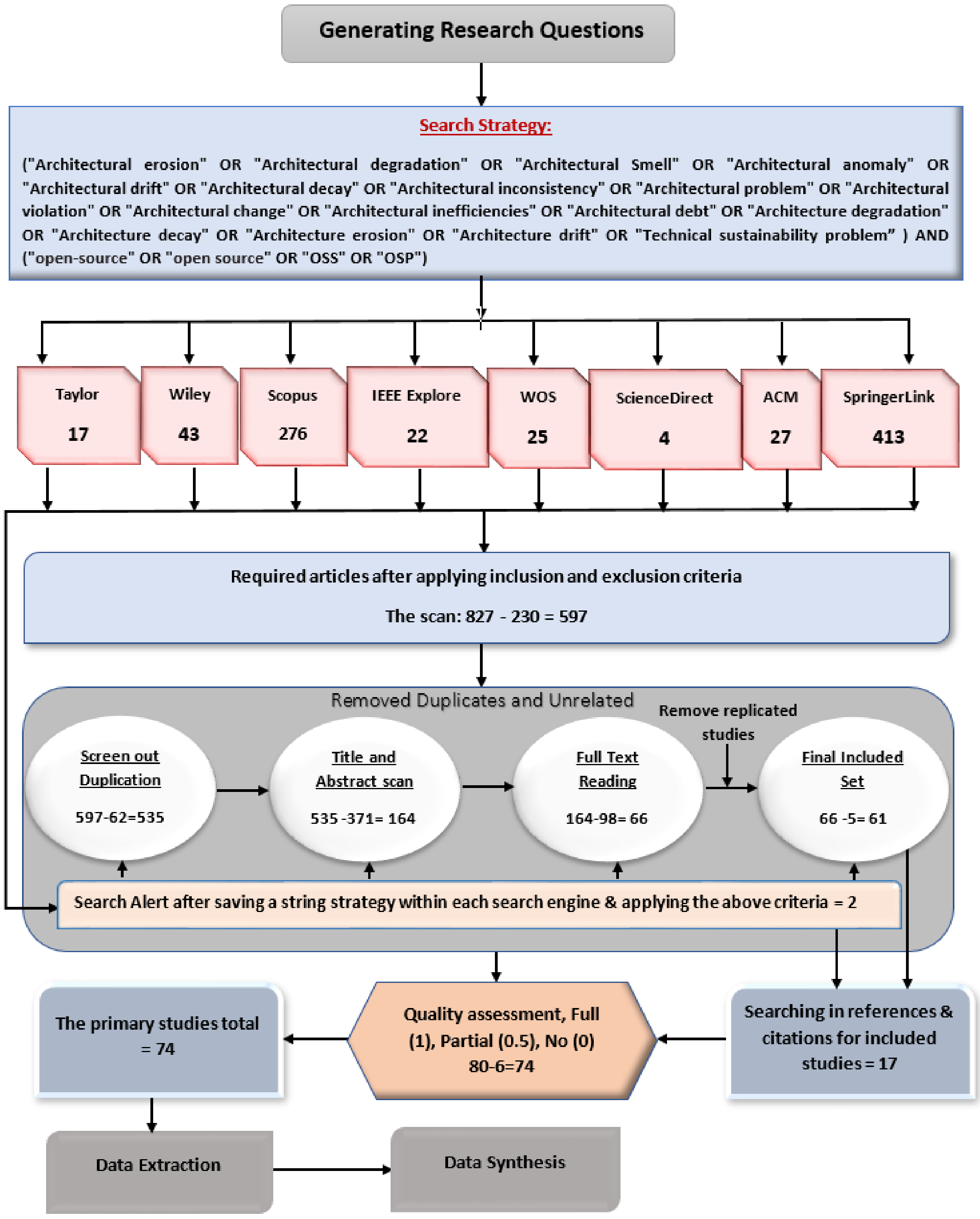
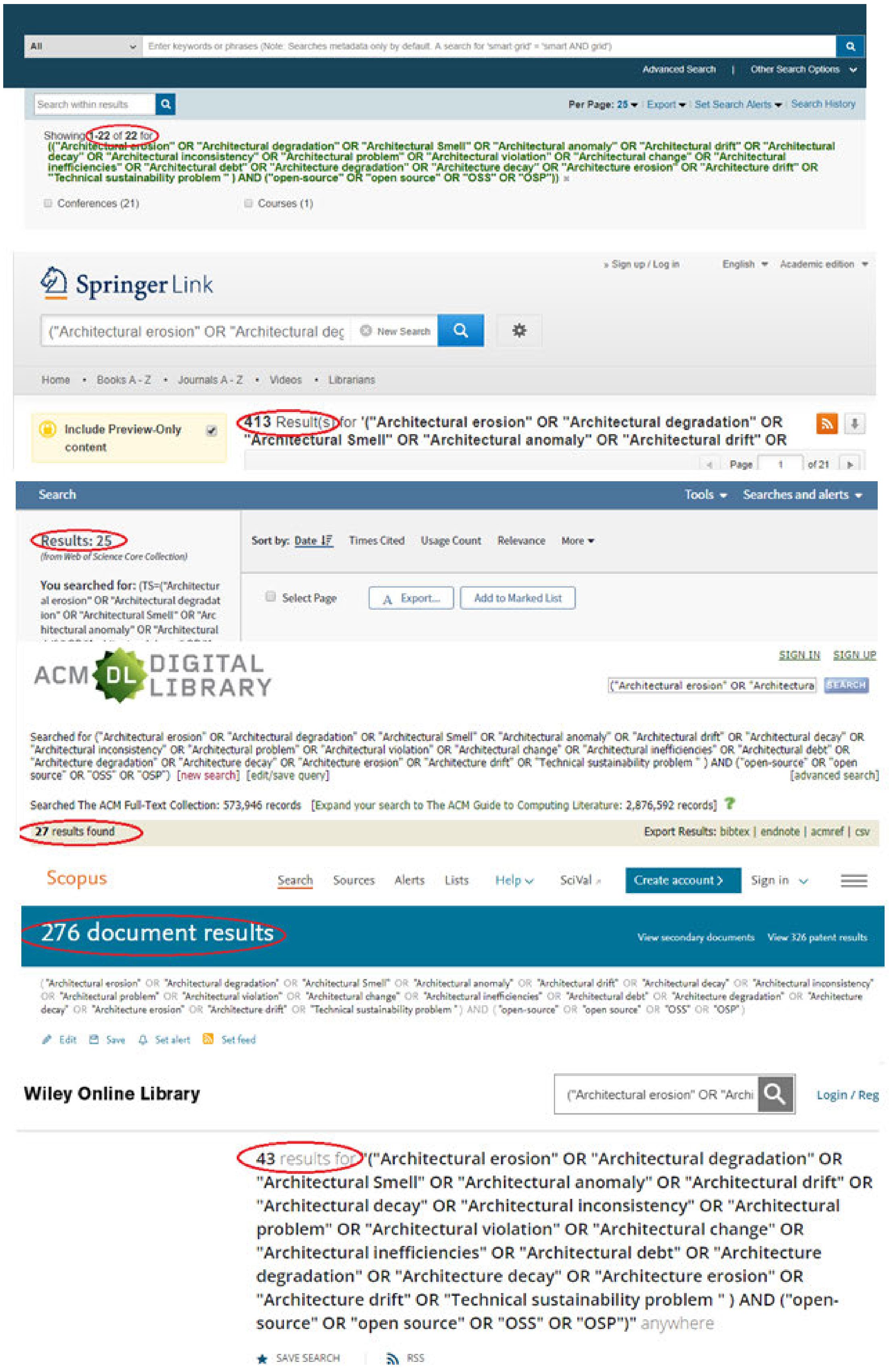

Generating Research Questions
Search Strategy:
("Architectural erosion" OR "Architectural degradation" OR "Architectural Smell" OR "Architectural anomaly" OR "Architectural drift" OR "Architectural decay" OR "Architectural inconsistency" OR "Architectural problem" OR "Architectural violation" OR "Architectural change" OR "Architectural inefficiencies" OR "Architectural debt" OR "Architecture degradation" OR "Architecture decay" OR "Architecture erosion" OR "Architecture drift" OR "Technical sustainability problem") AND ("open-source" OR "open source" OR "OSS" OR "OSP")
Taylor 17
Wiley 43
Scopus 276
IEEE Explore 22
WOS 25
ScienceDirect 4
ACM 27
SpringerLink 413
Required articles after applying inclusion and exclusion criteria
The scan: 827 - 230 = 597
Removed Duplicates and Unrelated
Screen out Duplication 597-62=535
Title and Abstract scan 535 -371= 164
Full Text Reading 164-98= 66
Remove replicated studies
Final Included Set 66 -5= 61
Search Alert after saving a string strategy within each search engine & applying the above criteria = 2
The primary studies total = 74
Quality assessment, Full (1), Partial (0.5), No (0) 80-6=74
Searching in references & citations for included studies = 17
Data Extraction
Data Synthesis

FIGURE 2. The stages of Review Protocol.

a: IDENTIFICATION OF SEARCH STRING

To compose relevant search string, we follow the guidelines stated by Kitchenham and Charters [57] and the procedures by Kitchenham [65], which are: a) getting the derivation of key terms from the research questions; b) looking for all the key terms in synonyms, alternative spellings, and


All
Enter keywords or phrases (Note: Searches metadata only by default. A search for 'smart grid' = 'smart AND grid')
Advanced Search | Other Search Options
Search within results
Per Page: 25 | Export | Set Search Alerts | Search History

Showing 1-22 of 22 for
(("Architectural erosion" OR "Architectural degradation" OR "Architectural Smell" OR "Architectural anomaly" OR "Architectural drift" OR "Architectural decay" OR "Architectural inconsistency" OR "Architectural problem" OR "Architectural violation" OR "Architectural change" OR "Architectural inefficiencies" OR "Architectural debt" OR "Architecture degradation" OR "Architecture decay" OR "Architecture erosion" OR "Architecture drift" OR "Technical sustainability problem ") AND ("open-source" OR "open source" OR "OSS" OR "OSP"))

Conferences (21) Courses (1)

Springer Link
» Sign up / Log in
English
Academic edition
("Architectural erosion" OR "Architectural deg
New Search
Home • Books A - Z • Journals A - Z • Videos • Librarians
Include Preview-Only content

413 Result(s) for '("Architectural erosion" OR "Architectural degradation" OR "Architectural Smell" OR "Architectural anomaly" OR "Architectural drift" OR

Page 1 of 21

Results: 25
(from Web of Science Core Collection)

You searched for: (TS=("Architectural erosion" OR "Architectural degradation" OR "Architectural Smell" OR "Architectural anomaly" OR "Architectural

Searched for ("Architectural erosion" OR "Architectural degradation" OR "Architectural Smell" OR "Architectural anomaly" OR "Architectural drift" OR "Architectural decay" OR "Architectural inconsistency" OR "Architectural problem" OR "Architectural violation" OR "Architectural change" OR "Architectural inefficiencies" OR "Architectural debt" OR "Architecture degradation" OR "Architecture decay" OR "Architecture erosion" OR "Architecture drift" OR "Technical sustainability problem ") AND ("open-source" OR "open source" OR "OSS" OR "OSP") [new search] [edit/save query] [advanced search]

Searched The ACM Full-Text Collection: 573,946 records [Expand your search to The ACM Guide to Computing Literature: 2,876,592 records]

27 results found Export Results: bibtex | endnote | acmref | csv

276 document results

("Architectural erosion" OR "Architectural degradation" OR "Architectural Smell" OR "Architectural anomaly" OR "Architectural drift" OR "Architectural decay" OR "Architectural inconsistency" OR "Architectural problem" OR "Architectural violation" OR "Architectural change" OR "Architectural inefficiencies" OR "Architectural debt" OR "Architecture degradation" OR "Architecture decay" OR "Architecture erosion" OR "Architecture drift" OR "Technical sustainability problem ") AND ("open-source" OR "open source" OR "OSS" OR "OSP")

Edit Save Set alert Set feed
Wiley Online Library
("Architectural erosion" OR "Archi
Login / Reg

43 results for '("Architectural erosion" OR "Architectural degradation" OR "Architectural Smell" OR "Architectural anomaly" OR "Architectural drift" OR "Architectural decay" OR "Architectural inconsistency" OR "Architectural problem" OR "Architectural violation" OR "Architectural change" OR "Architectural inefficiencies" OR "Architectural debt" OR "Architecture degradation" OR "Architecture decay" OR "Architecture erosion" OR "Architecture drift" OR "Technical sustainability problem ") AND ("open-source" OR "open source" OR "OSS" OR "OSP")" anywhere

FIGURE 3. Initial results of the search process.

abbreviations; c) checking through matching the keywords in any relevant research study for the stated previous steps; d) using the Boolean operators "OR", "AND", whereas "AND" operator employed to associate with the key terms, "OR" employed to link synonyms, alternative words, and abbreviations to each other; e) incorporating the key terms to compose the final search string.

To better design the key terms, the researchers identified the question structure based on the population, intervention, outcome, and experimental design, as stated below:

- Population: Software architecture, OSS.
- Intervention: Architectural degradation estimation.
- Outcomes: Architectural decay detection, improved software quality.
- Experimental Design: Empirical studies, experimental studies, and case studies.

After conducting and completing the previous steps to form the keywords and after making sure that some tests are conducted by the search string on the selected libraries, the following inclusive search terms were adopted in our study: [("Architectural erosion" OR "Architectural degradation" OR "Architectural Smell" OR "Architectural anomaly" OR "Architectural drift" OR "Architectural decay" OR "Architectural inconsistency" OR "Architectural problem" OR "Architectural violation" OR "Architectural change" OR "Architectural inefficiencies" OR "Architectural debt" OR "Architecture degradation" OR "Architecture decay" OR "Architecture erosion" OR "Architecture drift" OR "Technical sustainability problem") AND ("open-source" OR "open source" OR "OSS" OR "OSP")].

The search string was investigated in the digital libraries through the keywords, abstract, and title of each study except for SpringerLink and web of science libraries, the search string was investigated through the full text since it is not easy to use the advanced search by keywords, title, and abstract.

b: RESEARCH RESOURCES

The selection of the research resources has a crucial role in the identification of an efficiency result of the SLR. Accordingly, the researchers must identify relevant and appropriate research resources to conduct their research as well as research resources that are public and inclusive to most research. Fig. 2 shows the research resources in this study that had been identified. In some research resources, there may be a need to refine and reformulate the search string due to the difference in its search engine structure from others. Moreover, there is a divergence of the grammars from one search engine to another. For instance, the search in ScienceDirect engine requires up to 8 operators whether "OR", "AND", or integrate them. If the search terms are more than 8 operators, it must be divided into parts, considering the attention to link the operators among them. The search process of conducting the SLR is accomplished by two steps. The first step is an automatic search by searching eight online databases (repositories) as shown in Table 3. The second step is a manual search by the backward-forward search approach to identify the relevant studies among the selected primary studies [66]. Google Scholar search engine was used to determine the citation of relevant studies in the chosen primary studies.

TABLE 3. **Online databases.**

Name	URL
IEEE Xplore	http://ieeexplore.ieee.org
Springer Link	http://link.springer.com.
Science Direct	http://www.sciencedirect.com
Scopus	https://www.scopus.com
ACM Digital Library	http://dl.acm.org
Web of Science	http://www.webofknowledge.com
Wiley Online Library	https://onlinelibrary.wiley.com/
Taylor & Francis Online	https://www.tandfonline.com/

2) STUDY SELECTION PROCESS AND SELECTION OF PRIMARY STUDIES

The preliminary list of studies was extracted from the initial search process, containing 827 (as shown in Fig. 2). The study selection process was conducted by one author and the other two authors checked the selection process. In the case Many steps were conducted to exclude the articles that were irrelevant to the topic, and at the same time to include related articles by following the guidelines and procedures by Kitchenham and Charters [57], Kitchenham [65], and based on some recommendations to detect the relevant studies for conducting the SLR and developing search strategies [66], [67]. In the first step, the inclusion and exclusion criteria (as illustrated in Table 2) were applied to obtain the final studies from this stage, which resulted in 597 articles. In the second step, the duplicate studies, found in many database resources during the search process were removed by using the Endnote reference manager, resulting in 535 articles. In the third step, the abstract and title were considered and assessed, whether or not it is related to the topic and the defined research question, yielding 164 articles. In the fourth step, reading the full text to evaluate and consider in-depth in order to issue the final decision to be included or excluded, producing 66 articles. In addition to articles that have two duplicates in this stage, the journal article has to be included rather than the conference paper, provided that it is up to date as in our study, resulting in five articles. In the fifth step, a snowballing search strategy was employed for tracing citations and references of the included studies to identify the relevant missing articles, resulting in 15 articles in the first iteration. In the second iteration, 15 articles were analyzed and no additional studies were found. The search alert was utilized for all the search resources presented in Table 3 to determine the related published articles after the date of the initial research process, producing two articles. In the final step, at the same time as reading the full text, the quality of the articles was evaluated, and six studies were excluded. After applying the criteria for exclusion and inclusion, the total

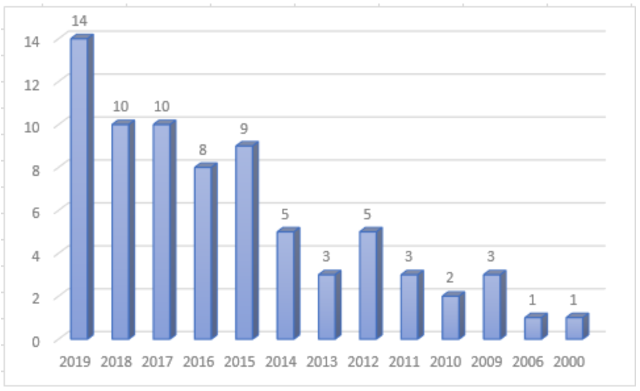


FIGURE 4. Initial studies over the year.

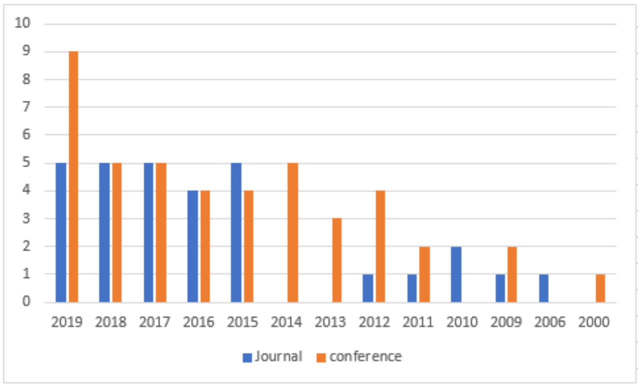


FIGURE 5. Initial studies per publication source.

of the primary studies was identified in this proposed study, involving 74 primary studies (as shown in Fig. 2). The primary studies are demonstrated in Appendix A. The classification of the primary studies is illustrated over the years in Fig. 4. The classification of the primary studies is described per publication source in Fig. 5.

3) STUDY QUALITY ASSESSMENT

Based on the inclusion criteria for primary studies as stated in [57], [65], an evaluation of the study quality was also applied in order to review and reconsider according to the assessment criteria that are more accurate and more detailed (as shown in Table 4). The evaluation of the study quality aimed at making the final decision for the included studies to ensure the quality in each study before the data extraction for analysis.

The quality assessment process was performed by one author and another author verified the selected studies for quality assessment. In addition, a discussion was also held at the point of disagreement. The study quality assessment was divided into three levels, which are high, medium, and low. Then, the scores for each question were identified in three parts. In the first part, number 1 means an accomplishment of the quality criteria entirely and clearly. In the second part, number 0 means does not meet anything from the stated quality assessment criteria. In the last part, number 0.5 means achieving the stated criteria partially.

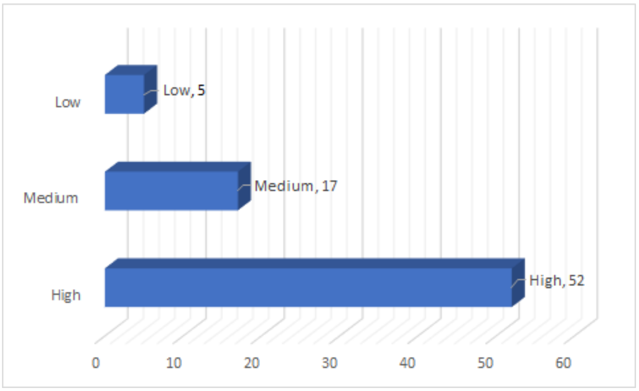


FIGURE 6. Distribution of study quality assessment levels.

TABLE 4. Quality assessment criteria.

QA ID	Quality checklist questions	Marked Score
QA 1	Is the goal(s) of the research clearly stated?	
QA 2	Does the research add well-motivated value to software architecture degradation in the OSS community?	The score "Yes" =1 / "No" = 0 / "partial" = 0.5
QA 3	Is the methodology (research design) well-defined and deliberate?	
QA 4	Is the study assessment explicitly reported?	
QA 5	Does the study provide a description of any one of the defined research questions?	

After applying assessment quality criteria, the scores of five criteria were collected for quality assessment. The range of the three levels was determined in the quality evaluation. If the range is between the score 4.5 – 5.0, it denotes a high level, and if the scope of score is between 3.5 - 4.0, it denotes a medium level, and if the range is between the score 2.5 – 3.0, it denotes a low level. Most of the scores are represented from the high level, while six studies were excluded since they did not meet the specified criteria. In Fig. 6, the distribution of studies is illustrated after applying the quality criteria for the three levels. The results of the quality assessment criteria for primary studies are described in Appendix B.

C. REPORTING PROCESS

Once the conducting process had been identified, the outcomes of a systematic review could be carried out adequately by extracting the appropriate data in line with the defined research questions. Then, the data were synthesized to identify the final view of the research and a conclusion was made on what revolves around the scope of this research in the current time and future research.

1) DATA EXTRACTION

Once the primary studies had been selected to be utilized for SLR, the data extraction proceeded to precisely record the information to address the defined research questions.

TABLE 5. Data extraction items of the primary studies.

Data extracted item	Description
Study ID	The unique identifier number for each study.
The publication year	The identification of published paper date until the year 2020.
Study type	Describe the publication type such as journal, conference, a chapter of the book.
Study title	The main goal of the document per each paper, which appears in the search stage.
Study aim	Outlining the main goal of each study
Focus of study	Essential topic scope, concepts, motivation/possible reasons, proposed solutions, and descriptions of the degradation symptoms indicators.
Constraints and limitations	Determining restrictions and downsides in the application of an approach along with the specified domain for future plans and recommendations in the research.
Study designs / Research method used for data collection	Qualitative/Quantitative/mixed/empirical, along with method design such as experimental, case study, survey, review.
Obtained results	Demonstrates the key finding per each study and extent the effectiveness of realized outcomes.
Datasets	The list of the OSS projects on which the study has been applied.
Degradation of software architecture	Characterize what specifically erodes in the system architecture

In order to facilitate the data extraction process, data design forms according to guidelines in [57] were used from the chosen studies based on their relevance to the research questions. Appendix A demonstrates the details of the primary studies references (SID, title, author, year publication, and publication source). Finally, the data extraction form design was implemented on 74 primary studies with a brief description of all the extracted data items presented in Table 5, which was considered as the main source of data synthesis. The data extraction was saved in the MS Excel spreadsheets and Endnote reference manager.

2) DATA SYNTHESIS

By tabulating the extracted data for the required items in the MS Excel Spreadsheets and the Endnote reference manager, it is possible to synthesize the data with the aims to summarize and collect the results of the extracted items for the selected primary studies to answer the defined research questions [57]. Besides, it is important to determine whether the results obtained from the primary articles are identical or inconsistent to one another. The data synthesis can include the descriptive data (non-quantitative), along with a descriptive synthesis and sometimes it is possible to involve a quantitative synthesis. In our study, the data were extracted to encompass descriptive data (e.g., causes of the decay, the proposed solutions of addressing the degradation, a list of the erosion indictors symptoms) and quantitative data (e.g., the value of obtained results accuracy, which contributes to the extent of effectiveness of the suggested solutions).

IV. RESULTS

This section presents the results of the review to answer the defined research questions outlined in Table 1. The research questions have been answered by the final sample of the primary studies, which was restricted after applying the stated general and quality criteria between the year 2000 to March 2020, in which the research process ended, and followed by the inception for synthesis and report of the final data.

A. RQ1) WHAT ARE THE POSSIBLE REASONS FOR THE OCCURRENCE OF ARCHITECTURAL DEGRADATION?

This question identifies the possible causes that contribute to architectural degradation in the OSS community. The results of the first research question show several reasons, which differ in terms of the actual contribution to the occurrence of architectural erosion.

The results indicated that most of the causes of architectural degradation, which have a significant actual impact, are the rushed evolution of systems, recurring changes, lack of developer's awareness, time pressure and accumulation of design decisions. In the rushed evolution, numerous architectural problems, such as smells, tend to increase through successive system versions. In recurring changes, there are escalated risks on software architecture in making mistakes whenever the systems become more complex due to the frequent changes. For example, adapting new functionalities and features, new technologies, irresponsible or unintended addition, uncontrolled and unsuitable changes in its implementation or removal, and modification of architectural design decisions. In relation to the lack of the developer's awareness, there are problems with a high level of severity, which are not introduced to identify the level of architecture decay severity by developers due to lack of an explicit understanding of architectural basic knowledge, insufficiency of business knowledge, adopting and selecting the inexperienced developers, absence of long-term developer commitment to the project, writing unsuitable and improper code, or the practice and training to generate OSS projects as a distraction or hobby. In time pressure, it may affect the introduction of temporary solutions by the developers under a deadline pressures

TABLE 6. Potential reasons for the occurrence of the architecture DECY.

Reference / study	Potential reasons
[S01], [S14].	Scarcity of traceability of history and analysis of software systems architecture.
[S03], [S04], [S55], [S45], [S39], [S41], [S27], [S28], [S31], [S33], [S20], [S21], [S10], [S14], [S72], [S56].	The rush of software evolution.
[S06], [S36], [S28], [S67].	Violations of the original Architecture rules.
[S07], [S74], [S71], [S43], [S46], [S40], [S28], [S30], [S17], [S23], [S67].	Lack of developer's awareness.
[S08].	Ignoring quality attributes.
[S09], [S46].	Insufficient development tools.
[S70], [S31].	Unresolved differences between conceptual and concrete software architecture.
[S68], [S57], [S55], [S32].	Structural complexity.
[S66],[60], [S55], [S43], [S45], [S48], [S40], [S25], [S26], [S33], [S19], [S20], [S21], [S72], [S59].	The recurring change.
[S62], [S30], [S32], [S33].	Lack of architecture documentation.
[S53], [S47], [S51].	The poor quality of the architecture design solutions.
[S43], [S46], [S28], [S30], [S17], [S23], [S24], [S51].	Time pressure.
[S46].	Inconsistent requirements.
[S46].	Organizational environment
[S38].	Accumulating architectural debts.
[S10].	Connection of the bug-prone files.
[S08], [S61], [S54], [S44], [S25], [S17], [S24], [S59].	Accumulation of architectural design decisions.

or time constraints and workloads, which contributes to accumulating architectural debts affecting software architecture over time.

In the accumulation of design decisions, architectural design decisions occur consciously or unconsciously, negatively influencing the quality attributes especially maintainability and evolvability.

Furthermore, there are other reasons such as structural complexity, lack of architecture documentation, poor quality of the architecture design solutions, organizational environment, insufficient development tools, violations of the original architecture rules, scarcity of traceability of history and analysis of software architecture, inconsistent requirements, accumulating architectural debts, the connection of the bug-prone files, and ignoring quality attributes demonstrating in Table 6.

The results also indicated the extent to which the frequency of the causes occurrence according to their statement and impact on the case study in the chosen primary studies at a different ratio. The rush evolution of the systems scored 19.77 %, the recurring changes 18.60 %, the lack developer's awareness 12.79 %, the time pressure and accumulation of architectural design decisions 9.30 %, the violation of the original architectural rules and structural complexity 4.65 %, the poor quality of the architecture design solutions 3.49 %, the scarcity of traceability of history and analysis of software systems architecture, insufficient development tools, and unresolved differences between conceptual and concrete software architecture 2.33 % and ignoring quality attributes, inconsistent requirements, organizational environment, accumulating architectural debts, and connection of the bug-prone files 1.16 %. Fig. 7 shows the frequency for the cause occurrence according to their statement in the chosen primary studies.

B. RQ2) WHAT ARE THE KEY INDICATORS THAT HAVE A PROMINENT ROLE IN THE PERMANENCE OF THE ARCHITECTURAL DEGRADATION SYMPTOMS?

In the first research question, we identified the potential reasons, which clarified the occurrence of architectural degradation. In this question, we specified the prominent key indicators of the architecture degradation symptoms, which appear as a result of the presence of the possible causes.

We found the four key indicators (as illustrated in Table 7), which are code smell, architectural smells, architecture technical debt, and the violation of the architectural constraints. As earlier stated, the key indicators of architectural degradation symptoms are classified. To illustrate, firstly, the code smells were divided into two groups; the code smells individual and the code smells agglomerations (as shown in Table 8). Secondly, the architectural smells were divided into three groups; architectural hotspots, architectural bad smells/architecture anti-patterns, and architectural change/instability (as shown in Table 9). Thirdly, architectural constraints violation was divided into three groups;

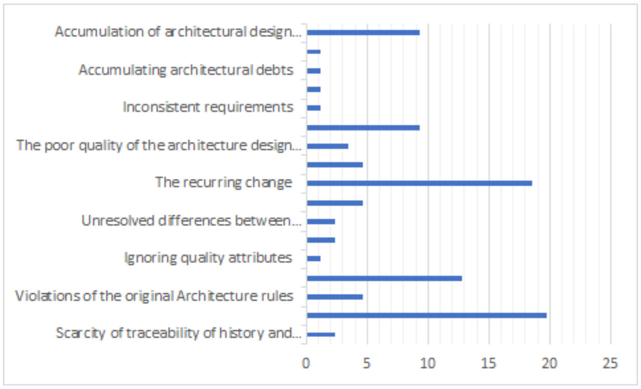


FIGURE 7. Frequency of the cause occurrence in chosen primary studies.

TABLE 7. Key indicators of the architecture degradation symptoms.

Reference / study	Key indicators
[S03], [S44], [S39], [S15], [S18], [S07], [S08], [S61], [S62], [S57], [S58], [S52], [S47], [S31],[S16], [S17],[S23], [S24],[[S64],[S42], [S69],[S34], [S50], [S51].	Code smells
[S52], [S47], [S29], [S21], [S11], [S12], [S13], [S10], [S14], [S65], [S01], [S68], [S60], [S20], [S45], [S50], [S19].	Architectural smells
[S07], [S54], [S38], [S41].	Architectural technical debts
[S04], [S09], [S74], [S70], [S47], [S43], [S36], [S28], [S08], [S63], [S46], [S47], [S28], [S72], [S67].	Breaking of the architectural constraints

TABLE 8. Code smells groups.

Reference / study	Code smells group
[S07], [S08], [S61], [S62], [S57], [S58], [S52], [S47], [S31], [S16], [S17],[S23], [S24], [S64], [S42], [S69], [S34], [S50], [S51].	Code smells individual
[S03], [S44], [S39], [S15], [S18].	Code smells agglomeration

internal attributes' violation, violations of object-oriented design characteristics, and architecture inconsistencies (as shown in Table 10).

The results show the extent to which the frequency of the key indicators appearance of the architectural decay symptoms as reported by conducting an empirical case study in the chosen primary studies at a different ratio. The code smells obtained 40.00 %, the architectural smells 28.33 %, the breaking of the architectural constraints 25.00 %, and the architectural technical debts 6.67 % (as demonstrated in Fig. 8).

According to each group of the key indicators groups of degradation symptoms, the results indicate that in code

TABLE 9. Architecture smells groups.

Reference / study	Architectural smells group
[S65], [S14].	Architectural hotspots
[S52], [S29], [S21], [S11], [S12], [S13] [S47], [S10].	Architectural bad smells / Architecture anti-patterns
S45], [S50], [S19], [S01], [S68], [S60], [S20],	Architectural change/ instability

TABLE 10. Group of the architectural constrain's violation.

Reference / study	Architectural violation group
[S08], [S68].	Internal attributes' violation
[S08], [S63], [S46], [S47], [S36], [S28], [S72], [S67].	Violation of object-oriented design characteristics
[S04], [S09], [S74], [S70], [S43].	Architecture inconsistencies

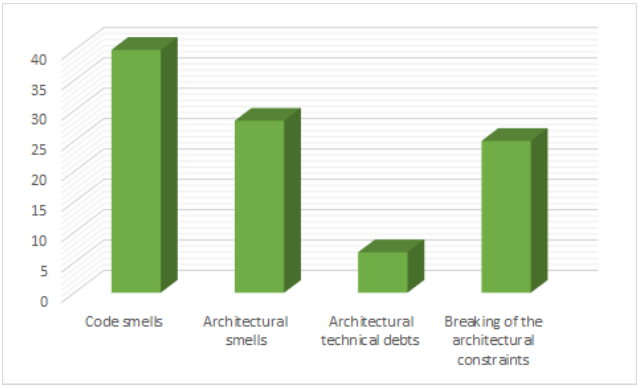


FIGURE 8. Frequency of key indicators appearance of the architectural decay symptoms in chosen primary studies.

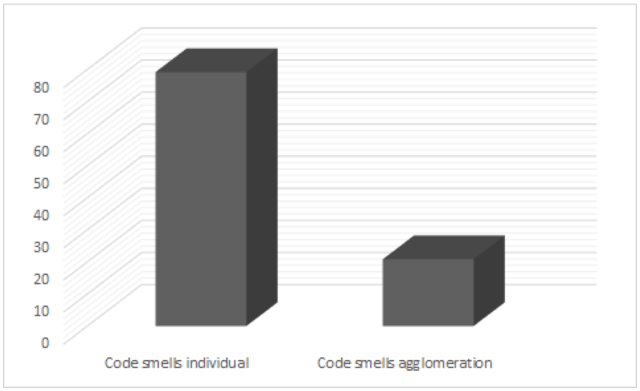


FIGURE 9. Frequency of the appearance key indicators of code smells group.

smells group, code smells individual obtained 79.17 %, while the code smells agglomeration obtained 20.83 % (as illustrated in Fig. 9). In architectural smells group,

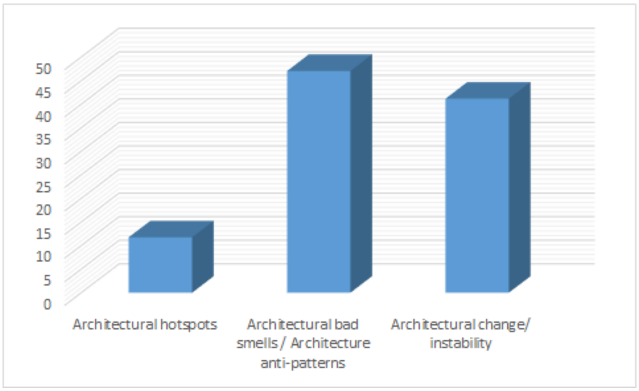


FIGURE 10. Frequency of the appearance key indicators of architectural smells group.

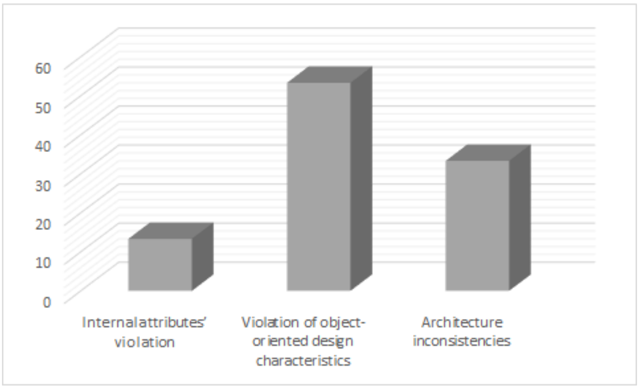


FIGURE 11. Frequency of the appearance key indicators of Group of the architectural constraint's violation.

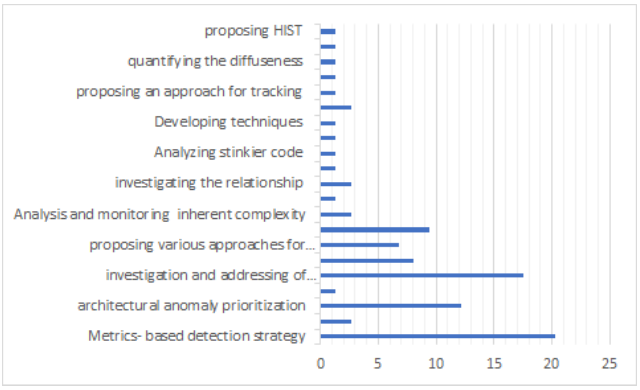


FIGURE 12. Proportion of used solutions to detect architectural decay in studies.

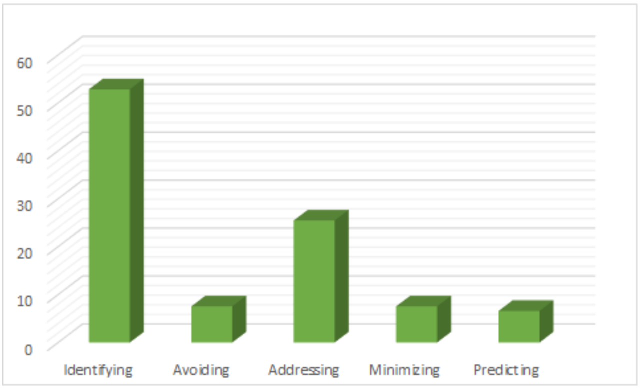


FIGURE 13. Classification proportion of proposed solutions type.

architectural hotspots obtained 11.76 %, architectural bad smells/architecture anti-patterns 47.06 %, and architectural change/instability 41.18 % (as stated in Fig. 10). In a group of the architectural constraints violation, the internal attributes violation obtained 13.33 %, violation of object-oriented design characteristics 53.34 %, and architecture inconsistencies 33.33 % (as shown in Fig. 11).

C. RQ3) ARE THERE PROPOSED SOLUTIONS THAT CONTRIBUTE TO IDENTIFYING / ADDRESSING / MINIMIZING / AVOIDING / PREDICTING ARCHITECTURAL DEGRADATION?

The second question discusses the key indicators of architectural decay symptoms and resulting problems as well as the risks for OSS projects. Based on the findings, this question presents the proposed solutions for symptoms of the key indicators in each primary study.

The results showed many solutions that vary in terms of strategies of the used solutions such as models, measurements, approaches, algorithms, tools, techniques, and methods, that may integrate more than one instrument and strategy according to the proposed solution (as shown in Table 18).

The most important of these solutions are the metrics-based detection strategy, prioritization of architectural anomalies, investigating and addressing of architectural rules violations, applying refactoring strategy, applying architectural recovery strategy, and other proposed solutions (as illustrated in Table 11). These stated solutions are originally considered general solutions. Hence, the most important used solutions have been detailed into more clarified solutions as shown in the following tables. For example, metrics-based detection strategy is shown in Table 12, prioritization of architectural anomalies is clarified in Table 13, investigation and addressing of architectural rules violations are indicated in Table 14, applying refactoring strategy is illustrated in Table 15 and applying architectural recovery strategy is shown in Table 16. The proposed solutions are also categorized according to the stated mechanism, whether this mechanism is the identification, avoidance, addressing, reduction, or prediction of architectural degradation within the environment of OSS projects. Table 17 shows the classification of presented solutions types.

The results also indicated that the most suggested solutions to be used are the metrics-based detection strategy obtained 20.27 %, investigation and addressing of architectural rules violations 17.57 %, architectural anomaly prioritization 12.16 %, architectural recovery strategy 9.46 %, refactorings strat-

TABLE 11. Proposed solution of architectural decay.

Reference / study	Proposed solutions
[S01], [S04], [S61], [S52], [S54], [S43], [S35], [S39], [S26], [S20], [S50], [S13], [S44], [S60], [S57].	Metrics-based detection strategy.
[S02], [S10].	Exploring the architectural impact of the bug-proneness and change-proneness that violates the design principles by connections among files.
[S03], [S58], [S48], [S31], [S16], [S17], [S13], [S41], [S42],	Prioritization of critical architecturally smells and code anomalies relevant to the architectural problems.
[S05].	Code-based multilevel analysis method to detect erosion points by detecting the changed pairs of each level.
[S06], [S63], [S55], [S47], [S33], [S72], [S74], [S67], [S38], [S56], [S36], [S28], [S32].	Investigation and addressing of architectural rules violations.
[S09], [S49], [S29], [S64], [S34].	Proposing various approaches for automatic detection, repair of architectural problems, and support software developers during the development in Java projects.
[S70], [S71], [S37], [S25], [S22], [S59], [S70].	Applying architectural recovery strategy.
[S68], [S65].	Analysis and monitoring inherent complexity.
[S66].	Presenting the graph kernel approach to calculate the structural distance of architecture between two revisions of a software system.
[S62], [S18].	Investigating the relationship between code anomalies and architecture problems.
[S45].	Identifying causes of architecture changes between developers through intermediary media by using a top-down and a bottom-up approach.
[S30], [S08], [S62], [S46], [S23], [S53].	Applying refactoring strategy
[S15].	Analyzing stinkier code rather than smells occurring in isolation based on the semiotic Engineering theory
[S40].	Detecting architectural tactics to understand underlying design decisions
[S19].	Developing techniques for identifying and predicting architectural changes by using the readily available information in the issue and code repositories of software systems
[S21], [S11].	Proposing a prediction model for identifying architectural smells link prediction (LP) and historical AS data
[S12].	Proposing an approach for tracking the individual smell instances along with system evolution
[S14].	Proposing a novel model, Active Hotspot (AH) for detecting and monitoring the emergence and evolution of software degradation by tracking how files and their relations are changed within each issue
[S24].	Quantifying the diffuseness of the problem in term of how frequently code smells occur together
[S69].	Proposing DEtection & CORrection (DECOR), a method for the specification and detection of code and design smells using a unified vocabulary and domain-specific language (DSL)
[S51].	Proposing an approach, named HIST (Historical Information for Smell detection), to detect smells based on the change in history information mined from the versioning systems

egy 8.11 %, proposing various approaches for automatic detection 6.76%, exploring the architectural impact, analysis and monitoring of inherent complexity, proposing a prediction model 2.70 %, code-based multilevel analysis, presenting the graph kernel approach, identifying causes of architecture changes, analyzing stinkier code, detecting architectural tactics, developing techniques, proposing an approach for tracking, proposing active hotspot, quantifying the diffuseness, proposing Detection, and proposing HIST 1.35 % as shown in Fig. 12.

Additionally, the results of the proposed solution type classification showed that identifying mechanism obtained 52.83%, addressing mechanism 25.47 %, avoiding and minimizing mechanism 7.55 %, and predicting mechanism 6.60% as demonstrated in Fig. 13. The proposed solutions were classified based on the introduced solution strategies in those primary studies. Fig. 14 demonstrates the proposed solutions taxonomy of the architectural decay of OSS projects.

D. RQ4) HOW EFFECTIVE ARE THE PROPOSED USED SOLUTIONS IN THE PRIMARY STUDIES?

The previous question about identifying proposed solutions to handle architectural degradation within the OSS projects environment has been earlier clarified. This question determines the extent to which the solutions are effective and efficient as well as to what extent these proposed solutions can achieve the contributions to address architectural degradation.

Since the proposed solutions are divided according to specific mechanisms, therefore, the effectiveness of the solutions is identified as previously stated in the mechanisms. There

TABLE 12. Detail solutions for the metrics strategy.

Reference / study	Details solutions of the metrics-based detection strategy
[S01], [S20].	Measuring the instability or stability to identify software components in developing future versions of evolving software systems.
[S04], [S52].	Exploring/estimating the effectiveness and effort of suitable code source metrics to reveal architecturally-relevant code anomalies.
[S61].	Improving the automatic detection of code anomalies by exploiting the extracted information based on a set of static code metrics
[S54].	Defining ATD indicators by modularity metrics.
[S43], [S26].	Characterizing the packages by maximizing cohesion and minimizing coupling in the hope that optimization will reorganize its modules to minimize dependencies among packages.
[S35].	Measuring how well the software is decoupled into small and independently replaceable modules.
[S44], [S39].	Measuring code anomalies that "flock together rather than individual anomalies that introduce a sufficiency of locating design problems system.
[S13].	Identifying architectural smells that most critical ones to prioritize refactoring efforts and prevent software architecture erosion in terms of locating in an important part of the system and criticality associated with each smell.
[S50].	Measuring the detection of code smells and acknowledging their relations and co-occurrences.
[S60].	Proposing a novel approach ADvISE to investigate some metrics (code decay indicators) on software.
[S57],	Exploring architecture-sensitive implementation metrics by the code elements.

are many solutions provided using metrics-based detection mechanism through which the effectiveness of the solutions is recognized.

The architectural instability remains growing and the metric values increases instead of decreases as the system evolves [S01], [S20]. Source code metrics show the probability that classes contribute to generating the architectural inconsistencies [S04] and the effectiveness of metrics-based strategies indicate that it could be applied with significant confidence [S52]. The detection strategies efficiency was not precise to determine code anomalies since more than 50 % of the explored code anomalies are not associated with architectural problems and surprisingly, more than 50 % of the false negatives that can be associated with architectural problems are made by automated strategies [S61]. The modularity indicators IPCI and IPGF showed a considerably negative interrelationship with the normalized ANMCC, while other indicators do not [S54]. The prediction models based on AbdeenMod+RM metrics achieved an acceptable accuracy, while conventional metrics is outperformed in the fault prediction modeling [S26]. On the other hand, the search-based module clustering showed that its original architecture has been lost because of the new features addition and maintenance [S43]. Decoupling Level (DL) measure has a considerably negative correlation with architecture maintainability, thereby can be considered a more valuable metric compared to the coupling level metric [S35]. The agglomerations anomalies are better than the individual anomalies. Around 50% of the syntactic agglomerations correlates with architectural problems while around 80% of the semantic agglomerations correlates with architectural problems [S44], [S39]. Both (high PageRank and high criticality) metrics provide valuable information to the developers, of which information can be utilized to explore the danger of architectural smells [S13]. The use of architecture-sensitive information for code anomalies detection will introduce critical knowledge for engineers to identify and address smells immediately [S57].

Regarding the solutions sufficiency of anomalies prioritization strategy, the results showed that the agglomeration flood standard introduced good outcomes for the determination of the architectural problem, but it does not necessarily represent the most critical smells [S03], [S13]. Furthermore, the recommended heuristics and architecture blueprints are able to improve and rank the prioritization process compared to the metric-based strategies, from 20 % to 60 % of critical code smells, thereby it motivates for prioritizing architecturally relevant code smells [S48], [S16], [S58]. [S31] indicated that the most optimal models are not accurate enough to specify classes relevant to architectural inconsistencies dependent on the code smell. In contrast, the context-based smell prioritization techniques indicated that relevant results introduce more improvement than the severity-based smell prioritization [S17]. Moreover, all algorithms achieved high performances such as the ones that were obtained by J48 and Random Forest while the worst performance was obtained by the support vector machines, suggesting that machine learning implementation to the detection of the code smells may provide high accuracy (>96 %) [S42].

TABLE 13. Detail solutions for prioritization anomalies.

Reference / study	Details solutions of prioritization architectural anomaly
[S03], [S13].	Identifying the prioritization of architecturally smell agglomerations to focus on relevant architectural problems.
[S58].	Prioritization of code anomalies based on architecture sensitiveness (proposed heuristics).
[S48], [S16].	Prioritizing code anomalies by architectural blueprints to identify the strength of the relationship between code anomalies and architectural design problems.
[S31].	Prioritizing classes that violate the intended architecture for architectural repair.
[S17].	Prioritizing code smells from code smell detectors by considering developers' current context to support of the prefactoring phase.
[S41].	Prioritization of identifying and estimating the debt at the architecture level.
[S42].	Machine learning technology for classifying code smells by examples.

TABLE 14. Detail solutions for architectural rule's violations.

Reference / study	Details solutions for applying refactoring's strategy
[S08], [S62].	Applying refactoring for removing architecturally design problems.
[S46].	Proposing architectural repair recommendation through a recommendation engine called ArchFix for providing refactoring guidelines.
[S23].	Proposing a technique to detect code fragments incompliant to the architecture as fine-grained architecture smells.
[S53].	Repairing architectural smells by changing the structure and the behaviors of the internal system elements without changing the external behavior.
[S30]	Applying longitudinal case studies from a SACC perspective to obtain a deeper understanding of architectural erosion, its impact, and evolution.

Considering the solutions to the architectural rules violations, 99 % of the systems introduced violations at least one architectural principle type, and one of the principles was not followed in 60 % of the investigated entities [S06]. In addition, many violations were solved and introduced with time, and they also reappeared after they have been solved in future releases of the project [S63]. In the case of DSpace, the violations of layering were identified by the tool and then added as a code cleanup activities list [S72]. For the modularity violations, 231 violations (47 %) were identified from 490 modification demands of Hadoop, and 152 (65 %) violations were confirmed, while from 3458 modification demands of Eclipse JDT, 399 (12%) were identified and 161 (40 %) violations were confirmed [S67], 325 dependency violations were identified, whereas 70 % recommended refactoring for removing code anomalies [S55]. All the inference rules have a significant impact and efficiency on the detection of the violation [S23]. [S33] used software architecture conformance checking (SACC) to find out the architectural violations in specific files compared to normal files. Appearing code churn for files that have architectural violations become similar to natural files after removing violations. ArchLint approach detected 389 and 150 architectural violations, with an inclusive precision of 62.7 % and 53.8 % [S36]. The DCL 2.0 language shows a carefully captured of the architectural model for the system where 771 architectural violations were detected; 74 % of the violations were discovered by the new restrictions suggested in DCL 2.0 [S28].

Regarding refactoring strategy solutions, refactoring has no positive effect on the variety and intensity of any the symptoms indicators, of which around 66 % of all refactoring did not contribute to the repairing of architecturally relevant code smells [S08], [S62]. On the other hand, ArchFix pointed out the proper refactoring for 655 (79 %) out of 828 violations discovered [S46]. The profound understanding of the architectural decay and its impact showed an increasing trend in the degradation pre-refactoring and a decreasing trend in post-refactoring, producing the erosion that is detached from size after the post-refactoring termination [S30]. As for the ArCatch, an architectural conformance checking approach proved to be beneficial in the specification of the current exception handling decay issues and its reasons by detecting 7 violations of the design rules where the 6 design rules corresponded to all the versions [S32].

As for architectural recovery strategy solutions, the structural- and lexical-based layering techniques outperformed structural-based approaches to recover the software architecture of object-oriented (OO) systems [S37]. A more

TABLE 15. **Detail solutions for applying refactoring's.**

Reference / study	Details solutions of the architectural rule's violations
[S06].	Proposing Design Tests approach to check whether the implementation is following the architectural rules or not.
[S63].	Focusing the architectural violations lifecycle and location over time to form an initial body of knowledge on architectural violations.
[S55].	Proposing an approach (move class) to repair architecture violations through reengineering and refactoring techniques.
[S47].	A novel technique to automatically identify and rank causes of violations, based on the detection of so-called symptoms that are similar to anti-patterns or bad smells.
[S33].	Measuring the code churn and code ownership before and after the refactoring to discover large files containing violations in increasing code churn than large files before the refactoring.
[S72].	Proposing a human-assisted approach to identify the intended organization of the modules into layers by analyzing the source code to avoid the violations.
[S74].	Removing most of these anomalies using forward and reverse architecture repair.
[S67].	Proposing an approach Clio to detect and locate modularity violations based on Baldwin and Clark's design rule theory and design structure matrix (DSM) modeling.
[S38].	Proposing an approach ArchDebt by novel history coupling probability (HCP) and index file groups to identify architectural flaws.
[S56].	Proposing TamDera1, a new Domain-Specific Language (DSL), for specifying rules to detect architectural degradation and providing rules to reuse them.
[S36].	Proposing ArchLint, an approach based on a combination of static and historical source code analysis.
[S28].	Proposing DCL 2.0—an extension of the original DCL (dependency constraint language) technique to tackle the problem of divergences between the planned architecture and source code.
[S32].	Proposing ArCatch: architectural conformance checking solution to deal with the exception handling design erosion.

TABLE 16. **Detail solutions for architectural recovery.**

Reference / study	Details solutions of the architectural recovery strategy
[S73].	Proposing an approach Focus for recovering and evolving architectures of undocumented OO applications and stem architectural erosion.
[S71].	Recovering the SE concepts of software design and architectures and identify architectures evolve or decay during the system's evolution.
[S37].	Proposing an approach that combines lexical and structural information of a given system to recover its layered architecture.
[S25].	Introducing a novel approach, Architecture Recovery, Change, And Decay Evaluator (ARCADE).
[S22].	Developing a technique RecovAr, based on ACDC and ARC for automatically recovering design decisions from the project's readily available history artifacts.
[S59].	Establishing a set of "ground truths for verifying the accuracy by the system's architects or developers who have intimate knowledge of the underlying application and problem domain.
[S70].	Extending the Software Reflexion Model and tailoring it for the analysis of MDSD projects using a clustering technique for architecture recovery.

suitable manner of evaluating and understanding architectural change is by investigating recovered conceptual architecture of the system either at the general structure level or the individual components level [S25]. RecovAr technique can accurately uncover the architectural design decisions embodied in the systems, recovering 75% of the decisions with a precision of 77% [S22]. In [S59], the results of the study showed that constructing a ground-truth architecture for the wide systems is feasible unlike prior intuition for recovering architecture, which claimed that it is infeasible. Hence, the outcomes can assist the improvement of the understanding of software architecture.

With respect to proposing various approaches for automatic detection, repair of architectural problems and support

TABLE 17. Classification of the proposed solution type.

Reference / study	Classification of the proposed solution type
[S01], [S04], [S61], [S52], [S54], [S35], [S39], [S26], [S20], [S13], [S50], [S44], [S02], [S03], [S58], [S31], [S16], [S17], [S05], [S06], [S63], [S55], [S47], [S72], [S41], [S62], [S67], [S23], [S38], [S09], [S49], [S29], [S64], [S34], [S22], [S25], [S37], [S71], [S66], [S60], [S62], [S18], [S57], [S45], [S36], [S28], [S30], [S40], [S19], [S12], [S14], [S24], [S42], [S69], [S56], [S51],	Identifying /detecting
[S13], [S33], [S70], [S70], [S68], [S45], S28], [S12], [S14],	Avoiding/ preventing
[S01], [S10], [S48], [S16], [S17], [S05], [S55], [S47], [S72], [S74], [S08], [S62], [S67], [S46], [S53], [S09], [S59], [S22], [S37], [S71], [S70], [S70], [S65], [S62], [S30], [S32], [S15],	Addressing/ repairing
[S43], [S35], [S03], [S58], [S48], [S08], [S64], [S19],	Minimizing
[S13], [S04], [S41], [S60], [S19], [S21], [S11],	Predicting

TABLE 18. Instruments and strategies of the used solutions.

Reference / study	Instruments and strategies of the used solutions
[S01], [S04], [S09], [S71], [S86], [S65], [S61], [S62], [S57], [S58], [S52], [S54], [S43], [S48], [S35], [S37], [S39], [S41], [S25], [S26], [S28], [S31], [S33], [S16], [S18], [S19], [S20], [S10], [S11], [S13], [S64], [S42], [S72], [S56], [S50],	Using metrics
[S02], [S04], [S05], [S70], [S66], [S61], [S62], [S63], [S47], [S31], [S33], [S11], [S14],	Using Model
[S03], [S02], [S04], [S06], [S08], [S09], [S71], [S68], [S60], [S61], [S62], [S63], [S57], [S58], [S52], [S53], [S54], [S55], [S43], [S44], [S45], S46], [S47], [S48], [S13], [S49], [S35], [S36], [S39], [S40], [S41], [S25], [S26], [S27], [S28], [S29], [S31], [S32], [S15], [S16], [S21], [S23], [S10], [S11], [S12], [S14], [S64], [S42], [S72], [S34], [S56],	Using tools / Techniques
[S03], [S05], [S60], [S55], [S39], [S25], [S31], [S15], [S19], [S20], [S21], [S14], [S42], [S69], [S72],	Using algorithms/machine learning
[S04], [S06], [S09], [S74], [S73], [S66], [S60], [S63], [S55], [S45], S46], [S49], [S36], [S37], [S38], [S41], [S25], [S27], [S28], [S30], [S32], [S17], [S19], [S21], [S22], [S10], [S12], [S42], [S72], [S56], [S67], [S59], [S51],	Using approach / concepts / theories and principles / logical assumption
[S05], [S08], [S68], [S65], [S61], [S62], [S57], [S58], [S52], [S53], [S54], [S43], [S48], [S35], [S44], [S39], [S40], [S26], [S28], [S31], [S33], [S15], [S16], [S18], [S13], [S24], [S69],	Using method/ Mechanism

software developers during the development in Java projects, the DARCY approach has a significant impact in minimizing the attack surface and enhancement of the encapsulation, maintainability and security [S09]. The automatic architecture validation approach introduced a considerable enhancement by the tool application during the development [S49]. Supporting the automatic analysis of software architecture by Arcan tool, reveals a precision of 100% of the architectural smells except for the external components, which reported false negatives [S29]. The experiment with various tools for code anomalies exploration showed providing various answers, although depending on similar detection algorithms and the tool precision usually differ based on the code anomalies, in which it is analyzed and the levels and contexts difference for the tools. [S64], [S34].

Concerning analysis and monitoring inherent complexity solutions, the evolutional changes that took place in the architectural stability and structural complexity showed the complex packages excessively as all dependencies are not generated balanced-well, and the dependency relationships nature between packages is relatively abstract [S68]. [S65] found out that the architectural flaws (or multiple-component) are more complex to repair than other flaws as over 50 % of the topmost flawed components must change coincidingly with other components to repair a multiple-component.

With respect to exploring the architectural impact of the bug-proneness and change-proneness that violate design principles by connections among files, the DRSpaces model showed that the architectural impact is persistently connected among bug-prone files over time [S02]. Moreover, the architecture anti-patterns detection approach indicated that it is robustly associated with greater change-proneness and bug-proneness files, which leads to increasing change and bug rate considerably. [S10].

In terms of investigating the relationship between code anomalies and architecture problems, the correlation of the architectural designs and code smells agglomerations

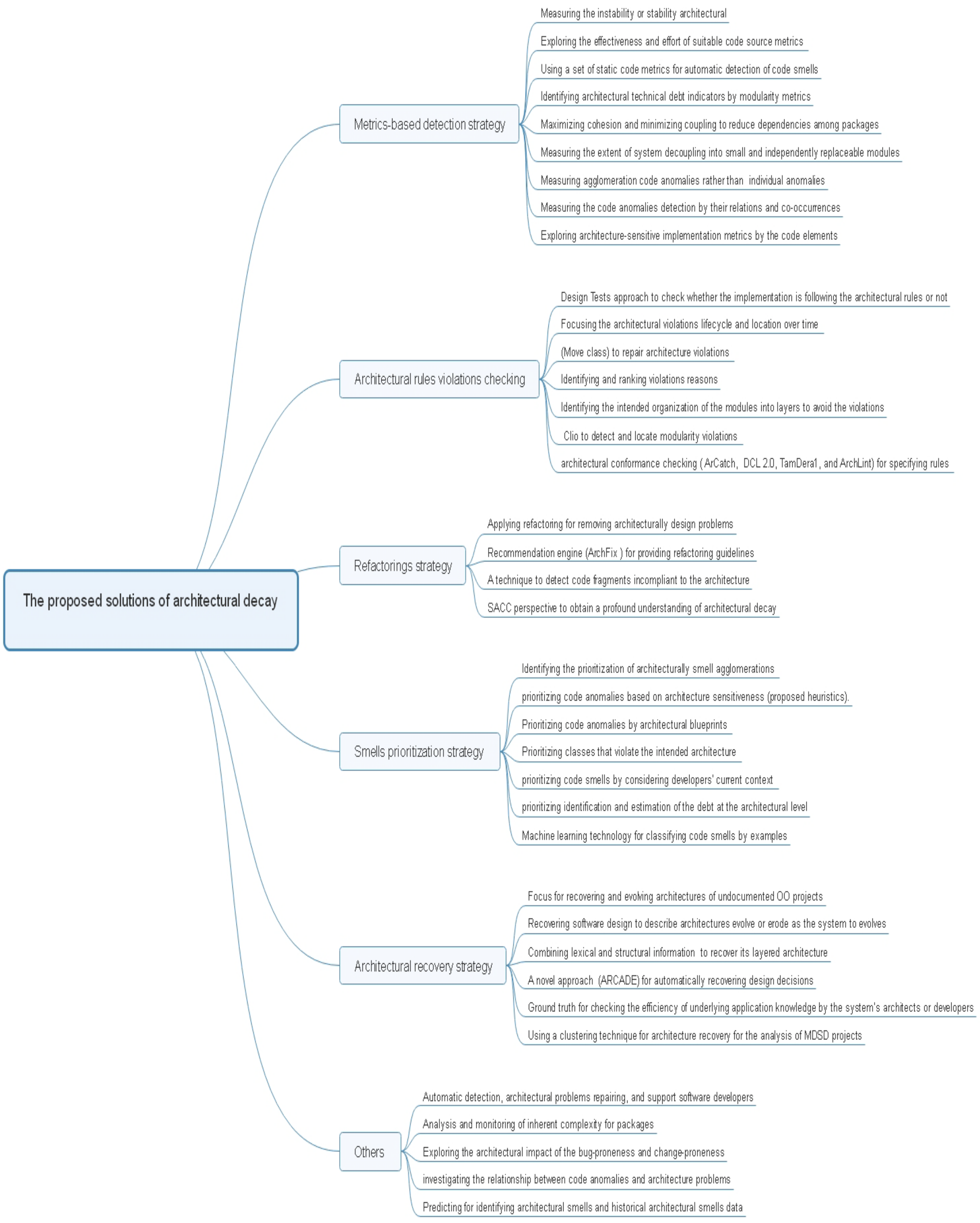

The proposed solutions of architectural decay
Metrics-based detection strategy
Measuring the instability or stability architectural
Exploring the effectiveness and effort of suitable code source metrics
Using a set of static code metrics for automatic detection of code smells
Identifying architectural technical debt indicators by modularity metrics
Maximizing cohesion and minimizing coupling to reduce dependencies among packages
Measuring the extent of system decoupling into small and independently replaceable modules
Measuring agglomeration code anomalies rather than individual anomalies
Measuring the code anomalies detection by their relations and co-occurences
Exploring architecture-sensitive implementation metrics by the code elements
Architectural rules violations checking
Design Tests approach to check whether the implementation is following the architectural rules or not
Focusing the architectural violations lifecycle and location over time
(Move class) to repair architecture violations
Identifying and ranking violations reasons
Identifying the intended organization of the modules into layers to avoid the violations
Clio to detect and locate modularity violations
architectural conformance checking (ArCatch, DCL 2.0, TamDera1, and ArchLint) for specifying rules
Refactorings strategy
Applying refactoring for removing architecturally design problems
Recommendation engine (ArchFix) for providing refactoring guidelines
A technique to detect code fragments incompliant to the architecture
SACC perspective to obtain a profound understanding of architectural decay
Smells prioritization strategy
Identifying the prioritization of architecturally smell agglomerations
prioritizing code anomalies based on architecture sensitiveness (proposed heuristics).
Prioritizing code anomalies by architectural blueprints
Prioritizing classes that violate the intended architecture
prioritizing code smells by considering developers' current context
prioritizing identification and estimation of the debt at the architectural level
Machine learning technology for classifying code smells by examples
Architectural recovery strategy
Focus for recovering and evolving architectures of undocumented OO projects
Recovering software design to describe architectures evolve or erode as the system to evolves
Combining lexical and structural information to recover its layered architecture
A novel approach (ARCADE) for automatically recovering design decisions
Ground truth for checking the efficiency of underlying application knowledge by the system's architects or developers
Using a clustering technique for architecture recovery for the analysis of MDSD projects
Others
Automatic detection, architectural problems repairing, and support software developers
Analysis and monitoring of inherent complexity for packages
Exploring the architectural impact of the bug-proneness and change-proneness
investigating the relationship between code anomalies and architecture problems
Predicting for identifying architectural smells and historical architectural smells data

FIGURE 14. **The proposed solutions taxonomy of architectural decay.**

indicated that most of the god classes' m-LOC (67%) flocked around architectural concerns [S18], almost 65% of all code smells were correlated to 78% of all architecture problems [S62].

As for the prediction model for determining architectural anomalies and historical architectural smells data, the machine learning techniques to predict architectural smells, which depends on historical information showed that the prediction performance is very high [S11]. Furthermore, the link prediction (LP) techniques showed an acceptable achievement for the positive class and also indicate high recall concerning the realistic anomalies identification although a specific anomaly type can impact the recall and precision [S21].

The evolution of architectural smells detected by a tool, namely Arcan, showed that Hublike dependency smell is better in terms of complexity, current and future maintenance effort compared to cyclic dependency [S12]. Active Hotspot (AH) model showed that measuring as 100 issue fixes, ranging usually between 2 and 3, and seldom more than 5, to detect significant problems architecturally [S14]. In terms of the repeated occurrence for code anomaly, 59% of anomaly classes are influenced by more than one anomaly [S24]. The DSL allows the identification of many various anomalies that support the detection technique's effectiveness and the generality of its DSL [S69]. The historical information for anomalies disclosure indicated that over 75% of the anomaly is regarded as the design problems where the recall is between 58% and 100% and the precision is between 72% and 86% [S51]. Several design problems were detected by 36.36% of the developers, where these problems explicitly appeared when the analysis of agglomeration smells compared to individual smells [S15]. The easy accessibility of information in the issue and code repositories do not adequately involve the architectural significance of the current classification issues [S19]. The architectural distance indicators can be utilized for both flaw assessment and localization on the dependency graphs of consecutive releases of systems [S66]. The architecture degradation based on the multi-level analysis method indicated that it is useful for detection erosion points and more efficient for fixing architecture decay [S05].

V. DISCUSSION

This section presents an analysis and discussion of the findings based on the objectives of the study and the research questions. Moreover, some recommendations are stated to be an inception point for the direction of future research in this domain.

A. THE POSSIBLE REASONS FOR THE OCCURRENCE OF ARCHITECTURAL DEGRADATION (RQ1)

The findings showed that 17 of the most commonly occurred causes contribute to the architectural decay of the OSS community. Essentially, architectural degradation has numerous causes, which have been discussed by several researchers in their studies [11], [12], [23], [30]. However, these reasons have been discussed from limited aspects such as aging because of changes over time [11], identifying the reasons through only one case study [12] or based on their investigation of industrial case studies [27]. Consequently, these causes need to be further identified in terms of the frequency of their occurrence, especially in the scope of the OSS projects. Moreover, identifying the most important reasons will indeed contribute to the erosion according to the chosen primary studies, which contain several case studies in different domains for the OSS community. These causes differ in their impacts with regard to the actual contribution to the architectural decay prominence.

The rapid evolution of software out of 17 represents a stated reason of 19.77%, which means that the rapid development of software provides a significant chance for the growth of architectural smells across the system versions that increases obscuring in identifying architectural problems within the system. In addition, increasing the conceptual distance between the existing architecture and the design-time one leads to an increase in the amount and complexity of interactions between the elements of the system software. This reveals that the rapid development of software deviates from the original structure by releasing new versions of the system, especially when the development violations of the implemented architecture are not controlled in a systematic manner. One of the reasons that plays a major role in architecture degradation is the frequent changes that represent 18.6% of the stated reasons, such as adding, removing or modifying new features or requirements that have a major impact on the deviation of the architecture far from its origin. A change that is made without adapting the requirements and components leads to the erosion of the architecture over time. This also reveals that changes must be made by considering the adaptation to the current and future requirements to avoid architectural contradictions in the subsequent versions of the system. The lack of developers' awareness is considered a significant reason for the contribution to architecture decay within the OSS, of which it represents 12.79% of the reasons as a whole. The lack of awareness results from a lack of basic knowledge of architecture, writing inappropriate codes that may cause errors that are difficult to maintain later, the practice of building systems as a hobby, lack of training for developers in developing their programming skills and educating them in an analysis of the inherent risks in the system.

These reasons are considered the most contributing to degradation, while the rest of the reasons demonstrated in Table 6 are less important than the three stated reasons depending on what was declared in the selected primary studies. However, further studies should be conducted to find out the other causes as a rooted-deep study in digging up new causes that could have a significant contribution to identifying the architecture erosion, whether over the OSS or industrial systems.

B. KEY INDICATORS OF ARCHITECTURAL DEGRADATION SYMPTOMS (RQ2)

The findings showed that four of the key indicators of architectural degradation symptoms were detected within the OSS, which are code smells, architectural smells, architectural technical debts, and violation of the architectural constraints.

In reality, the code smells are autonomous from architectural smells [68], therefore, architectural smells have to be regarded as one of the main sources for identifying the decay symptoms [69]. While the code smells have a significant impact to increase software's defectiveness [70], [71] and change proneness [70], [72], they correlate with architectural problems [43]. Hence, the architectural smells existence does not imply the code smells existence and vice versa.

The code and architectural smells are considered the highest impact symptoms on the software architecture (as demonstrated in Fig. 8), since several studies were conducted to address the code and architectural smells, including their subcategories.

The results also showed a details group of the key indicators per each stated symptom. In the code smells group, the code smells individual and code smells agglomeration are typically key indicators of the code smells within the OSS. The code smells individual is the most frequent study of the key indicators of code smells than code smells agglomeration (as shown in Fig. 9). However, several studies addressing code smells agglomeration have proven that these smells agglomeration have a significant correlation with architectural problems from code smells individual.

In the architectural smells group, architectural bad smells (architecture anti-patterns) stood out as the most effective smells compared to architectural change (instability) and architectural hotspots smells (as shown in Fig. 10). This reveals that the architectural bad smells were studied in isolation and not combined with more than one smell, which was covered in the prior code smells. Consequently, further research in this domain should be conducted to identify the effect of architectural smells agglomeration and its correlation with architectural problems rather than the architectural smells in isolation in order to prove its inclusion or exclusion as one of the key indicators of the architectural decay symptoms.

In a group of the architectural constraint's violation, violations of object-oriented design characteristics emerged as the most influencing violations of object-oriented properties such as abstraction, encapsulation, modularity, and hierarchy while Internal attributes' violation is regarded essential for architecture software design such as complexity, coupling, and cohesion. Architecture inconsistencies violation that refers to expression, declaration or statements emergence in the source code, which do not correspond to the restrictions forced by the planned architecture that will lead to improper implementation decisions accumulation resulting in an architectural decay or erosion within the community of the OSS.

C. PROPOSED SOLUTIONS FOR ARCHITECTURAL DEGRADATION (RQ3)

The results showed numerous solutions that can be nominated to address architectural erosion. The metrics-based detection strategy solutions are most commonly used to detect architectural decay. These metrics manifest its performance in exploring and estimating the source code metrics effectiveness, addressing modularity to identify indicators of architectural debt, detecting code anomaly problems within the code elements using architecture-sensitive implementation metrics, automatic detection of code anomaly, identifying instability or stability software components, determining architectural anomalies that most critical ones, and measuring code anomalies with regard to their relations and co-occurrences. The cause behind using these metrics in several studies is that the models or intended architecture documentation are usually missing. Therefore, the code is commonly the unique and most significant source of information about the possible violation identification of the architectural desired structures by code smell investigation founded on metrics of the source code.

In the same context, there are also proposed solutions that address architectural degradation, which are less common than the solutions using the metrics-based detection strategy. These solutions are such as investigation and addressing of architectural rules violations, which provide the focus on the locations of violations within the system or place architecturally inappropriate files inside packages or classes, violating the planned architecture rules.

Regarding the priorities of architectural anomalies, it depicts a focus on the most influential anomalies in structure and more closely related to architectural problems rather than the anomalies that have no strong correlation with those problems.

Concerning an architectural recovery strategy solution, it describes the recovering the basic concepts of the planned architecture to conform to straightaway implemented architecture and to identify design decisions that may harm the basic rules of the intended architecture.

The refactoring strategy solutions represent the recommendations and guidelines for changing the structure and the behaviors of the internal system elements. Additionally, there are many other solutions proposed in addressing architectural degradation within the OSS environment.

The results also showed the classification of the proposed solution type in addressing decay that refers to addressing, identifying, reducing, avoiding, and predicting the architectural erosion, whereas several studies revolve around the idea of identifying architectural erosion more than the classification of other solution types (as highlighted in Fig. 13).

With respect to the proposed solutions to address architectural degradation, we have developed its taxonomy to clarify the overall stereotype to contribute to identifying the paths of these solutions and the extent of their facilitation to researchers in enhancing this taxonomy (as demonstrated in Fig. 14). This also reveals that the extent of the proposed solutions to address the decay within the OSS environment is still expanding and discovering other features of these solutions or integrating convergent solutions would provide better meaningful results. However, it does not mean that these proposed solutions have great effectiveness in dealing with degradation, and this is what we will get acquainted within the next section on the effectiveness of the proposed solutions.

D. THE EXTENT TO EFFECTIVENESS OF THE PROPOSED SOLUTIONS (RQ4)

As already discussed in the earlier research question (RQ3), the proposed solutions by the present researchers are based on their studies to address the architectural degradation and the extent of taxonomy identification according to the explicit strategies and convergence of the solutions. Therefore, in this section we explore more about the effectiveness of these solutions and the impact of their potential benefit in addressing architectural erosion within the open-source software community. The solutions are discussed based on the taxonomy stated in the prior research question (RQ3) as shown in Fig. 14.

1) METRICS- BASED DETECTION STRATEGY

We observed that the metrics strategy solutions were the most frequently used in identifying the architectural decay, thereby these metrics can determine the architectural instability growth with the system evolution, identify the probability of the classes contributing to architectural inconsistencies, and diagnose the anomalies, whether agglomerations or individual is more correlated to architectural problems. However, the use of current metrics at the class level may be affected by size bias significantly and inefficiency automatically in detecting architectural problems, indicating that the most likely cause is the problem on how these metrics are implemented through tools and reconsideration in specifying the selection of the appropriate metrics at different locations of software components, especially when compared to the same results that achieved efficiency manually.

In another context, AbdeenMod+RM metrics introduced an acceptable satisfaction in the fault prediction modeling. This satisfaction may increase in the implementation of a large number of alternative metrics and other techniques used in diverse aspects to present different results, which may reflect the overall recommendations of the research.

Modularity metrics refers to the significantly negative interrelationship by modularity indicators IPCI and IPGF. This may improve the performance and accuracy or less effort needed to predict by assuming a new modularity metrics at the system level and adapting current modularity indicators specified in other perspectives (e.g., complex networks). Furthermore, the architecture-sensitive metrics for code anomalies discovery provides the majority of awareness to engineers for the existence of the smells code elements that are more significant to the architecture design than the most traditional metrics that are depending on source code and based on static code metrics combination. This means that the developers and engineers could detect and repair such anomalies promptly.

Therefore, more studies are needed in this field for other metrics to be analyzed in order to provide the most appropriate architecture without any impact of the size bias. Furthermore, there is a need for metrics that have a great ability to discover the inconsistent classes affected by the degradation from the consistent classes. In addition, there is a need to identify the effort required for the metrics strategy to architecturally detect related anomalies and also to derive more metrics that may have an impact on the quality relationships of other software that are closely related to architectural problems.

2) PRIORITIZATION ARCHITECTURAL ANOMALIES

In the anomaly's prioritization strategy, the agglomeration flood standard and most optimal models showed that some agglomerations are overburden with false positives and not precise enough to identify architectural inconsistencies in classes, leading to the inability to capture several various architectural problem types. In contrast, the recommended heuristics, architecture blueprints, and the context-based smell prioritization techniques show the ability to rank and improve in identifying the prioritization of anomalies related to architectural problems. This reveals the need for providing an initial enrichment of the possible results to adopt the solution with a tendency in the ideal combination to prioritize architectural anomalies. Consequently, there is a need to provide various prioritization criteria for seizing the diverse architectural problem types and enhancing the essential techniques used for discovering code anomalies. Moreover, the integration of two or more heuristics would get better ranking results' effectiveness. Additionally, it is possible to introduce the new strategies to produce ranking on numerous criteria in order to provide visualization capabilities that are most relevant to architectural problems for the developers.

3) ARCHITECTURAL RULES VIOLATIONS

In terms of using the architectural rules violations solutions, we observed that many violations were not restricted by the architectural principles of the system, which may not have defined the necessary rules to reduce the severity of major violations. This means that violations still appear over again and over again despite the good violation solutions that were introduced and the approaches that have a significant role in capturing violations with thorough accuracy. Therefore, it is important to highlight the identification of the necessary rules and identification of critical cores through a broad study on architectural conformance using multiple frameworks.

4) REFACTORINGS STRATEGY

The refactoring strategy solutions do not contribute significantly in addressing the architectural degradation, as it showed many excesses in identifying architectural problems and is not positively affecting them, however, it may have a simple contribution in addressing architectural erosion when its problems are minor and easy to repair. This reveals that the quality of these solutions is not efficient at all in the problems of deep analysis and the difficulty of address.

5) ARCHITECTURAL RECOVERY STRATEGY

Based on the obtained findings, the architectural recovery strategy can accurately detect and recover the conceptual design decisions of the system, rather some techniques used in recovery architecture provide an acceptable improvement and may outperform each other as illustrated in the technique of layers based on structural and lexical in its noticeable superiority from the structural-based approaches. This reveals that architecture may be recovered with acceptable accuracy unlike the prior intuition based on its claimed hypothesis in an application difficulty to recover conceptual architecture. However, there is a clear opportunity for many researchers to shed light on checking more efficient approaches to architecture recovery by enriching ARCADE's tool to add further different current code-level analyses to it. Besides, there is also a need to conduct more experiments on a wide scope on more systems, especially industrial systems as well as to increase approach accuracy through documentations, pull requests, commit messages, comments, tests, and much more.

6) OTHER SOLUTIONS

Based on our observation, the other proposed solutions to address the degradation differ in terms of the performance effectiveness of the solutions and highlight a specific part to reduce erosion. Automatic detection for various approaches introduces an acceptable enhancement in the discovery of architectural smells despite providing various answers and excluding the external components that are notified as false negatives. An inherent complexity grants a great opportunity to increase the complexity and its development over time when it is not monitored and analyzed, leading to the multiplication of the change in other components in an unbalanced way. This reveals the extent of the leniency risk for not monitoring the complexity since its early evolutionary of components. Concerning the connected files to each other when bug-proneness and change-proneness constantly occur, it reflects negatively on the architectural influence. This reveals that the increasing rates of change and errors of these files inevitably lead to violate the basic design principles of the architecture. Consequently, reconsideration must be given to how architectural smells evolve and their profound relationship with problems from an architectural perspective, through existing tools that may need to be re-analyzed to cover a wide scope of those anomalies and reduce the current and future maintenance efforts for these systems.

VI. IMPLICATIONS FOR RESEARCH AND PRACTICE

An SLR study provides directions for researchers and practitioners on architectural decay within the OSS domain as follows:

1) There are reasons that could contribute excessively to increasing architectural erosion such as rapid development of software, frequent changes, and lack of developers' awareness. Therefore, further studies should be conducted as a rooted-deep study to find out other causes and to identify architecture erosion whether on the OSS or industrial systems. Practitioners should follow guidelines to avoid architectural contradictions in the new and subsequent versions of the system in terms of identifying the potential reasons within the system environment.
2) Since architectural degradation symptoms present that code smells agglomeration has a considerable correlation to architectural problems compared to code smells individual, researchers should conduct further investigation on architectural bad smells in combination. Practitioners can change their detection way depending on the code smells agglomeration to identify degradation symptoms effectively.
3) The findings of the current study serve as evident that a metrics-based strategy is the most commonly used solution as compared to other proposed solutions. However, more studies are needed in this field for other metrics to be analyzed to provide the most architecturally appropriate solution and identify the effort required for the metrics to detect architecturally related smells. The essential techniques of ranking used should enhance the possibility to get better effectiveness results and the identification of critical cores of architectural violations. Also, there is a clear opportunity for many researchers to highlight enriching ARCADE's tool for efficient approaches to architecture recovery. Additionally, there are solutions, which show that it is not effective at all in the problems of deep analysis and the difficulty of address such refactoring strategy that has no considerably a positive impact to address architectural erosion.

VII. THREATS TO VALIDITY OF THE STUDY

The major threats influencing the validity of SLR are associated with the direction that might have biased our systematic literature review. We identified highlighting search, selection of the relevant studies, and extraction of the adequate data for our investigation. The threats to validity are described [73] as follows:

1) Internal validity: The studies indicated [57], [74] that the common threat to SRL is to explore all researches and relevant studies to the specific research question. We attempted to maximize internal validity by selecting online appropriate databases that include enough relevant studies. Accordingly, eight well-known online databases were selected such as Scopus, Springer, Science Direct, Web of Science, Wiley, and Taylor and some of them specialized in our field of the area such as ACM, IEEE, covering all journal papers and

conferences. We also tried to maximize internal validity by identifying effective and sufficient search terms, including the synonyms, other alternative words, and abbreviations that identify and explore, from several recent scientific papers to adequately cover the specific research question. Additionally, the snowballing search strategy was applied by backward and forward investigation of the selected studies in order to explore the relevant papers that may be missed if the lack of a sufficient search term was considered.

2) Construct validity: We evaluated the quality of included studies in order to make sure that all potential articles were properly selected according to the specified criteria. We also tried to maximize the construct validity by excluding or including papers by following the guidelines proposed by Brereton *et al.* [75] and Kitchenham and Charters [57]. Additionally, we designed data extraction forms to collect the information, check the gathered data, and generate a checklist in order to investigate the required information, which was accomplished by conducting the analysis and discussion among the researchers to reduce the circle of difference and increase the accuracy of extracted information in line with the answer to the specific research question as properly as possible.

3) External validity: The study of software architecture in open-source software was conducted on 74 studies, including the articles obtained by the forward and backward strategies for the snowballing process, thus the study findings are generalizable to other studies that may pertain to architectural erosion in an open-source environment.

VIII. CONCLUSION

In this study, we conducted a Systematic Literature Review (SLR) to explore architectural degradation within the open-source software (OSS) community. The main goal was to systematically investigate and review the existing architectural erosion of the OSS to identify eroded architecture from diverse perspectives: the reasons that assist in the occurrence of architectural degradation, key indicators to initialize a permanence of the architectural degradation symptoms, the proposed solutions contributing to addressing the architectural decay, and the extent of the efficiency of these solutions. Numerous methods and criteria were used to include the relevant studies and exclude studies that were irrelevant to the research questions. A total of 74 primary studies were identified to analyze architectural erosion within the OSS community.

Our analysis indicated that 17 reasons contribute to architecture erosion within the OSS projects and the causes most commonly occurred are the rapid of software evolution, frequent changes, and the lack of developers' awareness. Simultaneously, we observed that four of the key indicators of architectural erosion symptoms were revealed within the

TABLE 19. The primary studies references.

S-ID	References
S1	Aversano, L., D. Guardabascio and M. Tortorella (2019). "An Empirical Study on the Architecture Instability of Software Projects." International Journal of Software Engineering and Knowledge Engineering **29**(4): 515-545.
S2	Cai, Y., L. Xiao, R. Kazman, R. Mo and Q. Feng (2019). "Design Rule Spaces: A New Model for Representing and Analyzing Software Architecture." IEEE Transactions on Software Engineering **45**(7): 657-682.
S3	Vidal, S., W. Oizumi, A. Garcia, A. Díaz Pace and C. Marcos (2019). "Ranking architecturally critical agglomerations of code smells." Science of Computer Programming **182**: 64-85
S4	Lenhard, J., M. Blom and S. Herold (2019). "Exploring the suitability of source code metrics for indicating architectural inconsistencies." Software Quality Journal **27**(1): 241-274
S5	Wang, T., D. Wang and B. Li (2019). A multilevel analysis method for architecture erosion. The 31st International Conference on Software Engineering and Knowledge Engineering.
S6	Silva, T. M., D. Serey, J. Figueiredo, Jo, #227 and o. Brunet (2019). Automated design tests to check Hibernate design recommendations. Proceedings of the XXXIII Brazilian Symposium on Software Engineering. Salvador, Brazil, ACM**:** 94-103.
S7	Lenarduzzi, V., N. Saarimaki and D. Taibi (2019). On the Diffuseness of Code Technical Debt in Java Projects of the Apache Ecosystem. 2019 IEEE/ACM International Conference on Technical Debt (TechDebt).
S8	Eposhi, A., W. Oizumi, A. Garcia, L. Sousa, R. Oliveira and A. Oliveira (2019). Removal of Design Problems through Refactorings: Are We Looking at the Right Symptoms? 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC).
S9	Ghorbani, N., J. Garcia and S. Malek (2019). Detection and repair of architectural inconsistencies in Java. Proceedings of the 41st International Conference on Software Engineering. Montreal, Quebec, Canada, IEEE Press**:** 560-571.
S10	Mo, R., Y. Cai, R. Kazman, L. Xiao and Q. Feng (2019). "Architecture Anti-patterns: Automatically Detectable Violations of Design Principles." IEEE Transactions on Software Engineering: 1-1.
S11	Fontana, F. A., P. Avgeriou, I. Pigazzini and R. Roveda (2019). A Study on Architectural Smells Prediction. 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA).
S12	Sas, D., P. Avgeriou and F. A. Fontana (2019). Investigating Instability Architectural Smells Evolution: An Exploratory Case Study. 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME).
S13	Fontana, F. A., I. Pigazzini, C. Raibulet, S. Basciano and R. Roveda (2019). PageRank and criticality of architectural smells. Proceedings of the 13th European Conference on Software Architecture - Volume 2. Paris, France, Association for Computing Machinery**:** 197–204.
S14	Feng, Q., Y. Cai, R. Kazman, D. Cui, T. Liu and H. Fang (2019). Active Hotspot: An Issue-Oriented Model to Monitor Software Evolution and Degradation. 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).
S15	Oizumi, W., L. Sousa, A. Oliveira, A. Garcia, A. B. Agbachi, R. Oliveira and C. Lucena (2018). "On the identification of design problems in stinky code: experiences and tool support." Journal of the Brazilian Computer Society **24**(1)
S16	Guimarães, E., S. Vidal, A. Garcia, J. A. Diaz Pace and C. Marcos (2018). "Exploring architecture blueprints for prioritizing critical code anomalies: Experiences and tool support." Software - Practice and Experience **48**(5): 1077-1106.

TABLE 19. ***(Continued.)*** **The primary studies references.**

S17	Sae-Lim, N., S. Hayashi and M. Saeki (2018). "Context-based approach to prioritize code smells for prefactoring." Journal of Software: Evolution and Process **30**(6): e1886.
S18	Carvalho, L. P. d. S., R. Novais, M. Mendon and #231 (2018). Investigating the Relationship between Code Smell Agglomerations and Architectural Concerns: Similarities and Dissimilarities from Distributed, Service-Oriented, and Mobile Systems. Proceedings of the VII Brazilian Symposium on Software Components, Architectures, and Reuse. Sao Carlos, Brazil, ACM**:** 3-12.
S19	Shahbazian, A., D. Nam and N. Medvidovic (2018). Toward Predicting Architectural Significance of Implementation Issues. 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR).
S20	Maisikeli, S. G. (2018). Measuring Architectural Stability and Instability in the Evolution of Software Systems. 2018 Fifth HCT Information Technology Trends (ITT).
S21	Díaz-Pace, J. A., A. Tommasel and D. Godoy (2018). Towards anticipation of architectural smells using link prediction techniques. 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM).
S22	Shahbazian, A., Y. K. Lee, D. Le, Y. Brun and N. Medvidovic (2018). Recovering Architectural Design Decisions. 2018 IEEE International Conference on Software Architecture (ICSA).
S23	Hayashi, S., F. Minami and M. Saeki (2018). "Detecting Architectural Violations Using Responsibility and Dependency Constraints of Components." IEICE Transactions on Information and Systems **E101.D**(7): 1780-1789.
S24	Palomba, F., G. Bavota, M. Di Penta, F. Fasano, R. Oliveto and A. De Lucia (2018). "A large-scale empirical study on the lifecycle of code smell co-occurrences." Information and Software Technology **99**: 1-10.
S25	Behnamghader, P., D. M. Le, J. Garcia, D. Link, A. Shahbazian and N. Medvidovic (2017). "A large-scale study of architectural evolution in open-source software systems." Empirical Software Engineering **22**(3): 1146-1193.
S26	Shaikh, M., A. Ansari, K. Memon, A. H. Jalbani and A. Ali (2017). "Evaluating Dependency based Package-level Metrics for Multi-objective Maintenance Tasks." International Journal of Advanced Computer Science and Applications **8**(10): 345-354.
S27	Pruijt, L., C. Köppe, J. M. van der Werf and S. Brinkkemper (2017). "The accuracy of dependency analysis in static architecture compliance checking." Software: Practice and Experience **47**(2): 273-309.
S28	Rocha, H., R. S. Durelli, R. Terra, S. Bessa and M. T. Valente (2017). "DCL 2.0: modular and reusable specification of architectural constraints." Journal of the Brazilian Computer Society **23**(1): 12.
S29	Fontana, F. A., I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni and E. D. Nitto (2017). Arcan: A Tool for Architectural Smells Detection. 2017 IEEE International Conference on Software Architecture Workshops (ICSAW).
S30	Olsson, T., M. Ericsson and A. Wingkvist (2017). Motivation and Impact of Modeling Erosion Using Static Architecture Conformance Checking. 2017 IEEE International Conference on Software Architecture Workshops (ICSAW).
S31	Lenhard, J., M. M. Hassan, M. Blom, S. Herold and Acm (2017). Are code smell detection tools suitable for detecting architecture degradation? Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings. Canterbury, United Kingdom, ACM**:** 138-144.
S32	Filho, J. L. M., L. Rocha, R. Andrade and R. Britto (2017). Preventing Erosion in Exception Handling Design Using Static-Architecture Conformance Checking. Software Architecture, Cham, Springer International Publishing.

TABLE 19. ***(Continued.)*** **The primary studies references.**

S33	Olsson, T., M. Ericsson and A. Wingkvist (2017). The relationship of code churn and architectural violations in the open source software JabRef. Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings. Canterbury, United Kingdom, ACM**:** 152-158.
S34	Fontana, F. A., I. Pigazzini, R. Roveda and M. Zanoni (2016). Automatic Detection of Instability Architectural Smells. 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME).
S35	Mo, R., Y. Cai, R. Kazman, L. Xiao and Q. Feng (2016). Decoupling Level: A New Metric for Architectural Maintenance Complexity. 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE).
S36	Maffort, C., M. T. Valente, R. Terra, M. Bigonha, N. Anquetil and A. Hora (2016). "Mining architectural violations from version history." Empirical Software Engineering **21**(3): 854-895.
S37	Belle, A. B., G. E. Boussaidi and S. Kpodjedo (2016). "Combining lexical and structural information to reconstruct software layers." Information and Software Technology **74**: 1-16.
S38	Xiao, L., Y. Cai, R. Kazman, R. Mo and Q. Feng (2016). Identifying and Quantifying Architectural Debt. 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE).
S39	Oizumi, W., A. Garcia, L. D. S. Sousa, B. Cafeo and Y. Zhao (2016). Code Anomalies Flock Together: Exploring Code Anomaly Agglomerations for Locating Design Problems. 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE).
S40	Mirakhorli, M. and J. Cleland-Huang (2016). "Detecting, Tracing, and Monitoring Architectural Tactics in Code." IEEE Transactions on Software Engineering **42**(3): 206-221.
S41	Fontana, F. A., R. Roveda, M. Zanoni, C. Raibulet and R. Capilla (2016). An Experience Report on Detecting and Repairing Software Architecture Erosion. 2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA).
S42	Arcelli Fontana, F., M. V. Mäntylä, M. Zanoni and A. Marino (2016). "Comparing and experimenting machine learning techniques for code smell detection." Empirical Software Engineering **21**(3): 1143-1191.
S43	Barros, M. D., F. D. Farzat and G. H. Travassos (2015). "Learning from optimization: A case study with Apache Ant." Information and Software Technology **57**: 684-704.
S44	Oizumi, W. N., A. F. Garcia, T. E. Colanzi, M. Ferreira and A. V. Staa (2015). "On the relationship of code-anomaly agglomerations and architectural problems." Journal of Software Engineering Research and Development **3**(1): 11.
S45	Ding, W., P. Liang, A. Tang and H. Van Vliet (2015). "Understanding the causes of architecture changes using OSS mailing lists." International Journal of Software Engineering and Knowledge Engineering **25**(9-10): 1633-1651.
S46	Terra, R., M. T. Valente, K. Czarnecki and R. S. Bigonha (2015). "A recommendation system for repairing violations detected by static architecture conformance checking." Software: Practice and Experience **45**(3): 315-342.
S47	Herold, S., M. English, J. Buckley, S. Counsell and M. Ó. Cinnéide (2015). Detection of violation causes in reflexion models. 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER).
S48	Guimarães, E., A. Garcia and Y. Cai (2015). Architecture-sensitive heuristics for prioritizing critical code anomalies. Proceedings of the 14th International Conference on Modularity. Fort Collins, CO, USA, Association for Computing Machinery**:** 68–80.
S49	Goldstein, M. and I. Segall (2015). Automatic and continuous software architecture validation. Proceedings of the 37th International Conference on Software Engineering - Volume 2. Florence, Italy, IEEE Press**:** 59-68.

TABLE 19. *(Continued.)* **The primary studies references.**

S50	Fontana, F. A., V. Ferme and M. Zanoni (2015). Towards Assessing Software Architecture Quality by Exploiting Code Smell Relations. 2015 IEEE/ACM 2nd International Workshop on Software Architecture and Metrics.
S51	Palomba, F., G. Bavota, M. D. Penta, R. Oliveto, D. Poshyvanyk and A. D. Lucia (2015). "Mining Version Histories for Detecting Code Smells." IEEE Transactions on Software Engineering **41**(5): 462-489.
S52	Ferreira, M., E. Barbosa, I. Macia, R. Arcoverde and A. Garcia (2014). Detecting architecturally-relevant code anomalies: a case study of effectiveness and effort. Proceedings of the 29th Annual ACM Symposium on Applied Computing. Gyeongju, Republic of Korea, Association for Computing Machinery**:** 1158–1163.
S53	Andrade, H. S. d., E. Almeida and I. Crnkovic (2014). Architectural bad smells in software product lines: an exploratory study. Proceedings of the WICSA 2014 Companion Volume. Sydney, Australia, Association for Computing Machinery**:** Article 12.
S54	Li, Z., P. Liang, P. Avgeriou, N. Guelfi and A. Ampatzoglou (2014). An empirical investigation of modularity metrics for indicating architectural technical debt. Proceedings of the 10th international ACM Sigsoft conference on Quality of software architectures. Marcq-en-Bareul, France, Association for Computing Machinery**:** 119–128.
S55	Herold, S. and M. Mair (2014). Recommending Refactorings to Re-establish Architectural Consistency. Software Architecture, Cham, Springer International Publishing.
S56	Gurgel, A., I. Macia, A. Garcia, A. v. Staa, M. Mezini, M. Eichberg and R. Mitschke (2014). Blending and reusing rules for architectural degradation prevention. Proceedings of the 13th international conference on Modularity. Lugano, Switzerland, Association for Computing Machinery**:** 61–72.
S57	Macia, I., A. Garcia, C. Chavez and A. v. Staa (2013). Enhancing the Detection of Code Anomalies with Architecture-Sensitive Strategies. 2013 17th European Conference on Software Maintenance and Reengineering.
S58	Arcoverde, R., E. Guimarães, I. Macía, A. Garcia and Y. Cai (2013). Prioritization of Code Anomalies Based on Architecture Sensitiveness. 2013 27th Brazilian Symposium on Software Engineering.
S59	Garcia, J., I. Krka, C. Mattmann and N. Medvidovic (2013). Obtaining ground-truth software architectures. 2013 35th International Conference on Software Engineering (ICSE).
S60	Hassaine, S., Y. G. Guéhéneuc, S. Hamel and G. Antoniol (2012). ADvISE: Architectural Decay In Software Evolution. 2012 16th European Conference on Software Maintenance and Reengineering. T. Mens, A. Cleve and R. Ferenc**:** 267-276.
S61	Macia, I., J. Garcia, D. Popescu, A. Garcia, N. Medvidovic and A. v. Staa (2012). Are automatically-detected code anomalies relevant to architectural modularity? an exploratory analysis of evolving systems. Proceedings of the 11th annual international conference on Aspect-oriented Software Development. Potsdam, Germany, Association for Computing Machinery**:** 167–178.
S62	Macia, I., R. Arcoverde, A. Garcia, C. Chavez and A. v. Staa (2012). On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms. 2012 16th European Conference on Software Maintenance and Reengineering.
S63	Brunet, J., R. A. Bittencourt, D. Serey and J. Figueiredo (2012). On the Evolutionary Nature of Architectural Violations. 2012 19th Working Conference on Reverse Engineering.
S64	Arcelli Fontana, F., P. Braione and M. Zanoni (2012). "Automatic detection of bad smells in code: An experimental assessment." Journal of Object Technology **11**.
S65	Li, Z., N. H. Madhavji, S. S. Murtaza, M. Gittens, A. V. Miranskyy, D. Godwin and E. Cialini (2011). "Characteristics of multiple-component defects and architectural hotspots: a large system case study." Empirical Software Engineering **16**(5): 667-702.

TABLE 19. *(Continued.)* **The primary studies references.**

S66	Steff, M. and B. Russo (2011). Measuring Architectural Change for Defect Estimation and Localization. 2011 International Symposium on Empirical Software Engineering and Measurement.
S67	Wong, S., Y. Cai, M. Kim and M. Dalton (2011). Detecting software modularity violations. 2011 33rd International Conference on Software Engineering (ICSE).
S68	Sangwan, R. S., P. Vercellone-Smith and C. J. Neill (2010). "Use of a multidimensional approach to study the evolution of software complexity." Innovations in Systems and Software Engineering **6**(4): 299-310.
S69	Moha, N., Y. Gueheneuc, L. Duchien and A. L. Meur (2010). "DECOR: A Method for the Specification and Detection of Code and Design Smells." IEEE Transactions on Software Engineering **36**(1): 20-36.
S70	Biehl, M. and W. Löwe (2009). Automated Architecture Consistency Checking for Model Driven Software Development. Architectures for Adaptive Software Systems, Berlin, Heidelberg, Springer Berlin Heidelberg.
S71	Capiluppi, A. and T. Knowles (2009). Software Engineering in Practice: Design and Architectures of FLOSS Systems. Open Source Ecosystems: Diverse Communities Interacting, Berlin, Heidelberg, Springer Berlin Heidelberg.
S72	Sarkar, S., G. Maskeri and S. Ramachandran (2009). "Discovery of architectural layers and measurement of layering violations in source code." Journal of Systems and Software **82**(11): 1891-1905.
S73	Medvidovic, N. and V. Jakobac (2006). "Using software evolution to focus architectural recovery." Automated Software Engineering **13**(2): 225-256.
S74	Tran, J. B., M. W. Godfrey, E. H. S. Lee and R. C. Holt (2000). Architectural repair of open source software. Proceedings IWPC 2000. 8th International Workshop on Program Comprehension.

OSS projects, which are code smells, architectural smells, the architectural constraint's violation, and architectural technical debts. Our analysis also revealed the proposed solutions addressed degradation, which were categorized. They showed that most solutions were based on the metrics-based strategy. Other solutions are such as anomalies prioritization strategy, addressing and investigating architectural rules violations, refactoring strategy, and architectural recovery strategy, which were also identified and detected in terms of the extent of the effectiveness and accuracy of these proposed solutions as well as their contribution to identify, address or predict the architectural degradation within OSS projects.

It can be concluded that the problems of architectural erosion within the OSS projects, including identifying, addressing, avoiding and predicting are still open research issues, which need further analysis and investigation. Consequently, more efforts on this domain should be focused on identifying the other reasons that are still unclear and suggesting other solutions to provide more performance and accuracy to address architectural decay.

APPENDIX A PRIMARY STUDIES REFERENCES

See Table 19.

TABLE 20. Quality assessment criterion.

S-ID	QA1	QA2	QA3	QA4	QA5	Score
S1	1	1	1	0.5	0.5	4
S2	1	1	1	1	1	5
S3	1	1	1	1	1	5
S4	1	1	1	1	1	5
S5	0.5	1	0.5	0.5	0.5	3
S6	1	1	1	1	1	5
S7	1	1	1	1	1	5
S8	1	1	1	1	1	5
S9	1	1	1	1	1	5
S10	1	1	1	1	1	5
S11	0.5	1	1	0.5	1	4
S12	1	1	1	0	1	4
S13	0.5	1	1	0.5	1	4
S14	1	1	1	1	1	5
S15	1	1	1	1	1	5
S16	1	1	1	0.5	1	4.5
S17	1	1	1	1	1	5
S18	1	1	1	1	1	5
S19	1	1	1	0.5.	1	4.5
S20	1	1	1	0.5	1	4.5
S21	1	1	1	1	1	5
S22	1	1	1	1	1	5
S23	1	1	0.5	1	1	4.5
S24	1	1	1	1	1	5
S25	1	1	1	1	1	5
S26	1	1	1	1	1	5
S27	1	1	0.5	0	1	3.5
S28	1	1	1	1	1	5
S29	0.5	0.5	0.5	0.5	1	3
S30	1	0.5	0.5	0.5	1	3.5
S31	1	1	1	1	1	5
S32	0	1	1	1	1	4
S33	1	1	1	1	1	5
S34	1	1	1	0.5	1	4.5
S35	1	1	1	1	1	5
S36	1	1	1	1	1	5
S37	1	1	1	1	1	5
S38	1	1	1	1	1	5
S39	0.5	1	1	1	1	4.5
S40	1	1	1	1	1	5
S41	0.5	1	0.5	0	1	3
S42	1	1	1	1	1	5
S43	1	1	1	1	1	5
S44	1	1	1	0.5	1	4.5
S45	1	1	1	0.5	1	4.5
S46	1	1	1	1	1	5
S47	0.5	0.5	0.5	1	1	3.5
S48	1	1	1	1	1	5
S49	0.5	0.5	0.5	0.5	1	3
S50	0.5	1	0.5	0.5	1	3.5
S51	1	1	1	1	1	5
S52	1	0.5	0.5	1	1	4
S53	1	0.5	0.5	0.5	1	3.5
S54	1	1	1	1	1	5
S55	0.5	0.5	0.5	0	1	2.5
S56	0.5	1	1	1	1	4.5
S57	1	1	1	1	1	5
S58	1	1	1	1	1	5
S59	1	1	0.5	0	1	3.5
S60	1	1	1	1	1	5
S61	1	1	1	0	1	4
S62	1	1	1	1	1	5
S63	0.5	1	1	0.5	1	4
S64	1	1	1	1	1	5
S65	1	1	1	1	1	5
S66	1	1	1	1	1	5
S67	0.5	1	1	1	1	4.5
S68	1	1	1	1	1	5
S69	0.5	1	1	1	1	4.5
S70	0.5	1	1	0.5	0.5	3.5
S71	1	1	0.5	1	0.5	4
S72	0.5	1	1	1	1	4.5
S73	1	1	0.5	0.5	0.5	3.5
S74	1	1	0.5	0.5	0.5	3.5

APPENDIX B
QUALITY ASSESSMENT CRITERION

See Table 20.

ACKNOWLEDGMENT

The authors would like to acknowledge Universiti Putra Malaysia (UPM) for providing support on part works and IEEE Publisher for supporting the publication fees. The first author would also like to express his gratitude to Hadhramout University and the Hadhramout Foundation, Yemen, for their support in tuition fees.

REFERENCES

[1] R. Kitchin and M. Dodge, *Code/Space: Software and EverydayLife*. Cambridge, MA, USA: MIT Press, 2011.

[2] V. G. Cerf, "A brittle and fragile future," *Commun. ACM*, vol. 60, no. 7, p. 7, Jun. 2017.

[3] P. Naur, B. Randell, and F. L. Bauer, *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968*. Brussels, Belgium: Scientific Affairs Division, NATO, 1969.

[4] J. Bosch and P. Molin, "Software architecture design: Evaluation and transformation," in *Proc. IEEE Conf. Workshop Eng. Computer-Based Syst. ECBS*, Mar. 1999, pp. 4–10.

[5] P. Clements, D. Garlan, R. Little, R. Nord, and J. Stafford, "Documenting software architectures: Views and beyond," in *Proc. 25th Int. Conf. Softw. Eng.*, May 2003, pp. 740–741.

[6] M. Shaw and P. Clements, "The golden age of software architecture," *IEEE Softw.*, vol. 23, no. 2, pp. 31–39, Mar. 2006.

[7] D. Garlan, "Software architecture: A roadmap," in *Proc. 22nd Int. Conf. Softw. Eng., Future Softw. Eng. Track*, Oct. 2000, pp. 91–101.

[8] L. Dobrica and E. Niemela, "A survey on software architecture analysis methods," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 638–653, Jul. 2002.

[9] R. N. Taylor, N. Medvidovic, and E. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Hoboken, NJ, USA: Wiley, 2009.

[10] H. V. Vliet, *Software Engineering: Principles and Practice*. Hoboken, NJ, USA: Wiley, 2008.

[11] D. L. Parnas, "Software aging," presented at the Proc. 16th Int. Conf. Softw. Eng., Sorrento, Italy, May 1994.

[12] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? Assessing the evidence from change management data," *IEEE Trans. Softw. Eng.*, vol. 27, no. 1, pp. 1–12, Jan. 2001.

[13] A. MacCormack, J. Rusnak, and C. Y. Baldwin, "Exploring the structure of complex software designs: An empirical study of open source and proprietary code," *Manage. Sci.*, vol. 52, no. 7, pp. 1015–1030, Jul. 2006.

[14] M. Godfrey and E. Lee, "Secrets from the monster: Extracting Mozilla's software architecture," in *Proc. 2nd Int. Symp. Constructing Softw. Eng. Tools (CoSET)*, May 2000, pp. 1–9.

[15] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Softw. Eng. Notes*, vol. 17, no. 4, pp. 40–52, Oct. 1992.

[16] J. Garcia, I. Ivkovic, and N. Medvidovic, "A comparative analysis of software architecture recovery techniques," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2013, pp. 486–496.

[17] R. Grewal, G. L. Lilien, and G. Mallapragada, "Location, location, location: How network embeddedness affects project success in open source systems," *Manage. Sci.*, vol. 52, no. 7, pp. 1043–1056, Jul. 2006.

[18] S. D. Suh and I. Neamtiu, "Studying software evolution for taming software complexity," in *Proc. 21st Austral. Softw. Eng. Conf.*, Apr. 2010, pp. 3–12.

[19] 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," *Softw., Pract. Exper.*, vol. 49, no. 1, pp. 3–39, Jan. 2019.

[20] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Inf. Softw. Technol.*, vol. 55, no. 7, pp. 1165–1199, Jul. 2013.

[21] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: A review of current knowledge," *J. Softw. Maintenance Evol., Res. Pract.*, vol. 23, no. 3, pp. 179–202, Apr. 2011.

[22] F. Bachmann, L. Bass, M. Klein, and C. Shelton, "Designing software architectures to achieve quality attribute requirements," *IEE Proc. Softw.*, vol. 152, no. 4, pp. 153–165, 2005.

[23] L. Hochstein and M. Lindvall, "Combating architectural degeneration: A survey," *Inf. Softw. Technol.*, vol. 47, no. 10, pp. 643–656, Jul. 2005.

[24] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Reading, MA, USA: Addison-Wesley, 2012.

[25] R. G. Crispen and L. D. Stuckey, "Structural model: Architecture for software designers," presented at the Proc. Conf. TRI-Ada, Baltimore, MA, USA, Nov. 1994, doi: 10.1145/197694.197729.

[26] M. M. Lehman, "On understanding laws, evolution, and conservation in the large-program life cycle," *J. Syst. Softw.*, vol. 1, pp. 213–221, Jan. 1979.

[27] C. Stringfellow, C. D. Amory, D. Potnuri, A. Andrews, and M. Georg, "Comparison of software architecture reverse engineering methods," *Inf. Softw. Technol.*, vol. 48, no. 7, pp. 484–497, Jul. 2006.

[28] Z. Li and J. Long, "A case study of measuring degeneration of software architectures from a defect perspective," in *Proc. 18th Asia–Pacific Softw. Eng. Conf.*, Dec. 2011, pp. 242–249.

[29] M. Lindvall, R. Tesoriero, and P. Costa, "Avoiding architectural degeneration: An evaluation process for software architecture," in *Proc. 8th IEEE Symp. Softw. Metrics*, Jun. 2002, pp. 77–86.

[30] J. van Gurp and J. Bosch, "Design erosion: Problems and causes," *J. Syst. Softw.*, vol. 61, no. 2, pp. 105–119, Mar. 2002.

[31] N. Medvidovic, A. Egyed, and P. Grünbacher, "Stemming architectural erosion by coupling architectural discovery and recovery," in *Proc. 2nd Int. Softw. Requirements Archit. Workshop*, Apr. 2003, p. 61.

[32] A. Jansen, J. van der Ven, P. Avgeriou, and D. K. Hammer, "Tool support for architectural decisions," in *Proc. Work. IEEE/IFIP Conf. Softw. Archit. (WICSA)*, Jan. 2007, p. 4.

[33] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural mismatch: Why reuse is so hard," *IEEE Softw.*, vol. 12, no. 6, pp. 17–26, Nov. 1995.

[34] M. Riaz, M. Sulayman, and H. Naqvi, "Architectural decay during continuous software evolution and impact of 'design for change' on software architecture," in *Proc. Int. Conf. Adv. Softw. Eng. Appl.*, 2009, pp. 119–126.

[35] C. Izurieta and J. M. Bieman, "How software designs decay: A pilot study of pattern evolution," in *Proc. 1st Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, Sep. 2007, pp. 449–451.

[36] I. Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*. Reading, MA, USA: Addison-Wesley, 2004.

[37] J. Bosch, "Software architecture: The next step," in *Software Architecture*. Berlin, Germany: Springer, 2004, pp. 194–199.

[38] J. Lenhard, M. Blom, and S. Herold, "Exploring the suitability of source code metrics for indicating architectural inconsistencies," *Softw. Qual. J.*, vol. 27, no. 1, pp. 241–274, Mar. 2019.

[39] M. Lindvall and D. Muthig, "Bridging the software architecture gap," *Computer*, vol. 41, no. 6, pp. 98–101, Jun. 2008.

[40] X. Dong and M. W. Godfrey, "Identifying architectural change patterns in object-oriented systems," in *Proc. 16th IEEE Int. Conf. Program Comprehension*, Jun. 2008, pp. 33–42.

[41] I. Macia, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic, and A. V. Staa, "Are automatically-detected code anomalies relevant to architectural modularity? An exploratory analysis of evolving systems," presented at the Proc. 11th Annu. Int. Conf. Aspect-oriented Softw. Develop., Potsdam, Germany, 2012, doi: 10.1145/2162049.2162069.

[42] I. M. Bertran, A. Garcia, and A. V. Staa, "An exploratory study of code smells in evolving aspect-oriented systems," presented at the Proc. 10th Int. Conf. Aspect-Oriented Softw. Develop., Porto de Galinhas, Brazil, 2011, doi: 10.1145/1960275.1960300.

[43] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. von Staa, "On the relevance of code anomalies for identifying architecture degradation symptoms," in *Proc. 16th Eur. Conf. Softw. Maintenance Reeng.*, Mar. 2012, pp. 277–286.

[44] K. A. Dawood, K. Y. Sharif, A. A. Zaidan, A. A. Abd Ghani, H. B. Zulzalil, and B. B. Zaidan, "Mapping and analysis of open source software (OSS) usability for sustainable OSS product," *IEEE Access*, vol. 7, pp. 65913–65933, 2019.

[45] J. W. Paulson, G. Succi, and A. Eberlein, "An empirical study of open-source and closed-source software products," *IEEE Trans. Softw. Eng.*, vol. 30, no. 4, pp. 246–256, Apr. 2004.

[46] E. V. Hippel and G. V. Krogh, "Open source software and the 'private-collective' innovation model: Issues for organization science," *Org. Sci.*, vol. 14, no. 2, pp. 209–223, 2003.

[47] A. Mockus, R. T. Fielding, and J. Herbsleb, "A case study of open source software development: The apache server," in *Proc. 22nd Int. Conf. Softw. Eng. ICSE*, 2000, pp. 263–272.

[48] T. Dinh-Trong and J. M. Bieman, "Open source software development: A case study of FreeBSD," in *Proc. 10th Int. Symp. Softw. Metrics*, 2004, pp. 96–105.

[49] P. L. Li, J. Herbsleb, and M. Shaw, "Finding predictors of field defects for open source software systems in commonly available data sources: A case study of OpenBSD," presented at the Proc. 11th IEEE Int. Softw. Metrics Symp., Sep. 2005, doi: 10.1109/METRICS.2005.26.

[50] S. Spaeth, M. Stuermer, S. Haefliger, and G. von Krogh, "Sampling in open source software development: The case for using the debian GNU/Linux distribution," in *Proc. 40th Annu. Hawaii Int. Conf. Syst. Sci. (HICSS)*, Jan. 2007, p. 166.

[51] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and Mozilla," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 3, pp. 309–346, 2002.

[52] G. von Krogh, S. Spaeth, and S. Haefliger, "Knowledge reuse in open source software: An exploratory study of 15 open source projects," in *Proc. 38th Annu. Hawaii Int. Conf. Syst. Sci.*, Jan. 2005, p. 198.

[53] A. Capiluppi and J. F. Ramil, "Studying the evolution of open source systems at different levels of granularity: Two case studies," in *Proc. 7th Int. Workshop Princ. Softw. Evol.*, Sep. 2004, pp. 113–118.

[54] S. Christley and G. Madey, "Analysis of activity in the open source software development community," in *Proc. 40th Annu. Hawaii Int. Conf. Syst. Sci. (HICSS)*, Jan. 2007, p. 166.

[55] F. Zou and J. Davis, "A model of bug dynamics for open source software," in *Proc. 2nd Int. Conf. Secure Syst. Integr. Rel. Improvement*, Jul. 2008, pp. 185–186.

[56] A. Fuggetta, "Open source software—An evaluation," *J. Syst. Softw.*, vol. 66, no. 1, pp. 77–90, Apr. 2003.

[57] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," EBSE, Goyang-si, South Korea, Tech Rep. EBSE Ver. 2.3, 2007.

[58] B. Kitchenham, O. Pearl Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering—A systematic literature review," *Inf. Softw. Technol.*, vol. 51, no. 1, pp. 7–15, Jan. 2009.

[59] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, "Model-based performance prediction in software development: A survey," *IEEE Trans. Softw. Eng.*, vol. 30, no. 5, pp. 295–310, May 2004.

[60] F. Brosig, P. Meier, S. Becker, A. Koziolek, H. Koziolek, and S. Kounev, "Quantitative evaluation of model-driven performance analysis and simulation of component-based architectures," *IEEE Trans. Softw. Eng.*, vol. 41, no. 2, pp. 157–175, Feb. 2015.

[61] H. P. Breivold and I. Crnkovic, "A systematic review on architecting for software evolvability," in *Proc. 21st Austral. Softw. Eng. Conf.*, Apr. 2010, pp. 13–22.

[62] W. J. Dzidek, E. Arisholm, and L. C. Briand, "A realistic empirical evaluation of the costs and benefits of UML in software maintenance," *IEEE Trans. Softw. Eng.*, vol. 34, no. 3, pp. 407–432, May 2008.

[63] R. Britto, D. Smite, and L.-O. Damm, "Software architects in large-scale distributed projects: An ericsson case study," *IEEE Softw.*, vol. 33, no. 6, pp. 48–55, Nov. 2016.

[64] H. Koziolek, B. Schlich, S. Becker, and M. Hauck, "Performance and reliability prediction for evolving service-oriented software systems," *Empirical Softw. Eng.*, vol. 18, no. 4, pp. 746–790, Aug. 2013.

[65] B. Kitchenham, "Procedures for performing systematic reviews," Dept. Comput. Sci., Keele Univ., Keele, U.K., Tech. Rep. UKTR/SE-0401, 2004.

[66] J. Webster and R. T. Watson, "Analyzing the past to prepare for the future: Writing a literature review," *MIS Quart.*, vol. 26, no. 2, pp. 13–23, 2002.

[67] O. Dieste and A. G. Padua, "Developing search strategies for detecting relevant experiments for systematic reviews," in *Proc. 1st Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, Sep. 2007, pp. 215–224.

[68] F. Arcelli Fontana, V. Lenarduzzi, R. Roveda, and D. Taibi, "Are architectural smells independent from code smells? An empirical study," *J. Syst. Softw.*, vol. 154, pp. 139–156, Aug. 2019.

[69] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure it? Manage it? Ignore it? Software practitioners and technical debt," presented at the Proc. 10th Joint Meeting Found. Softw. Eng., Bergamo, Italy, Aug. 2015, doi: 10.1145/2786805.2786848.

[70] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia, ''On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation,'' *Empirical Softw. Eng.*, vol. 23, no. 3, pp. 1188–1221, Jun. 2018.

[71] T. Hall, M. Zhang, D. Bowes, and Y. Sun, ''Some code smells have a significant but small effect on faults,'' *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, pp. 1–39, Sep. 2014.

[72] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, ''The evolution and impact of code smells: A case study of two open source systems,'' in *Proc. 3rd Int. Symp. Empirical Softw. Eng. Meas.*, Oct. 2009, pp. 390–400.

[73] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln, *Experimentation in Software Engineering*. Berlin, Germany: Springer-Verlag, 2012.

[74] A. Nguyen-Duc, D. S. Cruzes, and R. Conradi, ''The impact of global dispersion on coordination, team performance and software quality—A systematic literature review,'' *Inf. Softw. Technol.*, vol. 57, pp. 277–294, Jan. 2015.

[75] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil, ''Lessons from applying the systematic literature review process within the software engineering domain,'' *J. Syst. Softw.*, vol. 80, no. 4, pp. 571–583, Apr. 2007.

AHMED BAABAD received the B.Sc. degree in computer science from Hadhramout University, Yemen, and the M.Sc. degree in information systems from Osmania University, India. He is currently pursuing the Ph.D. degree with the Department of Software Engineering, Faculty of Computer Science and Information Technology, Universiti Putra Malaysia (UPM). He is also a Lecturer with Hadhramout University. His research interests include software metrics, software quality, and software architecture.

HAZURA BINTI ZULZALIL received the B.Sc. degree in computer science, the M.Sc. degree in software engineering, and the Ph.D. degree in software engineering from Universiti Putra Malaysia (UPM). She is currently an Associate Professor with the Faculty of Computer Science and Information Technology, UPM. Her research interests include software metrics, software quality, and software engineering.

SA'ADAH HASSAN received the B.Comp.Sc. degree from Universiti Teknologi Malaysia, the master's degree in software engineering from the University of Malaya, and the Ph.D. degree from Ulster University, U.K. She is currently an Associate Professor with Universiti Putra Malaysia. Her research interests include software engineering, intelligent systems, and management information systems.

SALMI BINTI BAHAROM received the B.Comp.Sc. degree from Universiti Putra Malaysia (UPM), and the M.Sc. and Ph.D. degrees in software engineering from the Universiti Kebangsaan Malaysia (UKM). She is currently an Associate Professor with UPM. Her research interests include software testing, software quality, and software engineering.