The Journal of Systems & Software 194 (2022) 111505

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

A survey of software architectural change detection and categorization techniques☆

Amit Kumar Mondal *, Kevin A. Schneider, Banani Roy, Chanchal K. Roy

University of Saskatchewan, Canada

ARTICLE INFO

Article history:
Received 25 January 2022
Received in revised form 5 July 2022
Accepted 3 September 2022
Available online 10 September 2022

Keywords:
Software architecture
Change detection
Classification
Design review
Abstraction

ABSTRACT

Software architecture is defined as the structural construction, design decisions implementation, evolution and knowledge sharing mechanisms of a system. Software architecture documentation help architects with decision making, guide developers during implementation, and preserve architectural decisions so that future caretakers are able to better understand an architect's solution. Many modern-day software development teams are focusing more on architectural consistency of software design to better cope with the cost-time-efforts, continuous integration, software glitches, security backdoors, regulatory inspections, human values, and so on. Therefore, in order to better reflect the software design challenges, the development teams review the architectural design either on a regular basis or after completing certain milestones or releases. However, many studies have focused on architectural change detection and classification as the essential steps for reviewing design, discovering architectural tactics and knowledge, analyzing software stability, tracing and auditing software development history, recovering design decisions, generating design summary, and so on.

In this paper, we survey state-of-the-art architectural change detection and categorization techniques and identify future research directions. To the best of our knowledge, our survey is the first comprehensive report on this area. However, in this survey, we compare available techniques using various quality attributes relevant to software architecture for different implementation levels and types. Moreover, our analysis shows that there is a lack of lightweight techniques (in terms of human intervention, algorithmic complexity, and frequency of usage) feasible to process hundreds and thousands of change revisions of a project. We also realize that rigorous focuses are required for capturing the design decision associativity of the architectural change detection techniques for practical use in the design review process. However, our survey on architectural change classification shows that existing automatic change classification techniques are not promising enough to use for real-world scenarios and reliable post analysis of causes of architectural change is not possible without manual intervention. There is also a lack of empirical data to construct an architectural change taxonomy, and further exploration in this direction would add much value to architectural change management.

1. Introduction

Our daily life is becoming more dependent on various software applications and autonomous systems, starting from transportation[1] to healthcare services[2] (Durisic et al., 2011; Khan et al., 2008). However, in recent years, many are being severely impacted due to software bugs (such as hundreds of people have died in a plane crash caused by software glitch[3]), and security vulnerabilities of software services (such as the ransomware attack on energy sector[4]). Besides, regulatory inspections by the government authority of software internal functions are also increasing due to various concerns such as hidden unethical business practices, data privacy and human values (racial bias) (Kosenkov et al., 2021; Nurwidyantoro et al., 2021). These are some of the external challenges of a software project. Sometimes, the software industries require extra time and manpower (cost) for regular development activities, late-life-cycle change

☆ Editor: Alexander Chatzigeorgiou.

* Corresponding author.
E-mail addresses: amit.mondal@usask.ca (A.K. Mondal), kevin.schneider@usask.ca (K.A. Schneider), banani.roy@usask.ca (B. Roy), chanchal.roy@usask.ca (C.K. Roy).

1 waymo.com/waymo-driver.

2 www.objectivity.co.uk/blog/top-10-types-of-healthcare-software.

3 www.businessinsider.com/boeing-software-errors-jeopardized-starliner-spaceship-737-max-planes-2020-2.

4 www.cnn.com/2021/08/16/tech/colonial-pipeline-ransomware/index.html.

https://doi.org/10.1016/j.jss.2022.111505
0164-1212/

and continuous integration glitches[5] to cope with the rapidly evolving technologies and requirements (Carriere et al., 2010; Alves et al., 2016). These are the internal challenges of a software project. To better cope with the aforementioned external and internal challenges, the development and maintenance (DEVEM) teams spend significant efforts on design consistency through code comprehension, code review, and software documentation (Aghajani et al., 2020; Cornelissen et al., 2009; Zanjani et al., 2016).

Design consistency is crucial for both commercial and open-source software (OSS) projects. Because, apart from the DEVEM challenges, practitioners and researchers are reporting incremental performance and security risks[6] within the architectural component's inter and intra dependency relations (Zhang et al., 2015; Manadhata and Wing, 2011; Wang et al., 2020; Ghorbani et al., 2019), elevating the design concerns. In addition to risk reduction, proper design increases developer productivity (i.e., improved maintainability, portability, and flexibility) (Carriere et al., 2010; Roshandel et al., 2004). Therefore, in order to better reflect the software design challenges, the development teams review the architectural design change (probably with the motto of *"Prevention is better than cure"*) either on a regular basis or after completing certain milestones or releases (Tang and Lau, 2014). That said, software design concerns are captured through software architecture.

Software architecture is *a high-level representation that defines the major structure and interactions of the internal components of a system and the interactions between the system and its environment* (Gustafsson et al., 2002). Researchers and practitioners define software architecture as the structural construction (Clements et al., 2003), design decisions implementation (Taylor et al., 2009), evolution (IEEE-1471) and knowledge sharing (Zwinkau, 2019) mechanisms of a system. These definitions indicate that architectural design change has far reaching consequences that are needed to measure during or after implementation.

Software architecture can be changed during regular DEVEM activities — new feature addition, bug fixing, refactoring, etc. Williams and Carver linked the causes of architectural changes to four Lehman's law of software changes (Lehman, 1996) — continuing software change, increasing complexity, continuing growth, and declining quality. A typical software change may contain local code change or architectural change or both. However, compared to local code changes, design impactful (or architectural) changes are involved in the wider spectrum of code components, and dependency among multiple modules/components (Schmitt Laser et al., 2020) despite focusing on a single issue in such changes. As a result, comprehending their scopes and impacts are more complex to the reviewer (Tang and Lau, 2014; Uchôa et al., 2021), and elevate the change and maintenance cost and effort across a system's lifecycle (Williams and Carver, 2014). Thus, understanding and updating a system's architecture in elegant ways is crucial for DEVEM (Garcia et al., 2021). In this regard, architectural change management process helps predict what must be changed, facilitates context for reasoning about, specifying, and implementing change, and preserves consistency between system design and adaptive as well as evolutionary changes (Oreizy et al., 1998; Monschein et al., 2021).

However, for architectural change management, development teams categorize the changes based on different criteria, such as the cause of the change, the concept of concerns/features, the location of the change, the size of the code modification, or the potential impact of the change (Fluri and Gall, 2006; Hindle et al., 2008; Dragan et al., 2011). For example, causes of architectural changes are (Williams and Carver, 2010; Ding et al., 2015): *perfective* — indicates new requirements and improved functionality, *corrective* — addresses flaws, *adaptive* — occurs for new environment or for imposing new policies, and *preventative* — indicates restructuring or redesigning the system. Different categories trigger different strategies for change management.

There are many implicit implications of architecture change detection and categorization (ACDC) in software evolution support, maintenance support, and fault detection and change propagation (Williams and Carver, 2010; Rasool and Fazal, 2017; Jamshidi et al., 2013). Instead, we discuss some explicit contexts (and mostly regular supportive DEVEM activities) where ACDC are essentials.

- ***Design Review***: A typical software design review procedure consists of extraction of requirements, extraction of design and its change, construction of causal relationships, the discovery of potential design issues, etc. (Tang and Lau, 2014). Proper change information extraction helps to execute these procedures. However, 45% rejected pull requests in OSS projects contain design inconsistency (Silva et al., 2016). Moreover, software projects need to avoid degradation as erosion, drifts as well as architecture pendency (Jamshidi et al., 2013). To help eradicate these challenges, architectural change detection is required to check the differences between the proposed and implemented design and whether it complies with the design guidelines (Tang and Lau, 2014).
- ***Design Document Generation:*** More than 60% developers and 85% project managers are likely to use architecture/design documents (Buse and Zimmermann, 2012; Bachmann et al., 2010). Furthermore, more than 40% major release notes and 15% minor release notes contains design changes (Bi et al., 2020). In such software documents, change category (i.e., new feature addition, restructuring, etc.) is the fundamental information. There are also pieces of evidence that design change logs (even with the releases) are also maintained for the DEVEM teams by both the industrial and OSS projects.[7] That said, change detection and categorization are mandatory for various software change document generation.
- ***Architecture and Design Decision Recovery***: Many projects do not document architecture and design decision associativity with the components in the initial phases. But later detection and categorizing of the change revisions are essential to recovering and documenting the software architecture, and design decisions (Jansen and Bosch, 2005; Garcia et al., 2011; Shahbazian et al., 2018a).
- ***Change Tracing and Change Impact Analysis***: Change detection is required for change impact analysis (Arvanitou et al., 2017; Williams and Carver, 2010). Design decisions (i.e., flaw fixing and new features) and corresponding changes can only be traced through detection and categorizing techniques (Shahbazian et al., 2018a; Bhat et al., 2020; Hammad et al., 2009). Moreover, architecture change tracing would facilitate on-demand artifacts extraction for code comprehension and other decision-making purposes because architecture is considered the primary artifact to trace other artifacts (Bi et al., 2018).

[5] techcrunch.com/2021/10/04/facebook-messenger-instagram-whatsapp-are-all-down.

[6] github.com/advisories/GHSA-jfh8-c2jp-5v3q.

[7] search.maven.org/remotecontent?filepath=com/azure/resourcemanager/azure-resourcemanager-avs/1.0.0-beta.3/azure-resourcemanager-avs-1.0.0-beta.3-changelog.md.

- ***DEVEM tasks and Release Planning***: Design changes and their categorization is specially used for release and milestone planning (along with workforce assignment) for the DEVEM teams (Codoban et al., 2015). For example, the component that implemented a new feature in this release may require design review and restructuring in the near future. At the same time, components that have gone through restructuring may not be planned for refactoring in the near future. Design change partitions of the detected change instances and their categories are also helpful for timely and orderly backporting them (Li et al., 2017; Chakroborti et al., 2022).
- ***Developer's Profile Buildup***: Balanced team development and proper workforce utilization are crucial for a project (Linberg, 1999). Moreover, an organization may require searching for relevant experts to employ for resolving project design challenges (Montandon et al., 2019). Finally, mapping categorical architectural change tasks (i.e., design refactoring) with the involved developers is crucial for these purposes (Bergersen et al., 2014).

For developing automated tools for the above mentioned applications, a great many studies have focused on software architectural change detection and classification (ACDC) (Ding et al., 2015; Hindle et al., 2008; Williams and Carver, 2010; Swanson, 1976; Mockus and Votta, 2000). During detection and classification, various properties are extracted from the textual description and source code of the change tasks (such as commits). Researchers are exploring those properties with various machine learning, natural language processing and non-traditional techniques for developing supportive tools (Wang et al., 2019a; Hindle et al., 2008; Dragan et al., 2011; Hattori and Lanza, 2008; Mauczka et al., 2012; Gharbi et al., 2019; Yan et al., 2016; Levin and Yehudai, 2016; Hönel et al., 2020). However, to study architectural change, a ground truth architecture at a given point in time (or a version) is extracted, and modifications of architectural elements are detected (Baldwin and Clark, 2000; Tzerpos and Holt, 1999; Wen and Tzerpos, 2004; Le et al., 2015). To that end, a number of studies propose change metrics for detecting architectural change instances (Cai et al., 2013). These change metrics are defined based on various perspectives such as operations, semantic meaning and abstraction levels (Baldwin and Clark, 2000; Wen and Tzerpos, 2004; Xing and Stroulia, 2007; Dong and Godfrey, 2008; ben Fadhel et al., 2012; Durisic et al., 2013; AbuHassan and Alshayeb, 2019).

In this paper, our goal is to investigate state-of-the-art techniques for static (or design-time) architectural change detection and classification and to point out future research directions. We specifically focus on the following research questions.

> **RQ1:** How can we categorize existing studies on architectural change detection? How much has each category been explored?
> **RQ2:** What are the concerns of architectural change classification studies? Can we identify a comparative scenario among these studies based on these concerns?
> **RQ3:** What research opportunities exist in architectural change detection and classification?

To answer these research questions, we searched six research publication platforms (such as, ACM Digital Library and Springer) using various queries related to architectural change detection and classification. We selected 45 of the most relevant studies from the over 1800 publications returned from our search results. From these studies, we

- Identify the solutions related to architectural change detection and classification
- Categorize proposed solutions based on various quality attributes and concerns
- Provide intuitive comparisons to analyze the potential and limitations of current approaches
- Identify future research directions.

Our analysis of evidence exposes that there is a lack of lightweight technique that is feasible to process hundreds and thousands of change revisions of a code base contained in a single release without human intervention or longer delay. We also anticipate that no single approach can be used as the most suitable one to detect change instances because the deployment of an approach for a particular scenario depends on the types of architecture (and its views) and abstraction levels the development team would focus on. However, ARCADE (Behnamghader et al., 2017; Schmitt Laser et al., 2020) is the most promising tool for architectural change analysis that deploys several popular change detection metrics MoJo (Tzerpos and Holt, 1999), MoJoFM (Wen and Tzerpos, 2004), A2A (Le et al., 2015), C2C (Garcia et al., 2013) and ID-SD (Lutellier et al., 2015). We also realize that in order to deploy the detection techniques in the design review process fruitfully, efficient techniques are required for capturing the design decision associativity[8] of the changed elements. However, one of the major concerns that remain unresolved in architectural change classification is when a committed change task contains tangled changes (when multiple unrelated issues such as new feature addition and bug fixing are implemented in a single commit Wang et al., 2019b). Future research in this direction can make a significant contribution to the architectural change review process. In this regard, we assume that the proposed model by Yan et al. (2016) is the most promising as it also handles tangled messages and is compatible with architectural change classification (as was studied by Mondal et al. (2019)). Overall, our survey will help the researchers in this field quickly identify the existing tools and techniques and find the directions that are yet to explore and make a comparison in this field based on various perspectives of software architecture and maintenance.

The rest of the paper is organized as follows. Section 2 provides background. Section 3 presents the survey procedure. Section 4 compares the survey with related surveys. Section 5 describes change detection techniques, and Section 6 compares the different change detection approaches. Section 7 discusses change classification studies, and Section 8 compares change classification techniques. Section 9 discusses future research options. Section 10 answers the research questions, and Section 11 concludes the paper.

2. Background

2.1. Software architecture

Usually, software architecture is considered the structures of a software system and the process (Taylor et al., 2009; Garlan et al., 2010). It documents the shared understanding of a system design (Fowler, 2003). This understanding involves how the system is partitioned into components and how the components interact through interfaces. However, according to Grady Booch,[9] *a software architecture represents the significant design decisions that shape a system, where significance is measured by the cost of change*. Thus, frequent change analysis prevents the drifts of software architecture from the original design decisions.

[8] Design decision associativity indicates mapping, tracing, and predicting software requirements/issues with the components and their change operations (Shahbazian et al., 2018a,b).

[9] beza1e1.tuxen.de/definitions_software_architecture.html.

A software architecture is viewed as (i) prescriptive architecture, and (ii) descriptive architecture (Taylor et al., 2009). Prescriptive architecture documents what is intended prior to a system being built; it can be documented with the unified modeling language (UML), the architecture design definition language (ADDL), graphs, or natural language. In contrast, descriptive architecture captures/documents that have already been implemented. Another key difference between the two views is that prescriptive is prior to a system being built, and descriptive is after the fact. Note that UML, ADDL, natural language, and graphs can be used to document either prescriptive or descriptive architecture. A descriptive architecture is also documented with implementation-level entities such as packages. However, descriptive architecture is studied at three abstract levels: high level, intermediate level, and low level. On the one hand, implementation-level entities such as modules or libraries are considered high-level abstraction. On the other hand, packages, classes, and program files are regarded as intermediate-level abstraction. Apart from these, methods, functions, and procedures are low-level abstraction. Low-level abstraction is frequently studied for understanding procedural and legacy systems. Retrieving higher-level abstraction from an implemented code base is challenging. However, a change analysis process that focuses on these abstraction levels can utilize software artifacts from various sources (Blanco and Lioma, 2012; Ding et al., 2014; Paixao et al., 2019): design and requirement documents, the code-base, issue trackers, review comments, commit messages, developer discussions and messaging lists. This section briefly discusses how software architecture is modified and how the modifications are categorized, aggregating through our survey study.

2.2. Software architecture modification

Parts of software architecture can be changed during regular development and maintenance activities. Software systems have a static and a dynamic architecture (Grundy and Hosking, 2000). The static architecture is represented by a collection of modules that are structured in a module hierarchy during design or implementation time. Whereas, the dynamic architecture represents their configurations and actions during the execution of a system at runtime (Grundy and Hosking, 2000). Thus, changes in static and runtime architecture occur in different ways. Therefore, different strategies are required for managing different changes. In this paper, we discuss how static architectural change operations are defined in the literature. However, there are many ways that architecture views and properties can be modified. Some common ways in which architecture may be modified after it has been built are Ozkaya et al. (2010):

Common operations: Common change operations of software architecture are the addition of new components, upgrading existing components (e.g., performance tuning), removal of unnecessary components (both temporary and permanent). The change operations are also referred to as Williams and Carver (2010): kidnapping, splitting and relocating. Kidnapping is moving an entire module from one subsystem to another. Splitting involves dividing a module into two distinct modules. Relocating involves moving functionality from one module to another.

Configuration change: Software configuration is also a property of software architecture (Estublier et al., 2005). Configuration may control dependency, access rules and restrictions of both design-time and run-time components (Ghorbani et al., 2019). Therefore, change operations include reconfiguration of application architecture (reconnection of components and connectors) and reconfiguration of system architecture (e.g., modifying the mapping of components to processors). Moreover, modifying values of many configuration characteristics is also considered an architectural change operation.

Source-code layers: Source-code layers are directories, package structures, and location of code files within the directories (Lutellier et al., 2015). Static changes affect system structure and these layers, and consist of changes to systems, subsystems, modules, packages, classes, and relationships. In this context, class hierarchy changes consist of modifications to the inheritance view. Whereas, class signature changes describe alterations to system interfaces.

Design model change: Changes can be made to UML diagrams, and other informal models (such as component and package diagrams) where each diagram type signifies the nature of the changes (Wen and Tzerpos, 2004; Ma et al., 2004; Garcia et al., 2013). For example, changes in UML diagrams can include the addition or deletion of class attributes, modifying relationships, and the addition or deletion of classes (Ma et al., 2004). Moreover, the employed architectural pattern of a system can be changed, such as layered architecture pattern can be migrated to model-view-controller pattern. Furthermore, addition, deletion and migration of design patterns can also treated as architectural changes.

Architectural document change: Since architecture is documented with natural language in many projects, a change in the description of the document also indicates an architectural change (Ding et al., 2015).

2.3. Categorizing software design decisions and changes

In practice, software design decisions and changes are categorized from various perspectives. The categorization is done by focusing on the requirements, design decisions and concerns, design solutions, development and maintenance activities, design failings, and so on. Identifying these categories with associated changes is essential for proper trade-off analysis to reduce current and future inconsistencies (Verdecchia et al., 2021) among other things discussed in the Introduction. During our survey, we have extracted the following perspectives on architectural change categories:

2.3.1. Requirements and decisions

A change type should be captured considering the features, requirements, decision types, etc., for better communication among the DEVEM team members and stakeholders. These categories are as follows:

Software features: Common groups of software requirements (Tang and Lau, 2014; Kurtanović and Maalej, 2017) are functional requirements (FR) and quality (non-functional) requirements (QR). *QRs* are representative of performance, portability, security, usability, and so on. The requirements are also referred to as issues (mainly in version control systems) or design decisions (Shahbazian et al., 2018a).

Decision classes: Kruchten (2004) identified three major classes of design decisions based on how the system will implement the requirements. They are referred to as *existence* — indicates the systems' design or implementation by default, *property* — it can be design rules or guidelines or design constraints and *executive* — affect the development process, the people, the organization, and the selections of technologies.

Decision Relationships: Design decisions can be grouped based on how they are related to each other such as constrains, forbids, alternative to, and so on (Kruchten, 2004).

Concepts/concerns: Requirements can be grouped based on the role, responsibility, concept, or purpose of a software system (Garcia et al., 2011; Link et al., 2019). These concerns are GUI, networking, database, authentication, and so on. Concerns are more like the arbitrary number of clusters based on particular topics/concepts. Not all of them may exist within a single application.

Tactics: Based on the approaches of solving a particular concern (Mirakhorli et al., 2012) architectural tactics can be grouped to Heartbeat, Scheduling, Authentication, Resource Pooling, Audit Trail, and so on.

2.3.2. Development and maintenance tasks

Change categories should be defined based on the DEVEM tasks for better estimating, planning, and documenting software. Categories based on these definitions are as follows:

Change purposes: Based on change purposes, change types are grouped as adaptive, preventive, corrective, and perfective. Mostly these are first defined by Swanson (1976). *Adaptive* – this change could be a reflection (Williams and Carver, 2010; Lin and Gustafson, 1988) of system portability, adapting to a new environment or a new platform. *Corrective* – it is the reactive modification of a software product performed after deployment to correct discovered problems (Chapin et al., 2001). *Preventive* — this type of changes happen to improve file structure or to reduce dependencies between software modules and components may later impact quality attributes such as understandability, modifiability, and complexity (Mohagheghi and Conradi, 2004; Williams and Carver, 2010). *Perfective* — these are the most common and inherent in development activities. These changes mainly focus on adding new features or requirements changes (Swanson, 1976; Williams and Carver, 2010; Ding et al., 2015).

Change impact: Grouping changes based on the impact is required for proper cost estimation (Carriere et al., 2010) and resource (such as developers, testers and reviewers) allocation. According to change impact, change tasks are grouped into *no impact* — architecture is not affected, *cosmetic impact* — architecture is minimally affected, *substantial impact* – requires considerable attention, and so on (Williams and Carver, 2010).

Project code-base resources: Several studies have grouped the changes in the code-base of a project focusing on resource types, and large commits such as module management, legal, source control management, etc. (Hindle et al., 2009; Dragan et al., 2011).

2.3.3. Design failure scenario

Architectural change type should be defined focusing on design flaws to better reflect design guidelines and compliance. Design problems or design failure scenarios can be grouped into eight categories (Tang and Lau, 2014). In fact, they are identified during the design review phase. This type of grouping is done from the implemented/changed systems design decisions and their solutions. Some of the groups are similar to the decision relationships (Kruchten, 2004). Some the groups are *ReqMissing* — not formally specified or ambiguously defined, *ReqConflict* — two requirements conflict with each other, *ContextMissing* — design context is missing or ambiguous, *IgnoreConseq* — no consideration of further potential negative consequences, etc.

2.4. Review of ACDC techniques

The DEVEM teams and researchers need more specific knowledge about the existing architectural change detection and categorization techniques, at least focusing on the concerns that are discussed in Sections 2.1, 2.2 and 2.3. In this regard, a literature review that aggregates information from a large body of literature would be helpful and greatly valuable to the relevant community. Therefore to enrich the knowledge base, in this paper, we analyze the existing architectural change detection and categorization techniques focusing on various crucial perspectives so that the practitioners can easily select their most suitable technique/tool and researchers can focus on the most crucial and unexplored concerns. To that end, we consider the studies of Williams and Carver (2010) and Behnamghader et al. (2017) as the starting point of our survey study.

3. Survey procedure

To find the most relevant studies on architectural change detection and classification (ACDC), we conduct a rigorous search on six platforms containing research articles: *www.dl.acm.org* (ACM Digital Library), *ieeexplore.ieee.org* (IEEE Xplore Digital Library), *www.sciencedirect.com*, *link.springer.com*, *onlinelibrary.wiley.com*, *www.world- scientific.com*, and *digitallibrary.theiet.org* (IET Software). In this process, we employed three types of queries to optimize our search results for the ACDC related articles. These queries are shown in Table 1. Notably, searching ACDC papers involves three terms such as *architectural*, *change* and *classification*. However, searching papers on these platforms with three key terms is tricky. That said, for covering a wider range while avoiding overwhelming results for architectural change detection studies, we adjust the query as *detect* AND "architectural change"*; query formulation combining other terms has similar patterns. Likewise, our architectural change classification queries contain similar patterns such as *classif* AND "architectural change"*.

Nonetheless, many software change classification techniques are aligned with the architectural change classification to some extent. Consequently, we also include papers related to typical software change classification and systematically map them with the architectural change wherever possible. To that end, the queries of usual software change categorization are formed as specific phrases since we notice from the aforementioned two types of queries that many papers are not directly related to our focused areas. Some of the example queries are *"software change classification"*, *"classify commits"*, *"commits classification"*, and so on to restrict the overwhelming outcome (queries for ACD and ACC do not provide overwhelming research articles). Search results of the six platforms are shown in Table 2. However, many papers are not relevant to change detection and classification. To handle this challenge, the query results are filtered into relevant research areas of software maintenance and engineering based on the contents in the title, abstract, introduction, and background of the papers. Finally, we do not consider the papers that focused on run-time or dynamic software architecture.

3.1. Inclusion and exclusion criteria for the final list of papers

We intend to include as much knowledge as possible about architectural change detection and classification within our search context. The inclusion and exclusion criteria for our final list of papers are presented in Table 3. However, we found difficulties in including and excluding the papers. To handle the problems, we rely on the concepts within the discussion in the papers. Additionally, we have searched the referred papers of the first list of selected papers in Google Scholar Search[10] and filtered them based on the title and uniqueness (criterion 4 for ACD). In this process, selected papers with inclusion criteria 1, 2, and 4 must also follow criterion 3 in Table 3. For the final list of ACC and SCC papers, we include that are aligned to the categories based on the development and maintenance activities and change purposes (or rationale), i,e., the perfective, preventive, corrective, and adaptive categories defined by Swanson (1976), and Williams and Carver (2010). The inclusion criteria for them are summarized in the table. Overall, we have selected 28 papers for architectural change detection and 17 papers for change categorization, totaling 45 most relevant studies.

[10] https://scholar.google.com/.

Table 1
Queries for searching ACDC papers.

Topic	Key terms
ACD	detect, metric, "architectural change", "structural change", "architectural distance", "architectural mismatch"
ACC	classification, categorization, clustering, "architectural change"
SCC	classification, categorization, clustering, software/code/program change, commit

Here, ACD — architectural change detection, ACC — architectural change classification, and SCC — software change classification.

Table 2
Search results (June 2020) using the queries in Table 1.

Platform	ACD	ACC	SCC
http://ieeexplore.ieee.org/	19	9	8
https://dl.acm.org/	170	58	42
https://www.sciencedirect.com/	166	134	14
https://link.springer.com	1124	196	15
http://onlinelibrary.wiley.com	44	32	2
http://www.worldscientific.com	114	15	6
http://digital-library.theiet.org	181	66	0

4. Contrasting our survey with the existing surveys

Several survey studies are available for software and its architectural change analysis. However, we have not found any study that systematically analyzes various perspectives on architectural change detection and classification techniques. The closest study for architectural change metrics is the survey paper by Rasool and Fazal (2017) that focused on software evolution metrics. This study analyzes metrics, models, methods, and tools used for software evolution prediction and evolution process support. They include only a handful of architectural change metrics briefly that only consider descriptive architecture. By contrast, our survey consists of a wide range of studies for architectural change detection techniques and metrics for both descriptive and prescriptive architectures and classifies them considering various perspectives such as change operations, quantitative values, architectural elements, and supportive abstraction levels. In a recent paper, Alsolai and Roper (2020) presented a review of research works related to predicting the maintainability of a software system using machine learning techniques. The survey includes maintainability measurements, usual change metrics, datasets, evaluation measures, individual models, and ensemble models. It does not focus on architectural change detection and classification techniques explicitly. Yet, some of the reported metrics can be utilized for enhancing architectural change metrics.

Williams and Carver (2010) presented a review study for defining software architecture change characterization scheme (SACCS). This is one of the pioneering studies for formalizing architectural changes available in the existing literature based on the impact and causes of change. The provided survey helps understand various types of architectural changes. Later, Fu et al. (2015) briefly discussed the existing studies (up to 2015) focusing classification of reasons for typical software changes. This study neither covers other types of code changes nor architectural changes. In comparison, our survey defines some qualitative features of various types of change classification and clustering studies (filtering from a wide range of research articles up to 2020). Then, we group and compare the studies based on those features. In another study, Jamshidi et al. (2013) proposed a survey framework to classify and compare existing architecture-centric software evolution (ACSE) research works focusing on (i) type of evolution, (ii) type of specification, (iii) type of architectural reasoning, (iv) runtime issues, and (v) tool support. This study presents an excellent understanding of the research areas on architectural evolution and studies available in those areas. Nevertheless, the survey does not cover architectural change detection and classification studies. In a relevant study, Ahmad et al. (2014) aimed to identify, taxonomically classify, and systematically compare the existing studies focused on enabling or enhancing change reuse to support ACSE. This study is an excellent survey on the change reuse process but does not focus on architectural change detection, and classification works. Towards architecture analysis, Bi et al. (2018) presented a systematic mapping study on text analysis techniques. This survey covers studies on mining, recovering, and analyzing textual architecture information. It also includes textual methods for architectural traceability. However, this study does not cover architectural change analysis explicitly.

5. Architectural change detection

Architectural change instance prediction and detection are used for numerous purposes such as design change artifacts generation, documenting after the software implementation, architectural decay measurement, design decision recovery, and traceability analysis. In the software change analysis process, the first and foremost task is predicting and detecting change instances. The most critical challenge in this process is that architectural information can be of many forms, such as abstraction levels. In this chapter, we describe the synthesized data. In Fig. 1, we present the number of publications related to architectural change detection in the period 2004–2020. We notice that the published papers on architectural change detection studies have become more frequent in the last decade than in the previous decades. A summary of the change detection metrics of these papers is presented in Table 4. Analyzing the purposes of these studies, we found that many of the change metrics (along with the change analysis) are the sub-phases of the software stability estimation, automatic architecture synchronization from source code, architecture refactoring, design recovery, and tracing, and issue prediction. Some of the important perspectives we observe in these studies are the dependency on representing and extracting architecture, change operations, abstraction levels, software properties, effectiveness measures, etc. Please note that the metrics marked with the star (*) symbol are renamed by us, considering the initials of the key terms for describing them within the published works.

Implemented architectural (descriptive) change detection for different abstraction levels is challenging where architectural components are not defined. One of the pioneering techniques for structural change detection is MoJo (Tzerpos and Holt, 1999) which does not need to define a specific architectural model. It was originally devised for measuring the closeness and distance between the two clusters of code objects following *move* or *join* operations. The *move* means moving a resource from one cluster to another (that includes moving a resource into a previously non-existent cluster, thus creating a new cluster). *Join* means joining two clusters into one, thus reducing the number of clusters by one. *Split* is another operation that is considered an architectural change. The authors treated a series of move operations of the code components as the split operation. However, due to the exponential time complexity of clustering and tracking change for n code entities, a heuristic algorithm is deployed that approximately calculates the minimum operations for clustering. Furthermore, the MoJo metric is more stable because

Table 3
Exclusion/Inclusion criteria for the final papers list.

Topic	Inclusion criterion	Exclusion criterion
ACD	1 Explicitly focused on architectural change detection or change metrics within the discussion of the contents of the abstract, introduction, and background. 2 Intended to structural change, architectural change, software decomposition distance, design change, architecture delta, architecture modification, architecture decay, high-level change, and architecture to architecture comparison within the abstract, introduction, and background. 3 Change detection techniques and metrics are used in the sub-steps of other purposes or used in the case study of architectural/structure/design change analysis or claimed other detection techniques as architectural changes. 4 Title of the referred papers in Google Scholar Search from the final collection explicitly mention architectural change detection technique/change metric and further satisfy criteria 1, 2, and 3.	1 Papers that focused on run-time or dynamic software architecture. 2 Review or survey papers. 3 Papers written in languages other than English.
ACC SCC	1 Explicit classification into three types of development and maintenance activities as defined by Swanson (1976). 2 Explicit classification into four types of change purposes (or rationale) discussed by Williams and Carver (2010). 3 Some of the change classes focused in the papers are mostly aligned with perfective, preventive, adaptive, and corrective. 4 Title of the referred papers in Google Scholar Search explicitly mention the software/architectural change classification and further satisfies the criteria 1, 2 and 3.	1 Papers that focused on clustering or arbitrary number of changes. 2 Review or survey papers. 3 Papers written in languages other than English.

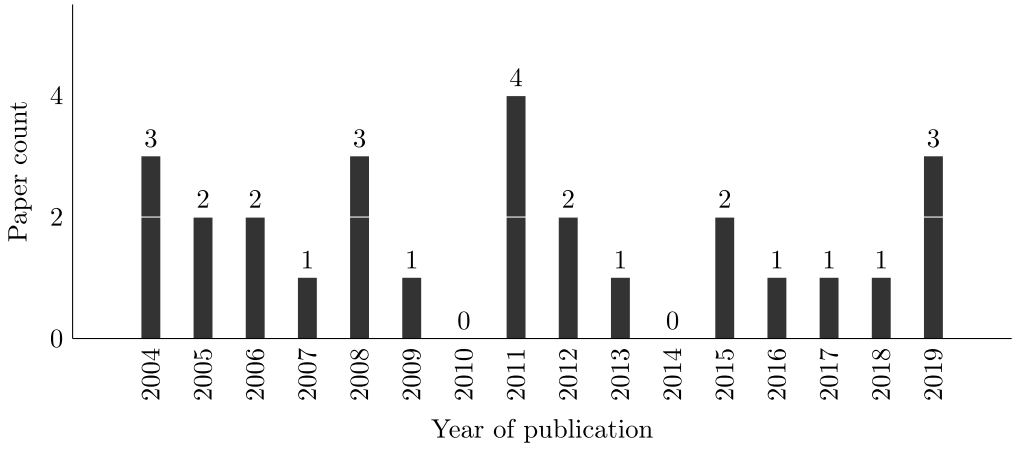

Fig. 1. Paper publication count related to change detection from 2004–2020 period.

a slight change in input codebase does not produce a significantly different outcome. The average quality (Q) value of MoJo can represent how many move operations (of partitions) have been performed from one revision of the software to another revision. One of their test systems achieved a 63.2% Q value against the partitions defined by the developers. Later, the MoJo approach is enhanced to develop a ground truth architecture recovery tool called ACDC (Tzerpos and Holt, 2000).

Implemented architectural change with design decision mapping with the components can be captured through design structure matrix (DSM) (Baldwin and Clark, 2000; Cai and Sullivan, 2006). Class or package level architecture (medium-level) is efficiently represented in DSM. From the early age of architectural change detection, the DSM has been applied. *It is a square matrix in which rows and columns are labeled with design dimensions where decisions are made*; a marked cell indicates that the code object on the row depends on the column. Although the DSM was first defined by Baldwin and Clark (2000), we found the first trace of a formal approach of constructing a DSM for software system and its application by Cai and Sullivan (2006). However, changes in values of the DSM is considered an architectural change. Furthermore, any of the dependency relations defined by module dependency graph (MDG) (Mancoridis et al., 1998), and augmented constraint network (ACN) (Cai and Sullivan, 2006) can be used for constructing a DSM from a software codebase. Therefore, logically, changes in MDG, and ACN can also be treated as architectural changes (Paixao et al., 2017, 2019). However, design rules and modules, as well as the structure of the design rule hierarchy, can be visualized using DSM.

UML (Unified Modeling Language) model can represent a higher-level proposed architecture (prescriptive) of a system. It is a de facto architectural modeling language for software. However, Ma et al. (2004) defined 17 object-oriented metrics for UML called *extent-of-change* metrics which indicate the stability of UML meta-models at the architectural level. These metrics are DSC (Design size in meta-classes), NOH (Number of Hierarchies), MNL, NCC (Number of concrete meta-classes), and so on. Since these metrics are defined for UML meta-model evolution, change of any of these values can be considered as an architectural change of a system model.

Many systems have high-level architecture defined into components and configuration management concepts (require ADDL tools). Roshandel et al. (2004) developed a tool called *Mae* by extending xADL 2.0 focusing these concepts for architectural change

Table 4
Architectural change detection studies.

ID	Authors	Description	Purpose
MoJo	Tzerpos and Holt, 1999	This metric is used for measuring the difference between two clusters using move or join operations of the constituent objects of a cluster.	Architecture change
DSM	Baldwin and Clark, 2000; Cai and Sullivan, 2006	A DSM (design matrix) is a square matrix that shows dependencies between classes/code files in a software code.	Architecture analyze
Mae	Roshandel et al., 2004	It is a tool developed using the configuration model for architectural evolution management.	Architectural change manage
EOC*	Ma et al., 2004	17 *extent-of-change* object-oriented metrics are defined for UML meta-model evolution at the architectural level.	Architecture stability
MoJoFM	Wen and Tzerpos, 2004	MoJoFM is an extension of MoJo metric considering minimum move and join operations between two structures.	Architecture change
GKD*	Nakamura and Basili, 2005	Architecture is represented in a Graph, and distance is determined using Kernel mechanism between two Graphs.	Architecture change
LILO*	Vasa et al., 2005	In-degree links (L_I) and out-degree links (L_O) of the nodes of a dependency graph indicates architectural changes.	Architecture change pattern
RC*	Dig et al., 2006	This technique utilized Shingle encoding for detecting structural changes (in refactoring operations) from two AST graphs.	Architecture refactor
UMLDiff	Xing and Stroulia, 2007	UMLDiff generates two graphs from UML models and then detect changes at all levels of logical designs.	Architecture change
HEAT	Dong and Godfrey, 2008	This tool can detect change in hybrid model composed of a collection of aggregate components and the connectors between them (derived from a package diagram).	Architecture change
$A\Delta$	Jansen et al., 2008	This technique can detect architectural delta in the multi-view model. First, module view and component-connector view are extracted from the code-base.	Design decision recover
CDI*	Khan et al., 2008	Quantitative metrics such as concentration, dispersion and inclusion are defined to detect architectural changes.	Architecture to design decision change
srcTracer	Hammad et al., 2009	Developed a tool for detecting design changes (UML diagram) from code changes.	Design tracing
CB*	Bouwers et al., 2011	Component balancing metric for architectural analyzability is devised focusing system breakdown (SB) and component size uniformity (CSU).	Architecture analyze
HEML*	Cicchetti et al., 2011	This study adopts the Model differencing technique for detecting changes in hybrid multi-view models.	Architecture change
NHK*	Steff and Russo, 2011; Russo and Steff, 2014	Presented a Neighborhood Hash Kernel based method for reducing heavy computation for detecting structural changes (between classes and dependencies).	Architecture change
CSM	Durisic et al., 2011, 2013	Complexity structure matrix (CSM) is an alternative of DSM for detecting and measuring architectural changes. CSM is calculated from *fan-in* and *fan-out*.	Product quality
VM*	Wimmer et al., 2012	A viewpoint metric (combination of add, move and modifications) is defined for detecting changes in Multi-view models. This is a part of developing a modeling language.	Architecture change
GAR*	ben Fadhel et al., 2012	A genetic algorithm is leveraged for detecting architectural changes in refactoring operations.	Architecture change
A2A	Le et al., 2015	It is based on MoJoFM, but simplified with add, remove, move operations between two structures with bipartite graph.	Architecture change analyze
C2C	Garcia et al., 2013	This metric measures the degree of overlap between the clusters (implementation-level entities).	Architecture change analyze
ID-SD*	Lutellier et al., 2015	Include dependencies and symbol dependencies (ID-SD) are used to measure distance from the ground truth architecture.	Architecture recover
DM*	Le and Medvidovic, 2016	Six architectural decay metrics are described to predict architectural change concerns.	Bug prediction
QC*	Haitzer et al., 2017	This technique is capable of detecting prescriptive architectural (defined by an ADDL) changes by analyzing source code of the implemented architecture.	Architecture synchronize
CSD*	Rástočnỳ and Mlynčár, 2018	A tool for detecting changes in UML sequence diagram analyzing source code structure.	Architecture synchronize
TUM*	AbuHassan and Alshayeb, 2019	25 metrics are defined for detecting changes in three types of UML models (class, use case and sequence diagram).	Architecture stability
MLC*	Wang et al., 2019c	A tool for change detection based on directory, files, methods, and statements.	Architecture change
TD	Mondal et al., 2019	Explored co-occurred keywords within the mailing lists for determining messages containing architectural change.	Architecture change

management. However, this tool requires manual intervention to provide specification information in XML format.

Similar to MoJo, MoJoFM (Wen and Tzerpos, 2004) is studied for higher-level implemented structural changes in software systems. MoJoFM is an enhancement of the MoJo metric to remove limitations on the effectiveness measure of the actual distance between two structures. However, the main difference between MoJo and MoJoFM is determining the minimum number of moves or join operations to represent structural differences among the cluster objects in one direction. That said, MoJoFM assumes that the component sets in the architectures undergoing comparison will be identical and limits its application in many cases. However, an alternative to the Q value of MoJo, it measures the effectiveness of the predicted move operations in between two software revisions. Overall, the best MoJoFM value for a single project is 65.82% against the developer's defined software partitions.

Existing literature considers connected graphs to represent class level implemented architecture of systems where each class is a node and the relation between two classes represents a connected edge. Vasa et al. (2005) found that the measurement of in-degree links (L_I) and out-degree links (L_O) of nodes in this graph indicates structural changes. We combinedly call them LILO. The authors investigated the unexpected range of its values with the predictability of architectural changes. However, LILO is calculated from the connectivity of the nodes of a dependency graph (DG). The DG is formed based on structural dependency relations defined by Myers (2003). Associations and type/interface inheritance are all treated as a dependency relationship. However, the authors did not investigate the accuracy of predicting the change instances.

Like Vasa et al. an intermediate-level implemented architecture is represented in a graph where nodes are architectural elements such as packages/classes/methods and edges are connections/dependencies (inheritance, association, and use dependency). Nakamura and Basili (2005) presented a metric for detecting architectural change by calculating graph kernel difference (GKD) between two software versions. In the change detection process, constructing the GKDs can be exponentially large. However, difference calculation based on *random walk kernel* reduces the computation load significantly, but it is still not feasible for detecting changes within thousands of commits of large software. Finally, the GKD metric is employed for the cost prediction of software architecture transition. However, the reported limitations of it are: (i) it cannot ensure the exact part of the structural difference, and (ii) it cannot reveal the meaning of distance.

Class level and method level refactoring operations within an implemented system are considered as a part of a large (high-level abstraction) architectural change (Dig et al., 2006). To that end, Dig et al. (2006) leveraged Shingle (fingerprints) encoding from two ASTs for detecting changes on RenamePackage (RP), RenameClass (RC), RenameMethod (RM), PullUpMethod (PUM), PushDownMethod (PDM), MoveMethod (MM), and ChangeMethodSignature (CMS). Here, PullUpMethod is a special MoveMethod that occurs when a method is pulled up from the subclass into a superclass, whereas PushDownMethod is opposite to PullUpMethod. The presented algorithm calculates the likelihood of similarity and dissimilarity from the changed code tokens contained in the mentioned entities from two AST graphs (of two software versions). Although the technique focuses on only detecting refactorings (RC), the aggregated operations are considered large architectural changes. However, this technique could detect 85% of the component change operations.

As we mentioned earlier, the UML model is representative of higher-level prescriptive architecture. For detecting changes between two UML metamodels, a promising approach called UMLDiff (Xing and Stroulia, 2007) was proposed. UMLDiff first generates two graphs from two subsequent UML models and then computes intersection and margin sets between vertices and edges. It considers both lexical (name of the code entities) and structural properties to measure similarity. It can detect additions, removals, moves, and renamings of subsystems, packages, classes, interfaces, attributes and operations, and changes to the relations among elements presented by the UML properties. Additionally, it is capable of detecting changes at all levels of logical designs (aka prescriptive architecture). A tool JDEvAn (Xing and Stroulia, 2007) has been developed based on this UMLDiff algorithm. Performance evaluation of UMLDiff shows that the recall and precision rate of it are 86% and 97%, respectively. However, this tool requires a special representation of the models for input data.

A hybrid model based on the semantic relationships of the components is considered the high-level abstraction of a software system. Following this context, Dong and Godfrey (2008) proposed a method to extract the architectural model and detect change for object-oriented software. First, their method extracts a hybrid model (various semantic relationships) from the code-base and then detects changes in aggregate components, assembly connectors, and delegation connectors. Here, an aggregate component is a collection of objects that are instantiated from the concrete classes of a package, an *inport* is the set of resources that a component provides and is used by other components, an *outport* is the set of resources that a component requires from others, a delegation connector is the required and provided interfaces of a contained component to the corresponding interfaces of its container components, and an assembly connector specifies the client–server relationship between two components. However, a hybrid model, composed of a collection of these aggregate components and the connectors between them, is derived from a package diagram preserving the containment hierarchy. In sharp contrast to the package diagram, it explicitly describes the boundary of the objects and emphasizes their usage dependencies. At the same time, classes within the package are semantically annotated as a defined class — those are implemented, ghost class — duplicate classes that the defined class inherits, exiled class — only declared, inport, outport, and connectors. We notice that the concepts used in this study for describing the metrics express the semantic behaviors of components within a system. Finally, the *HEAT* (Dong and Godfrey, 2008) tool is developed based on these metrics.

The module view and component-connector view representing key design decisions are the high-level prescriptive architecture of a system. Considering these views, Jansen et al. (2008) followed a technique to determine architectural delta after implementation as a substep of recovering architectural design decision. They first extracted the module view and component-connector view through manual analysis of the code-base. After that, change is detected on those views as modification, addition, moving, and deletion of elements. Finally, a tool has been developed to recover the architectural design decision that includes architectural delta detection. However, the performance of the tool was not investigated in this study due to the requirement of robust time, efforts, and availability of the architects/experts.

Requirements' dependency on components and their decomposition into layers is a semantic architecture of software (Hewitt, 2019). Khan et al. (2008) defined a set of quantitative metrics named concentration, dispersion, and inclusion (CDI) for such semantic architectural changes. Here, concentration indicates the requirements' dependency on components (classes, operations, and interfaces) and their decomposition into layers. On the other hand, dispersion calculates the percentage of components affected by the change in a layer or multiple layers. Then again,

inclusion measures the number of components added when a new change is integrated. We realize that these metrics can directly indicate the change impact as well.

UML model changes are architecture design changes of a proposed system. Hammad et al. (2009) implemented a tool, srcTracer (Source Tracer), to automatically identify design changes from code changes. In this tool, design changes are identified from the added/removed classes, methods, and changes in relationships (added/removed generalizations, associations, and dependencies). However, it assumes that the UML design model is available for the corresponding version of changes. Overall, this tool is less error-prone than the experts for design change traceability.

Software components and their systematic decomposition are crucial properties of the high-level architecture (during implementation). That said, component balancing (CB) between the system breakdown (SB) and the uniformity of component size (CSU) can be used to measure the decay of this architecture (Bouwers et al., 2011). CB represents a quantitative value of a system, SB indicates whether a system is decomposed into a reasonable number of components, and CSU captures whether all the components are reasonably sized. Thus, the CB metric expresses a systematic meaning. However, for this metric, the number of components needs to be specified, which is only feasible when architectural components are defined and mapped by the development team. For experimentation, authors (Bouwers et al., 2011) considered each of the first levels of source-code directory folders as a component. We notice that this metric is ill-suited where architecture is not well documented, but it is more reliable as the experts have evaluated it. The Spearman rank correlation (SC) of the CB metric associated with the expert opinion is 0.80. In summary, this is an excellent study describing how a semantic metric should be measured.

Software architecture can be described by different views focusing different concerns such as structural, and behavioral. Cicchetti et al. (2011) presented a technique that can detect changes in hybrid multi-view models, which is a sub-step of an incremental model synchronization process. In this study, a collection of class diagrams are treated as Multi-view which is a special architectural description of a system. A view is usually an area of concern in a multi-view and is mainly represented by a class diagram. In this technique, a set of rules defined by the Epsilon Comparison Language (ECL) is applied to two versions of a model (basically a class diagram). These rules measure similarity and dissimilarity between two models comparing the elements' unique identifiers (this is called model differencing technique). Here, addition, deletion, and modification of metaclasses are considered as change operations.

Classes and their dependency relations are crucial for presenting an implemented software architecture. Steff and Russo (2011) and Russo and Steff (2014) presented a Neighborhood Hash Kernel (NHK) based method for detecting structural changes (including refactorings) between classes and dependencies (defined as fan-out count). Fan-out implicitly covers added and deleted dependencies. This metric is an enhancement of the metric defined by Nakamura et al. In this technique, at first, the dependencies are presented in a graph. Generally, a graph size may be huge, and finding similarity and dissimilarity requires heavy computation. However, the NHK technique is utilized to reduce computation significantly (which is a major focus of this study). We notice that the limitation of this study is that the metric does not represent meaningful change information in the graph. In this study, Spearman correlation (SC) between defect vs fan-out is found to be up to 0.54, whereas it is up to 0.82 for defect vs NHK feature.

Similar software architecture perspectives facilitated by DSM are captured through complexity structure matrix (CSM). Durisic et al. (2011, 2013) proposed CSM as an alternative to DSM for detecting and measuring architectural changes. CSM is a square matrix where each system module is assigned to its one row and column with the same index, but its fields contain a value derived from the CSM formula. It is calculated from *fan-in* and *fan-out* (including various weight factors representative of connectivity types in between two modules). However, this study uses packages as modules/components for measuring the CSM.

As mentioned earlier, software architecture can be described by different views focusing different concerns. Wimmer et al. (2012) proposed a viewpoint metric for detecting changes in multi-view architectural model. A viewpoint is a set of concerns and modeling techniques to define the system from a certain perspective such as the functional/computational viewpoint. A multi-view can be extracted using Maude language. However, this metric is measured by structuring fine-grained changes into coarse-grained ones that represent the conceptual units. Thus, following three operations are considered as architectural change: (i) If a model element of the initial model is not matched then it generates a deletion; (ii) If a model element of the revised model is not matched then it generates an addition; (iii) If a model element of the initial model is matched to an element of the revised model then they are compared for each feature the values of both model elements (an update is generated). Finally, authors introduce a modeling language based on graph transformations and Maude for expressing high-level changes along with changes in the viewpoints.

Similar aspects of refactorings and architectural changes as discussed earlier (for the Dig et al. (2006) study) are explored by ben Fadhel et al. (2012). They explored a genetic algorithm (as a heuristic method) for detecting design structure changes in refactoring operations (we call it GAR). These operations are: Pull Up Feature — moving the attribute, and Collect Feature — one feature is moved to a class that comprises a containment reference to the class originally containing the moved feature. With GAR, they found 95% precision and 95% recall of the proposed method for the 125 test samples.

Similar to MoJo and MoJoFM, C2C (Garcia et al., 2013) is introduced to detect component-level change. This metric measures the degree of overlap between the implementation-level entities contained within two clusters. To apply this metric, clusters of the structural entities must be generated. However, several techniques are available to generate architectural unit clusters that can be adopted by C2C (Garcia et al., 2011; Link et al., 2019; Tzerpos and Holt, 2000; Mancoridis et al., 1999; Maqbool and Babri, 2004, 2007; Wong et al., 2009). This metric found an average of 56 strong matches for concern based architectural clusters (ARC) and 31 strong matches for comprehension driven clusters (ACDC). Here, clusters are similar to semantic modules/components.

Similar aspects of implemented architecture to MoJo, MoJoFM, and C2C, A2A (Le et al., 2015) metric has been recently introduced to assess system-level architectural change. A2A is inspired by the widely used MoJoFM metric: additions, removals, and moves of implementation-level entities from one module to another. This metric also includes additions and removals of modules themselves. Later, five operations from C2C and A2A are combined to construct the minimum-transform-operation (*mto*) and *aco* (operations needed to build initial architecture) metrics for transforming architecture into another one.

As mentioned earlier, classes/components and their dependency relations are crucial for an implemented architecture. A few of the studies utilized *include* dependencies and *symbol* dependencies (ID-SD) (Lutellier et al., 2015) or import change (Kim and Lee, 2014) to recover a ground truth architecture and measure changes with that architecture. Include dependency is based on information when a class or program file includes another

one irrespective of using any function or method. In contrast, a symbol dependency ensures that at least one function or method of the included class or code file is called. Import change is the modification of include dependency of object-oriented software. Overall, ID-SD is reliable for defining the ground truth architectures (concrete) and detecting changes in large systems.

Some semantic properties of software components are indicative of the stability of software architecture. Focusing on this, Le and Medvidovic (2016) described six architectural decay metrics reflecting the change concerns. These are the Ratio of Cohesive Interactions (RCI), Instability, Modularization Quality (MQ), Bi-directional Component Coupling (BDCC), Architectural Smell Density (ASD), and Architectural Smell Coverage (ASC). Other studies define the first three, and Le defines the last three. For measuring them, architectural components are extracted by ARCADE. Arguably, these metrics predict architectural change proneness and smells; thus can be associated with architectural changes. However, they found strong correlations between these metrics with bugs. Moreover, these metrics have semantic meaning expressing the concerns on maintenance and the quality of the implemented codebase.

The high-level software architecture view is represented by ADDL frameworks. Haitzer et al. (2017) presented a technique for detecting changes on architectural model designed by QVT-o ADDL framework (Query/View/Transformation - operational). This technique is the fundamental step for reconciling the prescriptive architecture with the structural changes in the source code. First, this study extracts an architectural model using ADViSE ADDL, then applies QVT-o for change transformation as addComponent, deleteComponent, addConnector, deleteConnector, and updateAbstractionSpecification involving manual intervention. These operations need to be documented in each evolution step, which is a precondition to working with the proposed approach. Moreover, this change transformation requires defining an architecture with an ADDL language (implemented with QVT-o language) that might introduce somewhat overhead over source code implementation of the system with a development platform (such as Java). However, this study reports that the tool needs the highest 0.50% manual efforts for transformation if the architecture is predefined with ADDL.

A UML sequence diagram captures the behavioral architecture of a software system. Rástočnỳ and Mlynčár (2018) developed a tool for detecting changes in UML sequence diagram (behavioral architecture) analyzing source code structure. First, the code structure and the sequence diagram are represented in hierarchical trees using the graph transformation technique, and then change is detected on both trees.

As discussed in a few paragraphs, UML models can represent various architectural views of software systems. AbuHassan and Alshayeb (2019) aggregated 25 metrics for characterizing three types of UML models (class diagram, use case diagram, and sequence diagram). Some of the metrics are: class types changed, class relationship deleted, actor relationship changed, and so on. Here, relationship indicates typical object-oriented relationships, such as association. At the same time, the class diagram represents the structural view, the sequence diagram expresses the behavioral view, and the use case diagram represents the functional view of a system. Arguably these metrics can be used for architectural change detection. In the case study, they showed that the R^2 values (represents the proportion of variations between two variables) of this metric are 0.62 for analyzability (AZ) and 0.70 for modifiability (MD).

Directory, files, and methods are various levels of architectural components of implemented software. Wang et al. (2019c) conducted a study for multi-level change (MLC) detection based on directory, files, methods, and statements. They also cluster the edits of the statements within a code change according to type. This technique can be used for intermediate and low level architectural change detection. However, for a single project, this approach detected directory level (DR) change with a 100% F1 score.

Empirical studies report that developer's discussions, textual descriptions, and mailing lists contain architectural information (Ding et al., 2015). Recently, Mondal et al. (2019) explored textual description (TD) to identify messages from the mailing lists that might trigger architectural changes in the code-base. However, the $F1$ score of the proposed approach is 60% which is not applicable reliably.

6. Perspective analysis of the change detection techniques

After the preliminary review of the architectural change detection metrics, we have performed a comparative analysis of them based on the following perspectives and attributes of software systems.

- Required input and supportive tools of the change detection techniques.
- Design decision associativity with the changed elements.
- Frequency of change analysis.
- Programming languages supported by the proposed techniques.
- Types of operations considered as baseline for architectural change.
- Levels of abstraction of software structure supported.
- Code properties used for defining an architectural instance.
- Dependency on techniques of extracting ground truth architecture.
- Effectiveness measures of the proposed metrics.
- Relationships among the proposed metrics.

6.1. Analysis focusing input data

The most important information for architectural change detection techniques is which data is required to input into the techniques. Therefore, we analyze the studies based on the complexity of providing input data for measuring the change detection metrics. In doing so, we needed to analyze at least two extra papers per study to extract input data and required tools. For example, we studied additional four papers to obtain input data and tools for MoJo. After the initial phase of the analysis, we realize that the metrics requiring manual effort, ADDL language coding, and byte code and abstract syntax tree (AST) generation are heavyweight techniques (at least for large projects). For example, ARC (Garcia et al., 2011) may take 14 h to recover architecture for a single revision of the Chromium software (Lutellier et al., 2015). However, they are feasible for change analysis for a few revisions (e.g., commits) of the code base. In contrast, they are infeasible for both researchers and developers to frequently deploy them for processing many revisions (or commits) of the code base. For instance, a release of software may contain hundreds of revisions, and recovering design decisions requires processing all of them (Shahbazian et al., 2018a). Although developers usually have compiled bytecode for each revision, they need to provide both the source code and bytecode pairwise (two versions each time) to the most promising change analysis tool ARCADE, which is built on MoJoFM, C2C, A2A, ID-SD, etc. (Schmitt Laser et al., 2020). ① For 100 revisions of the code base, they need to provide input about 99 pairs of complete code base + bytecode (or converted code) subsequently in this tool, which is a quite tedious and time-consuming process. ② Moreover, for large project like Chromium, recovering architecture with ARC for change detection

Table 5
Case study, design decision associativity (DDA) and input data.

Study id	Language	Case study	Project type	DDA	Input data	Input tool
MoJo	C,C++	∼	OS	∼	RDM	Bunch (Mancoridis et al., 1999) ← CIA, Acacia (Chen et al., 1998)
Mae	ADDL	Yes	CSP	Yes	RDM	xADL 2.0
DSM (Cai and Sullivan, 2006)	Java	Yes	CSP+ OS	Yes	Design Net	Simon (Cai and Sullivan, 2005)
EOC✓	UML	X	OS	X		Manual
MoJoFM	C,C++	Yes	OS	Yes	RDM	Bunch ← CIA, Acacia
GKD✓	Java	Yes	OS	X	AST/Graph	
LILO✓	Java	Yes	OS	X	Graph	BCEL (BCEL, 2020)
RC✓	Java	Yes	OS	X	AST	
UMLDiff	Java	Yes	OS	X	AST/DOM	JDEvAn
HEAT✓	Java	Yes	OS	X	AST/Graph	
AΔ✓	CORBA	Yes	OS	Yes		Manual
CDI✓	Java	Yes	CSP	Yes		Manual
CB✓	C,C++, Java, .NET	Yes	CSP+OS	X		Manual
srcTracer	C,C++	Yes	OS	Yes	XML of code	srcDiff ←srcML (Collard et al., 2003)
HEML	ADDL	Yes	OS	X	ADDL	ECL (ECL, 2020)
NHK✓	Java	Yes	OS	X	Graph	CDA (CDA, 2020), BCEL
CSM	OOP	Yes	CSP	X		∼
VM✓	ADDL	X	X	X	ADDL	Maude
GAR✓	GMF	X	OS	X	GMF	Manual model
A2A✓	C,C++, Java	X	OS	∼	AST	ACDC, PKG
C2C✓	C,C++, Java	∼	OS	∼	AST	ARC (Garcia et al., 2011)
ID-SD✓	C,C++, Java	∼	OS	∼	AST	LLVM (LLVM, 2020)
DM✓	Java	∼	OS	∼	AST	ARCAD (Behnamghader et al., 2017)
QC✓	ADDL	Yes	OS	Yes	ADDL	Manual
CSD✓	UML	Yes	CSP	Yes	AST	MoDisco (Bruneliere et al., 2010) ← EMF (EMF, 2020)
TUM✓	UML	X	X	X		Manual
MLC✓	Java	X	OS	X	AST	
TD	Independent	X	OS	X	Texts	

OS — open-source, CSP — close source project, RDM — data model of code, X — not exists, ∼ — implicit info.

requires at least 1400 h for those revisions (Le et al., 2015). These challenges are appalling for the researchers because they usually need to construct a dataset containing thousands of change samples for better evaluation and insights. For them, building projects to generate byte code require human intervention to fix 3rd party library dependency issues, which may need several hours. Then again, the bottlenecks of ① and ② exist. Overall, heavyweight techniques are not feasible where many revisions (or commits) are processed.

Analysis of the change detection studies focusing on the input data helps us detect heavyweight techniques. Input data extraction tools for the metrics are presented in Table 5. The checkmarks in column *Study id* in this table represent the confirmed heavyweight techniques. We notice that many tools are available for input data for Java-based systems. However, those are dependent on the byte code generation. For C and C++ projects, Bunch (Mancoridis et al., 1999), CIA, ACACIA (Chen et al., 1998), and srcML tools are frequently used for input data extraction, requiring the source code to convert into a relational data model format (RDM). Furthermore, some studies deployed commercial tools for dependency relation extraction, such as CDA (2020) and Understand (2020). Thus, our observation of data extraction and processing strategies reveals that most of the techniques are heavyweight.

6.2. Analysis focusing design decision associativity of the metrics

Design decision associativity (DDA) consists of a design decision, its design elements and reasoning for the solution (simply features/concerns) (Shahbazian et al., 2018a). We have analyzed the change studies based on DDA because the number of components (i.e., *a*, *b* and *c*) or their dependencies involved in implementing a feature can be changed in the next version (i.e., *a*, *b* and *d*). Thus, mapping the features with the changed components is helpful for design review and traceability. However, to find the DDA, we synthesized the original purpose of the metrics (i.e., the techniques' focus on design decision recovery), case study with developers and design decisions, and artifacts used for the experiments (i.e., use of the requirements/decisions for the evaluation), and explicit or implicit explanation of associativity with the requirements specifications. The analysis outcomes are summarized in Tables 4 and 5. During our analysis, we have found that two studies explicitly target the design decision recovery (*AΔ* and CDI). Another study also focused on flaw analysis (DM Le and Medvidovic, 2016). However, sixteen studies provide case studies of the metrics focusing on the design decisions. In conclusion, at least seven metrics are directly/indirectly applicable for extracting the DDA.

6.3. Frequency of change analysis

We have observed the steps and purposes of the change detection, which are presented in the *Purpose* column of Table 4. Analyzing those, we realize that most techniques focused on the difference and distance between two software revisions. Thus, they are adaptable to consecutive commits/revisions, releases, or development stages for architectural change analysis.

6.4. Programming languages considered by the studies

We have extracted the programming language information from the projects considered in the case studies and evaluation of the proposed techniques. Table 5 summarizes this information. This table illustrates that most of the techniques support C, C++, Java, or ADDL. However, focusing on developing legacy system techniques has been rare in the last two decades.

6.5. Change operations type centric analysis

From the literature, we have extracted the following operations in the design models (such as UML), architecture view, ADDL, and relevant code elements considered to be architectural

Table 6
Change operations and code properties considered for change detection.

Metrics	Add	Delete	Move	Other	Value	Properties
MoJo	Y		Y			Directory, code file
DSM					Y	Dependency
EOC					Y	UML elements
Mae				Y		ADDL
MoJoFM	Y		Y			Directory, code file
GKD					Y	Code files, methods
LILO					Y	Class
RC			Y	Y		Package, class, method
UMLDiff	Y	Y		Y		UML elements
HEAT				Y		Code model
*A*Δ	Y	Y	Y	Y		Module, component-connector
CDI					Y	Class, operations, ports
CB					Y	Module numbers, size
srcTracer	Y	Y		Y		Class, method, relations
HEML	Y	Y		Y		Multi-view model
NHK					Y	Class, dependency
CSM					Y	Package
VM	Y	Y		Y		View model
GAR			Y	Y		Metamodels
A2A	Y	Y	Y			Directory, code file
C2C	Y	Y	Y			Directory, code file
ID-SD	Y	Y				Code file
DM					Y	Package
QC	Y	Y	Y			ADDL properties
CSD				Y		Sequence diagram
TUM				Y		Three types UML
MLC				Y		Directory, ..
TD				Y		Co-occurred words

change: add, delete, move, split, update, and component renaming. We realize that existing techniques have considered various combinations (either all or a subset of them) of these operations for determining architectural changes. Apart from these, fluctuations in various quantitative values (such as fan-in and fan-out count) are also indicative of an architectural change. In this regard, DSM (Baldwin and Clark, 2000), EOC (Ma et al., 2004), LILO (Vasa et al., 2005), GKD (Nakamura and Basili, 2005), CDI (Khan et al., 2008), CB (Bouwers et al., 2011), NHK (Steff and Russo, 2011), CSM (Durisic et al., 2013), and DM (Wimmer et al., 2012) metrics are based on quantitative value. The rest of the metrics and techniques directly consider change operations. However, quantitative metrics (QM) require fine-tuning parameter values that may not be versatile. Moreover, in contrast to other metrics, QM cannot extract change instances of the architectural properties. It is noteworthy that the co-occurrence of various keywords (TD Mondal et al., 2019) within the textual description also indicates architectural changes (to some extent). Interestingly, we have found one study (Mae Roshandel et al., 2004) that directly considered configuration and constraints for detecting architectural changes, which are the common elements of documenting software architecture. Metrics, their corresponding change operations, and which properties are used for measuring them are presented in Table 6. *Other* column in the table represents updating an element or changing connection and dependency relation (connectors), and so on.

From our change operation centric analysis, we see that defining the modification operations is essential before selecting a change detection technique. We also noticed that the HEAT (Dong and Godfrey, 2008), CDI (Khan et al., 2008), CB (Bouwers et al., 2011), GAR (ben Fadhel et al., 2012), and DM metrics (Le and Medvidovic, 2016) attempted to define (and a few of them validate) the systematic meanings of the change operations related to concerns and flaws in the codebase. For example, in HEAT, some change operations associated with a particular area of the systems are defined as the change in client–server relationships. A logical definition of the change operations such as this is more understandable to the development and maintenance team. However, other studies have not experimented with or discussed the expressiveness or semantic meaning for comprehending them properly.

6.6. Property centric analysis of the detection features

We have identified a list of concrete software documentation and codebase properties used for the change metrics: directory, package, code file inclusion, component size and numbers, class reference, and procedure call. Mapping of these properties with the associated studies is listed in the *Properties* column of Table 6. We found that many prescriptive architecture metrics leveraged UML properties, while a few utilized the ADDL properties (HEML and QC). From the *properties* column, we also notice that most of the detection techniques consider object-oriented properties. Nonetheless, one study (TD) explored textual properties within the developer's discussion for predicting a change task. Although the directory, code file, and method properties can be used to detect primary architectural changes, little focus has been given to detecting meaningful architectural changes (in terms of modules, components, constraints, and connectors) considering various architectural views of the non-object-oriented systems. In terms of the descriptive architecture, we feel concerned about the recent consideration of architectural properties (Mo et al., 2019). In this concern, researchers can focus on the ambiguity of whether logical change coupling (two local segments from two components are frequently changing together without direct dependencies) would be appropriate for treating an architectural change instance or not.

6.7. Analysis regarding software abstraction level

Systematically modeling the changes of software architecture at different granulates and from different viewpoints is beneficial to the system maintainer to understand exactly why a system's design is the way it is and plan accordingly for the future change (Dong and Godfrey, 2008). As we have discussed in Section 2, software architecture can be viewed and described in three abstraction levels. In this regard, during the design review process, different development teams may give emphasis

Table 7
Software abstraction level supported.

Metrics	High	Medium	Low	Type	Depends	Outcome
MoJo	Y:cluster, module	∼	∼	Descriptive	Extract cluster	Top 63.2% Q
DSM		Y:class, package, file	Y	Descriptive		NM
EOC	Y: model			*Prescriptive*	Define metamodel	
Mae	Y:config+model			*Prescriptive*	Define model	Usable, scalable
MoJoFM	Y:cluster, module	∼	∼	Descriptive	Extract cluster	Highest 65.8%
GKD		Y:class,file	Y	Descriptive		NM
LILO		Y:class		Descriptive		NM
RC		Y:package, class	Y	Descriptive		P,R > 85%
UMLDiff	Y: UML			*Prescriptive*	Define model	R 86, P 97
HEAT	Y:component view	∼		Descriptive	Extract model	NM
$A\Delta$	Y:modules			*Prescriptive*	Extract two views	NM
CDI		Y:class		Descriptive		NM
CB	Y:modules, views			Descriptive		0.80 SC
srcTracer		Y:class	Y	Descriptive	Define uml	Better than experts
HEML	Y:hybrid view			*Prescriptive*	Extract hybrid view	NM
NHK		Y:class		Descriptive		0.5 SC
CSM		Y:package		Descriptive		NM
VM	Y:multi view			*Prescriptive*	Multi-view extract	NM
GAR		Y:attribute		Descriptive		P 100, R 92
A2A	Y:cluster, module	Y:package	∼	Descriptive	Define module, clusters	NM
C2C	Y:cluster	∼	∼	Descriptive	Extract cluster	56 ARC, 31 ACDC matches
ID-SD		Y:file, class		Descriptive		NM
DM	∼	Y:package		Descriptive	Extract component/cluster	NA
QC	Y:ADDL			*Prescriptive, Descriptive*	Define and extract model	.50% efforts
CSD	Y:Sequence diagram			*Prescriptive, Descriptive*	Define sequence diagram	NM
TUM	Y:use case, sequence diagram	Y:class		*Prescriptive*	Three models	R^2 .62 AZ, .70 MD,
MLC		Y:package, class	Y	Descriptive		100% F1 for DR
TD	∼	∼		Descriptive		60% F1

NM — Not mentioned.

to different abstraction levels. For example, neither low-level nor intermediate-level abstraction is feasible to understand and communicate by the team members of software systems consisting of thousands of classes or code files. Therefore, labeling the studies focusing on the abstraction levels would be helpful for the review team to deploy the appropriate tool for a particular scenario. Hence, the grouping of the existing studies based on these abstraction levels is shown in Table 7. The table also summarizes what types of architectural models or presentations are considered for those abstractions by the studies.

From the discussion of these studies, we realize that DSM, MDG, RC, GKD, CDI, and MLC are suitable for low-level change. Whereas LILO, srcTracer, ID-SD, A2A, DSM, and GKD can be used for intermediate-level change detection. MoJo, EOC, MoJoFM, UMLDiff, HEAT, $A\Delta$, CB, HEML, VM, C2C, A2A, DM, QC, CSD, and TUM focused on high-level change detection (some of them also consider the intermediate-level change). However, some of the metrics may be tuned for intermediate and low-level architectural changes; those are marked by the ∼ symbol in Table 7. We assume that MoJo, MoJoFM, A2A, and C2C metrics can be adjusted for intermediate-level (i.e., classes) and low-level (i.e., methods) architecture.

Furthermore, our selected studies emphasized prescriptive and descriptive architectures in different ways. Since high-level abstraction is the conceptual/logical model, most of the prescriptive architectural change detection techniques fall under this abstraction. Interestingly, the studies by Haitzer et al. (QC) (Haitzer et al., 2017), and by Rástočný and Mlynčár (CSD) (Rástočný and Mlynčár, 2018) detect changes both in the descriptive and prescriptive architectures. In a few of the studies, a prescriptive model (such as a sequence diagram) is synchronized with the descriptive change. However, change detection techniques are hardly found in the literature for supporting legacy systems. Overall, we realize that no single metric is said to be the best-suited metric due to the requirements of supporting various models, abstraction levels, views, and types of software architecture.

6.8. Analysis regarding dependency of extracting architectural representation

We found that thirteen metrics and techniques for architectural change detection are explicitly dependent on other techniques to extract clusters, models, subsystems, and modules. Several detection metrics rely on architectural view and model extraction methods: EOC, UMLDiff, HEAT, $A\Delta$, HEML, VM, A2A, CSD, and TUM. In the *Depends* column of Table 7, *extract* indicates that it requires another technique for extracting relevant cluster or model, and *define* indicates necessity of manual intervention. This table also illustrates additional information about supportive architectural types and models/views. From our analysis, we found that TUM (AbuHassan and Alshayeb, 2019) aggregates the most prescriptive architectural models.

We noticed that software systems built on packages and classes are good candidates for detecting intermediate or low-level architectural changes by many automatic techniques. Therefore, in those cases, other metrics such as DSM (Baldwin and Clark, 2000), GKD, DG, NHK, CSM, and ID-SD would be straightforward in measuring changes in the descriptive architecture, but they may ignore some important information about the separation of concerns and granularity. According to Wen and Tzerpos (2004), an automated architectural clustering should reach 92.8% matching compared to clustering by human experts. However, it is almost impossible (so far) to achieve (even) close to this standard by the available approaches (Tzerpos and Holt, 1999; Wen and Tzerpos, 2004; Maqbool and Babri, 2004; Tzerpos and Holt, 2000; Garcia et al., 2011; Corazza et al., 2011). Therefore, automatic change (high-level) detection of the descriptive architecture, which depends on clustering, cannot be said to detect 100% pure changes (false positive is expected). Only expert intervention can ensure architectural change detection's highest accuracy in cluster/higher levels (MoJo, MoJoFM, C2C, A2A, CB). Moreover, clustering large systems is computation-intensive, and the analysis of thousands of change versions is almost infeasible. Yet, change metrics that depend on clustering can be tuned to detect changes in non-object-oriented systems.

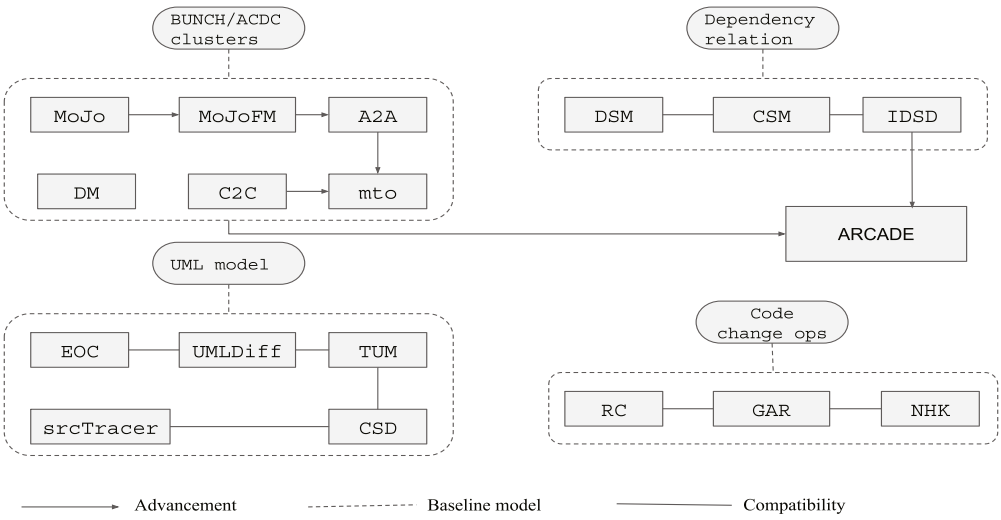

Fig. 2. Change metrics relation.

6.9. Analysis regarding ways of measuring effectiveness

Analyzing the existing studies, we categorize the effectiveness measures of the architecture change detection approaches based on the architecture representation. The cluster-based techniques are measured by the distance between clusters, number of moves, and join operations. At the same time, module or component and value-based methods used accuracy — precision (P), recall (R), F1 scores, and correlations (SC). However, detection approaches that utilize the available models or views use architects' efforts, analyzability (AZ), and modifiability (MD). Surprisingly, we have found at least 13 studies that do not mention any explicit information about the effectiveness of the change detection techniques. The performance outcome and effective measures of various metrics are summarized in the *Outcome* column in Table 7. From the outcome data, we found that performance and effective measures have a significant gap, especially for detecting module-level changes.

6.10. Relationship among the change metrics

We have analyzed the relationship among the change metrics based on the subsequent enhancement of limitations and baseline models/techniques used. In this regard, we discuss the metrics based on the explicit information provided by the studies. That said, MoJo, MojoFM, A2A, C2C, and DM are closely related and compatible. They all rely on produced clusters from BUNCH, ACDC, or similar tools. Moreover, they are subsequently enhanced to eradicate the limitations of the previous one. Finally, they are all employed to develop the most promising architectural change analysis tool ARCADE (Behnamghader et al., 2017; Schmitt Laser et al., 2020). Another group of metrics, EOC, UMLDiff, srcTracer, CSD, and TUM are based on the UML model. We also found that DSM, CSM, and ID-SD metrics are compatible as they are based on component dependency relations. Similarly, RC and GAR are studied based on identical change operations. All these relationships are presented in Fig. 2 to understand them fast. Which metrics are subsequently enhanced based on other metrics are shown with the "arrow" in the diagram. This analysis helps the researchers and practitioners get insights for quick decision-making to employ them. For example, it is noticeable that the *mto* metrics might be the most promising representative of some other metrics due to the enhancement of their limitations. However, some of the included metrics might have implicit relations among them.

7. Architectural change categorization

Our survey also focuses on the automatic techniques for architectural change categorization. Because, as soon as the architectural change revision is detected, it is also required to determine the cause or purpose of the change to better represent the change knowledge and generate various software documents (e.g., change logs for a release Bi et al., 2020). Moreover, change categorization is essential for design decision recovery (Shahbazian et al., 2018a,b). It is first introduced by Swanson (1976) focusing the software development and maintenance activities. Since then, researchers have conducted many studies to classify software change and maintenance activities; many of them are directly or indirectly related to architectural change (Swanson, 1976; Hindle et al., 2008; Hönel et al., 2020). However, typical software changes (revisions or commits) may be either local code changes or architectural code changes. Thus, architectural change revisions are the subsets of typical (local+structural) software change revisions. That said, typical change revisions classification models are adaptable to architectural change classification to a greater extent. Fig. 3 shows the number of published papers in the period of 2000–2020 that we surveyed relevant to change categorization. From the figure, we observe that the number of published papers has increased significantly in recent years, indicating the growth of importance of change category information to the open world. Researchers considered various sources of software artifacts for grouping the changes focusing on multiple concerns. In the following, we discuss state-of-the-art change classification techniques related to both typical and architectural changes. Up to date, existing literature (Ding et al., 2015; Williams and Carver, 2010) have extracted four types of causes of software changes (i.e., perfective, preventive, corrective and adaptive) as discussed in Section 2. We notice that software change classification approaches adopt both code and textual properties for classifying them.

7.1. Textual properties

Textual descriptions are major sources of various architectural information. Most of the change classification methods focused on natural language processing, text retrieval, and machine learning techniques leveraging textual properties. The followings are the popular classification methods.

One of the pioneering studies for classifying the causes of code change is conducted by Mockus and Votta (2000) employing

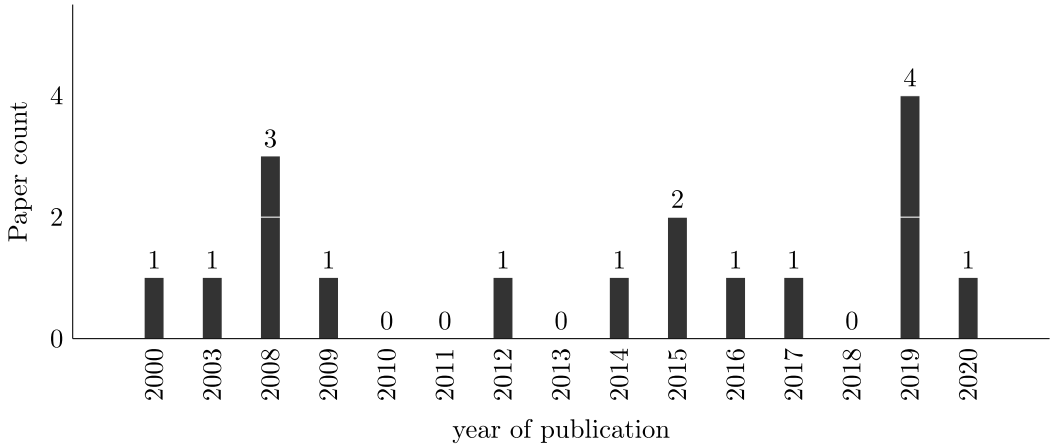

Fig. 3. Minimum paper publication count for change categorization from 2000–2020 period.

direct lexical analysis (word presence and count) into *adaptive* (A_m), *perfective* (P_m), *corrective* (C), *inspective* (I), and not sure categories. However, adaptive and perfective concepts are different than Swanson (1976) classes (as explained in Section 2). The authors first extracted discriminative keywords from the description texts of 20 modification requests for each of the classes. Then, this word list is employed for classifying where the presence of a keyword is the indicator of a type, but if more than one type of keyword is present, the type with the most number of keywords in the description is considered. However, if that measure also has the same value, then it prioritizes perfective and then corrective types. The outcome of the technique has 61% agreement with the developers measured by the *Kappa* coefficient. One interesting finding of this study is that the more restructuring tasks (they call them the perfective tasks) are linked to the longevity of the commercial products.

Hassan (2008) followed the work of Mockus and Votta. Their model classifies bug fix (BF), feature introduction (FI), or general maintenance (GM) types based on the keywords similar to Mockus and Votta (2000). During the classification process, it first prioritizes the BF class because the keywords for this class are less ambiguous than those of other classes. In both of these studies, the outcome is checked with some developers using the *Kappa* coefficient (Hassan, 2008). However, the predicted samples have a 70% agreement with the developers. Again, the category which cannot be classified is separated into not sure (NS) types; thus, the performance is not indicative of the actual outcome. One key finding of this study is that there are ambiguities among the senior and less-experienced developers on how they determine a change type.

Hindle et al. (2008) explored the classification of large change commits that impact more than one code file and other major concerns such as licensing documents. These classes are as follows: *implementation* — new requirements, *maintenance* — maintenance activities, *module management* — the way the files are named and organized into modules, *legal* — related to the license or authorship of the system, *quality (non-functional) source changes* — source code that did not affect the functionality of the software, *source control system change* — the manner the Source Control System is used by the software project such as branching and *meta-program* — files required by the software that are not source code. Many of these classes are the extension of Swanson (1976) maintenance categories. However, since their process is manual, it is difficult to employ this process for large commits and other projects. Following the manual classification, Hindle et al. propose an auto classifier (Hindle et al., 2009) using machine learning techniques to categorize the seven large commits. They combined three types of features: word distribution, author information, and module or file type. Word distribution is calculated using Bayesian type learning on word frequency. Machine learning techniques employed in their studies are J48 (decision tree), Naive Bayes, and so on. However, the best model achieved more than 50% accuracy. As the authors report, the manual annotation for creating a ground truth remains problematic. Overall, the most valuable outcome of their study is a set of keywords that are later experimented with, modified, and extended by various studies. Finally, many of the large commits are architectural.

Hattori and Lanza (2008) proposed a keyword-based technique for classifying commits into *forward engineering, reengineering, corrective engineering, and management activities*. Here, *forward engineering* activities are related to the integration of new features and implementation of new requirements. *Reengineering* activities are related to refactoring, redesign, and other actions to enhance the quality of the code without adding a new function. *Corrective engineering* indicates defects, errors, and bugs in the software. *Management activities* are unrelated to codification, such as formatting code and cleaning up and updating documentation. In this study, first, they group the commits into four based on the number of files affected: tiny, small, medium, and large. For that, classification is experimented with these groups in separate cases. After that, the authors curated a list of keywords for each of the categories, and then they predicted a commit based on the keyword appearance. However, the first keyword in the commit description indicates the relevant change category. Overall, this technique produces a 76% $F1$ score.

Mauczka et al. (2012) presented a word-dictionary based technique to categorize the commit message into Swanson (1976) maintenance categories. They adopted the classifiers presented by Hassan (2008) as bug fixing, feature introduction, or general maintenance changes based on keywords. For that purpose, the authors extracted a set of words for each of the categories and developed a plugin called *Subcat*. Furthermore, they created a benchmark dataset annotated by the real-world developers (Mauczka et al., 2015). Additionally, they merged keywords from the early study of Hindle et al. (2008). However, the most important finding of this study is that there were ambiguities among developers regarding the explanation of the adaptive and perfective categories.

Ding et al. (2015) explored the four causes (adaptive, perfective, corrective, and preventive) of architectural changes in open-source software. They analyze the communication and discussion messages of the developers based on the explanation of Williams and Carver (2010). However, their categorization technique is manual.

Fu et al. (2015) proposed a semiLDA (semi-supervised) technique by extending LLDA (Ramage et al., 2009) for classifying the Swanson (1976) maintenance categories. The main difference between LDA and semi-supervised LDA is the generation of the topic. In the semi-supervised LDA model, they added signifier documents to change the unsupervised training process into a semi-supervised fashion. The signifier documents can influence the generation process of the words because they can increase the co-occurrence frequency of the keywords which belong to the same category used by human supervision. However, they used keywords extracted by Mauczka et al. (2012). Their test set was 50% samples, and they did not employ cross-fold validation. Among the explored models, SemiLDA achieved 70% $F1$ score and can determine 80% change message, but the proposed method cannot decide 20% of the messages. Moreover, too brief messages are pruned during the dataset creation (but did not mention the criteria of being too brief).

Yan et al. (2016) presented a Discriminative Probability Latent Semantic Analysis (DPLSA) technique to classify the Swanson classes. This model initializes the word distributions for different topics using labeled samples to categorize the change messages. Their proposed technique creates a one-to-one mapping between the extracted topics and the change classes. However, authors utilized the keywords extracted by Mauczka et al. and claimed that the multi-category (tangled) classification is improved compared to LLDA, SemiLDA and Naive Bayes. Moreover, the study also included the not sure (NS) category like Hassan et al. Yet, they have not tested the DPLSA classifier with the standard cross-fold validation to reduce the over-fitting problem; only the model was tested with a separate set. Thus, DPLSA's performance might not be conclusive enough.

Mondal et al. (2019) explored text classification techniques for categorizing architectural change instances (into four) based on the explanation of Ding et al. and SACCS (Williams and Carver, 2010). They adopted LLDA, SemiLDA, and DPLSA for developing a auto classification model (and a set of keywords). However, the models produce poor performance (around 45% $F1$ score), and do not support tangled messages.

Gharbi et al. (2019) presented a multi-label active learning-based approach (using LibCat) to handle the problem of multi-classification of commit messages into three categories. The explanation of the adaptive category is contradictory to Swanson. They used keywords as features for the classification model. The adopted features were automatically generated from the original commit messages using the Term Frequency-Inverse Document Frequency (TFIDF) technique over the change types. With the proposed model, the average $F1$ score of the classification is 45.79%. However, they did not experiment with individual change types' performance with respect to tangled changes. Then again, they discard the commit messages having less than six characters for their experiment. Another concern of their study is that only a portion of the dataset is manually labeled; the rest is labeled by an automatic technique. Thus, there is a doubt about the reliability of this study in practice since previous techniques have achieved around 60% accuracy when all types of samples are considered. A key finding of this study is that the keywords have multiple meanings with ambiguities.

7.2. Source code properties

Dagpinar and Jahnke (2003) explored code and class properties for predicting perfective/adaptive, corrective, and preventive classes. The number of statements TNOS, Non-inheritance class-method import coupling NICMIC, Non-inheritance method-method import coupling NIMMIC, and Non-inheritance import coupling NIIC are the best metrics producing 62–99.7% R^2 values for perfective/adaptive and corrective classes. Most of the metrics represent architectural properties as discussed in Section 6. That said, this study can be treated as the first to use architectural metrics to classify changes. However, the outcome of the preventive class is not discussed.

Levin and Yehudai (2017) utilized Fluri's (Fluri and Gall, 2006) taxonomy of 48 source code changes (for example *statement_delete, statement_insert, removed_class, additional_class*, etc.) to classify commits into Mockus (Mockus and Votta, 2000) classes. However, they also attempted keyword searching within the commit message, which produced a poor result. Later, they explored a classification model based on a hybrid classifier (J48, GBM, and RF) that exploits commit keywords and source code changes (e.g., statement added, method removed, etc.). This approach can select an alternate model during predicting a class if there is an ambiguity in determining a type. Finally, the best class's precision is 56%, recall is 96%, and F1 scores would be 70.7%.

Mariano et al. (2019) proposed an improvement of Levin and Yehudai approach to classify commits into Mockus (Mockus and Votta, 2000) classes. Particularly, they included three additional features (method change, statement change, and LOC change) in the classification model for XGBoost (a boosting tree learning algorithm). Hence, the total number of features is 71: 20 keywords, 48 change types, and additional three features. The best accuracy (ac) with this technique was 77% with Levin's (Levin and Yehudai, 2017) dataset. However, precision (how many retrieved samples are relevant to a particular change type), recall, and F1 scores are not presented, and the overall outcome is unlikely to be better than (Levin and Yehudai, 2017).

Wang et al. (2019a) proposed a method to classify large review commit (having more than two patch sets) into nine categories: bug fix, resource, feature, test, refactor, merge, deprecate, auto, and others. They have employed various types of features for classification: text, developers' profiles, and change operations. Additionally, various machine learning classifiers explored by Hindle et al. (2008) are experimented with in this study. However, the specialty of the approach is that the different combinations of features are used for predicting different groups (achieves 67% $F1$ score).

Unlike the previously mentioned classification models, Hönel et al. (2020) introduced source code density, a measure of the net size of a commit. The experimental outcome shows it improves the accuracy of automatic commit classification compared to Levin's approach. However, code density is the ratio of functional code and gross size, and functional code is the total code that is executed any how. In comparison, gross code contains comments, whitespaces, dead code, and so on, along with called code. This density feature is calculated on each of the change operations (file or statement deletes or add) individually. For all data, the best accuracy is 73%, and Kappa is 0.58. For cross-project, the best accuracy is 71%. However, they did not present precision, recall, and F1. It is highly likely that the best F1 scores are similar to the baseline study of Levin and Yehudai (2017). However, they reported that the textual properties produce better outcomes in a few cases.

8. Qualitative analysis of the change categorization techniques

In the previous section, we have briefly discussed around 23 studies of software code change clustering and classification (including both manual and automatic). The summary of those studies are presented in Table 8. However, only a few of the techniques are available for architectural change classification. We have also performed a qualitative analysis of the studies on the basis of the following characteristics.

Table 8
Software change classification studies.

ID	Authors	Description
SD0	Swanson, 1976	This is the baseline study for classification of the maintenance tasks as perfective, adaptive, and corrective. This work defined an initial set of keywords for them.
SD1	Mockus and Votta, 2000	This study extracted discriminative keywords for each of the five classes and then create a classifier (**two extra are Inspective and ambiguous**) using simple lexical analysis technique (word presence, count, and prioritization).
SD2	Dagpinar and Jahnke, 2003	Authors explore code and class properties for predicting three change types. TNOS, NICMIC, NIMMIC, and NIIC are found to be the best metrics.
SD3	Hassan, 2008	A similar technique of Mockus and Votta to classify into BF, FI, GM, and not sure (NS) from commit messages. They also prioritized BF category. *Kappa* has 64.
SD4	Hindle et al., 2008	Manually categorize the large commits involving more than one code files, licensing, documentation, and so on into seven classes from commit messages and other non-code files.
SD5	Hattori and Lanza, 2008	Propose a keyword based technique for classifying commits into forward engineering; and reengineering, corrective engineering and management activities.
SD6	Hindle et al., 2009	Explored auto classifiers using commit messages, file types, and authors information to classify the seven large commits. Authors extracted discriminative key-words for each of them using Bayesian type learning on word frequency.
SD7	Mauczka et al., 2012	Proposed a classifier for classifying commits into three categories. Combined keywords from Mockus and Hassan, and extended the keywords list systematically.
SD8	Fu et al., 2015	Proposed a classifier of changes based on semiLDA utilizing the Keywords extracted by Mauczka et al.
SD9	Ding et al., 2015	Classify architectural changes from the developers communication based on SACCS explanation (Williams and Carver, 2010)
SD10	Yan et al., 2016	Proposed a DPLSA classifier using keywords of Mauczka et al. However, their technique **can detect tangled tasks** within a single commit message.
SD11	Levin and Yehudai, 2017	Proposed a commit classifier **utilizing Fluri's 48 code change taxonomy**. This study also employ keywords as an alternate model for classification.
SD12	Gharbi et al., 2019	Explore an active learning technique for the detection of tangled changes within a single commit using keywords. This technique also try to auto label the dataset with the help of Libcat.
SD13	Mondal et al., 2019	Explore architectural change classification using SemiLDA, LLDA and DPLSA.
SD14	Mariano et al., 2019	Authors proposed an improvement approach to classify commits using XGBoost. They **added three code change features** with 20 keywords and 48 Fluri et al. taxonomy of code change.
SD15	Wang et al., 2019a	Study a method to classify large review commit into nine categories: bug fix, resource, feature, test, refactor, merge, deprecate, auto, and others. Various types of features are used for the classifier model.
SD16	Hönel et al., 2020	This study introduces **source code density — the ratio of functional code and gross code size**, to classify the change commits into three classes.

- Types of meaningful changes defined by the literature.
- Capability of the existing techniques in classifying the architectural changes specifically.
- Properties used for constructing the classification models and their efficiency.
- How change instances are annotated into various types.
- Capability of detecting tangled changes within a single task.
- Validity of the change prediction outcome by the developers/experts.
- Types of classification algorithms are used for categorization and their efficiency.
- Relationships among the proposed techniques.

8.1. Architectural change type centric analysis

Existing change classification studies and their associated change categories are summarized in Table 9. From our extensive analysis, we realize that the literature mostly focused on two variations in explaining the change categories (of DEVEM activities). These variations are: (i) Swanson three classes SD0 (Swanson, 1976), and (ii) Mockus three classes SD1 (Mockus and Votta, 2000). Some of the studies considered the conceptual explanation of Swanson (1976) classes: adaptive (*A*), perfective (*P*), and corrective (*C*). Those are aligned with the explanation of the three of the architectural changes are SD4 (Hindle et al., 2008), SD6 (Hindle et al., 2009), SD7 (Mauczka et al., 2012), SD8 (Fu et al., 2015), and SD10 (Yan et al., 2016). In contrast, studies SD3 (Hassan, 2008), SD12 (Gharbi et al., 2019), SD11 (Levin and Yehudai, 2017), SD14 (Mariano et al., 2019), and SD16 (Hönel et al., 2020) followed Mockus definition; these approaches cannot be used directly for grouping the architectural changes. Here, we follow the definition of Williams and Carver (2010) and Ding et al. (2015) as the baseline concepts for the four types of architectural change (aligned with Swanson's definition in many ways). Following this, we have mapped the classes of usual change with the classes of architectural change based on their explanation, which is shown in Table 9. Some studies split these categories into further groups, such as general maintenance (*GM*) and quality change (*Nf*), which are aligned with the preventive class of architectural changes (SD3 and SD6) as per conditions. However, we notice that only Mondal et al. (SD13) explored automatic techniques adopting SD8 and SD10 for determining four architectural changes explicitly. Overall, during analyzing the change types, some of the studies either envision or figure out some crucial information (displayed in *Key finding* column) related to software maintenance, such as the preventive category was found to be linked with longevity, efficient test case design, risk estimation, the possibility of recommending future changes, etc.

8.2. Feature centric analysis for categorization

Our surveyed studies emphasize change categorization in the descriptive (concrete and implemented systems) contexts, and features utilized in the change classification processes are two types: (i) textual description and (ii) source code properties. Most of the studies consider the textual description, which is less

Table 9
Code change categories and alignment with architectural change.

Study	Change types	Link to architecture change	Key finding
SD1 (Mockus and Votta, 2000)	Five: A_m, C, P_m, Ip, Ns	A, P are contradictory with Swanson. One category is aligned	Linked P_m with longevity
SD2 (Dagpinar and Jahnke, 2003)	Three: P, C, PV	A, P are consider one type. Two categories are aligned	
SD3 (Hassan, 2008)	Three: BF, FI, GM	FI is contradictory with Swanson. Two categories are aligned	Developers are also confused for some types
SD4 (Hindle et al., 2008)	Seven: Fa, Nf, legal, maintenance, module manage, SCS, meta-program	As per explanation two of them are aligned	Many of them are architectural
SD5 (Hattori and Lanza, 2008)	Four: forward engineering; and reengineering, corrective engineering and management activities	As per explanation three of them are aligned	Separate combination of features
SD6 (Hindle et al., 2009)	Seven: Fa, Nf, legal, maintenance, module manage, SCS, meta-program	As per explanation two of them are aligned	Author information can predict types significantly
SD7 (Mauczka et al., 2012)	Three Swanson classes: A, C, P	As per explanation two categories are aligned	Subcat tool
SD8 (Fu et al., 2015)	Three Swanson classes: A, C, P	As per explanation two of them are aligned	Many commits are difficult to define types
SD9 (Ding et al., 2015)	Four classes: A, C, P, PV	Four archi categories	Change is frequent
SD10 (Yan et al., 2016)	Three Swanson classes: A, C, P, Ns	As per explanation two of them are aligned	Same as SD8
SD11 (Levin and Yehudai, 2017, 2016)	Three Mockus classes: A_m, C, P_m	As per explanation one category is aligned	Developer's profile, balanced team, development process anomalies
SD12 (Gharbi et al., 2019)	Three: A, C, P. A is contradictory with Swanson	As per explanation one category is aligned	Different performance metrics
SD13 (Mondal et al., 2019)	Four classes: A, C, P, PV	Four archi categories	New model
SD14 (Mariano et al., 2019)	Three Mockus classes: A_m, C, P_m	As per explanation one category is aligned	Same as SD11
SD15 (Wang et al., 2019a)	Nine categories: bug fix, resource, feature, test, refactor, merge, deprecate, auto, and others.	As per explanation a few of them are aligned	
SD16 (Hönel et al., 2020)	Three Mockus classes: A_m, C, P_m	As per explanation one category is aligned	Same as SD11

Table 10
Feature types used for categorization.

Study	Text	Code	Description	Baseline
SD1	Y		Keywords, words for P have priority	
SD2		Y	Statements and class properties	
SD3	Y		Keywords, words for C have priority	Mockus (SD1)
SD4	Y		Keywords	
SD5	Y		Keywords, first appeared word has the priority	
SD6	Y		Keywords, author and file info	
SD7	Y		Keywords	Hindle (SD4)
SD8	Y		Keywords	Mauczka (SD7)
SD9	Y		Explanation from SACCS	Williams and Carver (2010)
SD10	Y		Keywords	Mauczka (SD7)
SD11	Y	Y	Keywords, 48 code types	Fu (SD8), Fluri and Gall (2006)
SD12	Y		Keywords	
SD13	Y		New Keywords	
SD14	Y	Y	20 Keywords, 51 code features	Fluri and Gall (2006)
SD15	Y	Y	Keywords, author and code	
SD16		Y	12 code density attributes	Levin (SD11)

dependent on architectural abstractions, views, component models, and implementation language. Some proposed techniques in these studies leveraged source code properties and change operations, which are mostly dependent on the previously mentioned aspects. A few of them also explore author information and file types (Hindle et al., 2009). Furthermore, some of the proposed techniques combined both source code and textual description. However, we do not find any sophisticated technique for measuring the textual features like the strength of sentiments of a sentence or a word (Esuli and Sebastiani, 2017). These features are keywords, mostly extracted by straight-forward frequency count processes. Various source code properties used in the change categorizing processes are: identifiers of the changed statements, AST, change operation types, method body, role of the methods within a class, code density, and code type. For these properties, the work of Dagpinar and Jahnke (2003), Fluri and Gall (2006), and Hönel et al. (2020) are prominent. Other works either directly used a subset of features from Fluri et al. or extended a bit (SD11 Levin and Yehudai, 2017, SD14 Mariano et al., 2019). Among the textual features, key-words extracted by Mockus and Votta (2000), Hindle et al. (2009, 2011), and Mauczka et al. (2012) are prominent. These keywords are extracted manually or through a frequency count process. Other studies in this group either directly used them or extended them a bit (SD3 Hassan, 2008, SD8 Fu et al., 2015, SD10 Yan et al., 2016, SD12 Gharbi et al., 2019). However, SD13 extracted a new set of keywords for four architectural change types. Overall, the list of feature types and relationships among the studies based on them is presented in Table 10.

In our feature-centric analysis, we found that a few techniques prioritize the extracted keywords for a few specific categories (Mockus and Votta, 2000; Hassan, 2008; Hattori and Lanza, 2008). When features from the multiple classes appear, SD1 prioritizes the perfective keywords, SD3 emphasizes the corrective keywords, and SD5 prioritizes the first keyword. However, many techniques (SD3, SD5, SD8, SD10) that only consider keywords cannot predict all change commits (X mark of *All Cover* column of Table 11 indicates this). The percentages of skipped samples are 20%–28% of the experimental datasets. Finally, one noticeable phenomenon is that source code properties are used recently compared to textual features for change classification (since 2017). Since the existing classification techniques based on code properties do not consider architectural changes explicitly, an empirical study should be conducted for the effectiveness of the properties for enhancing classification techniques.

8.3. Analysis regarding validity of the prediction of change intentions

In the surveyed papers, we have observed that defining the purposes and types of many change samples is ambiguous even to the developers. It appears that concerns in annotating the experimental dataset and validating the predicted samples are important in the change classification research. Therefore, we identify the studies that have validated the classification outcome by the developers or experts. Knowing this analysis provides much confidence to the audiences of the classification studies. In this regard, we found eight studies that evaluate the outcome by a small number of developers (SD1, SD3, SD7, SD8, SD10, SD11, SD15, and SD16). This list is shown in *Developer involved* column of Table 11. These studies have reported the performance as the agreement among the developers with *Kappa* coefficient. We notice that the best technique proposed in the study SD16 that considers three maintenance classes achieved 78% agreement on the predicted samples. However, other techniques evaluated the outcome by traditional classification metrics (such as accuracy and $F1$ score) against their own annotated samples.

8.4. Analysis regarding tangled changes within a change task

In practice, developers implement multiple purposes within a single change task (or a commit) called tangled changes. Therefore, for a more accurate representation of change types, detecting tangled changes within a change task (if they exist) is essential. Four of the studies focused on supporting classifying tangled changes in a single change task: SD10, SD12, and SD15, which are listed in the *Tangled Change* column of Table 11. These studies leveraged keywords to construct the classification models (SD15 considers both keywords and source code). Other studies only focused on one purpose in a single change task or commit. Furthermore, a handful of techniques (SD1, SD3, SD5) ignore detecting tangled classes by prioritizing the presence of the keywords, as we have discussed in Section 8.2. However, extracting semantic information from the change operations (in the source code) might help to address the multiplicity challenge (Wang et al., 2019b).

8.5. Analysis regarding annotating topics for describing change concept

In the conducted studies for change classification, we have noticed frequent concerns about how experimental change samples are annotated. Usually, the change purposes and intentions are extracted from the commit messages and comments; such a message is shown in Section 9.2.1. Additionally, the literature has reported various challenges in explaining the ambiguities in the concepts of the change purposes among the developers. This is because of the presence of tangled changes within a single task and time-consuming manual analysis of the source code. Consequently, it was impossible to experiment with a large collection of representative samples to devise a more efficient technique and a more intuitive conclusion. Yet, to address the challenges, existing works attempted to deploy various annotation processes: annotation by a single author, annotation by multiple authors, annotation by multiple developers (Yan et al., 2016), and annotation by semi-automatic approaches (Gharbi et al., 2019). These approaches are listed in the *Annotation* column in Table 11. As we have noticed, SD12 partially proposed automatic annotation; SD8, SD11, SD13, SD15, and SD16 employed multiple authors; SD10 employed multiple developers for annotation; the rest of the studies considered one author for annotating experimental samples. Most of these studies have focused only on usual software changes. However, various concerns in the annotation process would be more acute for architectural change categorization, and the process requires the involvement of experienced developers or experts. In this regard, we envision that an automatic annotation technique would enable experimenting with a good collection of samples for a more conclusive outcome.

8.6. Analysis regarding algorithm types and efficiency

In this section, we bisect the change classification studies from various algorithmic perspectives and their corresponding performance. Following the perspectives, we can group them into four types (Lai et al., 2015): (i) manual process, ii) discriminating feature selection model (DFM) where textual features are engineered by human such as removing stop words, (iii) traditional machine learning (ML) models where numerical feature values are calculated from tokens (either from source code or textual descriptions), and (iv) non-traditional (NT) way (word presence). This information is displayed in *Process* column of Table 11. However, two studies manually categorize the changes: SD4, and SD9. The rest of the studies propose automatic techniques for change classification. As can be seen, most of the automatic techniques leveraged the traditional machine learning (SD2, SD6, SD11, SD14, SD15, SD16) algorithms. Another four studies applied the discriminative topic models (SD8, SD10, SD13). Surprisingly, four studies proposed some non-traditional techniques that show moderate performance close to machine learning techniques (SD1, SD3, SD5, SD7). This indicates that an efficient feature model is really important for a classification process rather than a complex classifier. However, we have not found any studies that explore bag-of-words (BOW) model where bigrams, n-grams words or so on is employed, or deep learning models.

From the performance table, it is noticeable that *Kappa* coefficient is important for measuring the change classification performance where validation is involved in real users or developers. In this case, the kappa statistic is frequently used to test the interrater reliability. However, with combined features of keywords and source-code, SD14 (Mariano et al., 2019) achieves the highest 77% accuracy and 64% *Kappa* coefficient by employing XGBoost tree learning for the typical changes. On the other hand, code density based approach achieves 88% *Kappa* for a single project (SD16). Still, the code density features do not cover change operations for some higher granular levels that we have discussed in Section 6.7, and may not be promising for architectural changes. From our performance analysis of the change classification studies, we notice that no automatic technique crossed 76% $F1$ and 64% *Kappa* performance measures after pruning some change samples with weak information.

From the survey, we realize some critical concerns of the existing studies. One concern is that at least eight studies cannot

Table 11
Dataset, validation and process type for categorization.

Study	Tangled change	Developer involved	Annotation	Methodology	Performance	All cover
SD1		Y	One author	Non-traditional	>50% *Kappa*	
SD2			Not mentioned	Regression model	62%–99% R^2	∼
SD3		Y	One author	Non-traditional	64% *Kappa*	X
SD4			One author	Manual		
SD5			Not-mentioned	Non-traditional	76% *F*1	X
SD6			One author	Machine learning	70% ROC	
SD7		Y	One author	Non-traditional	61% *Kappa*	∼
SD8		Y	Multiple annotators	DFM	70% *F*1, avoiding 20% samples	X
SD9			Not mentioned	Manual concept	NA	
SD10	Y	Y	Multi-developer	DFM	76% *F*1 AVG	X
SD11		Y	Two authors	Machine learning	70.7%*F*1, 63% *Kappa*	X, NI filter
SD12	Y		Manual, auto annotation	Active learning	45.8% *F*1	X
SD13			Two authors	DFM	45% *F*1	
SD14			One author	Machine learning	77%*Ac*, 64% *Kappa*	X, NI filter
SD15	Y	Y	Multiple	Machine learning	48%–67% *F*1	∼
SD16		Y	Two authors	Machine learning	70.7%*Ac*, 54% *Kappa*	X, NI filter

NI filter — filtered the non-informative samples.

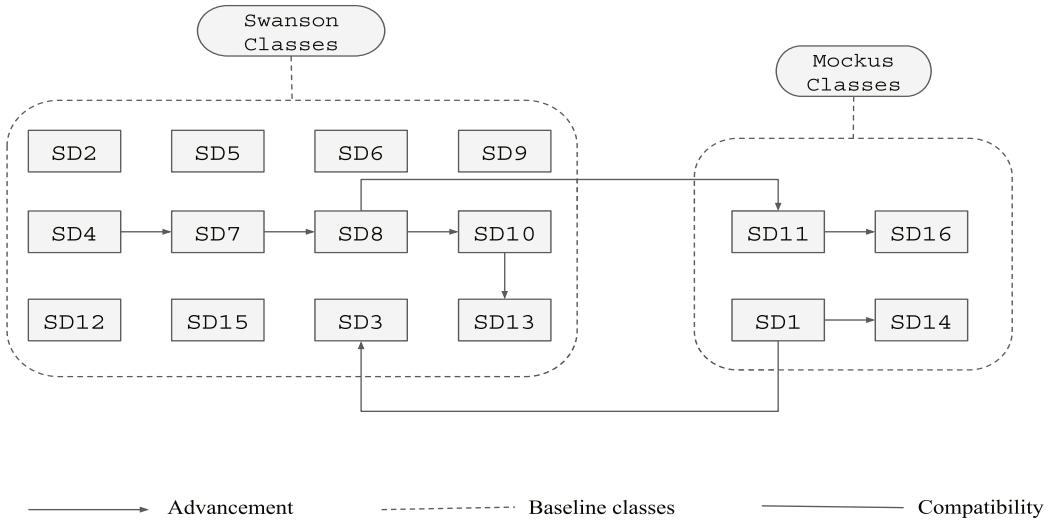

Fig. 4. Change classification relation.

predict all change samples reliably because they pruned commits having weak information (i.e., non-informative messages) from the experimental dataset. Another important point is that change grouping techniques are mainly dependent on analyzing developer discussions or commit messages. Then again, empirical studies found that many change commit messages have ambiguous descriptions (Herzig et al., 2013) (along with the tangled changes described previously). Thus, we envision that the existing automated techniques are not applicable to a real-world scenario reliably without resolving these concerns. Therefore, a rigorous focus is required to handle the message triad — ambiguous messages, non-informative messages, and messages containing tangled concepts for real-world applications of automated change categorization techniques.

8.7. Relationship analysis among the proposed techniques

We have analyzed the relationships among the proposed classification studies based on the concepts used to define the classes and how the proposed models are enhanced from study to study. These relations are presented in Fig. 4. We discuss the relations based on the explicit information available in the studies. According to their discussion, SD11, SD16, SD1, and SD14 follow the change type definition of Mockus and Votta (2000). The rest of the studies follow the change type definition of Swanson (1976). However, the studies SD4, SD7, SD8, SD10, and SD13 are subsequently advanced based on the feature models (keywords). Whereas SD8, SD11, and SD16 are subsequently advanced to improve the accuracy. Although SD11 and SD16 proposed source code feature, they also employ the common keywords model of SD8. Another group of studies, SD1, SD3, and SD14, are explored and enhanced based on the same keywords model. However, there might be implicit relations among some studies which we could not assume explicitly.

9. Future research possibilities on architectural change detection and categorizing

From our analysis on the existing architectural change detection and classification research, we feel the necessity of further research in the directions discussed in the following paragraphs. While our discussion is mostly based on our findings from Sections 5, 6, 7, and 8, following sections discuss some yet unexplored or rarely explored areas in change detection and classification.

9.1. Research opportunities in architectural change detection

From the perspective analysis of the existing studies, we have identified a few of the challenges yet to be solved for change instance detection those are mostly for descriptive architecture.

9.1.1. Light-weight tool for change detection

As we have discussed in Section 6.1, the techniques that require bytecode generation, ADDL coding, and manual data extraction are infeasible (at least for large projects) for processing hundreds and thousands of change revisions for architectural change detection. For example, as reported by Lutellier et al. (2015), ARC took 14 h, BUNCH took 24 h, and LIMBO and ZBR were timeout (or exceeding memory) to recover architecture for a single revision of the Chromium software. The most promising tool, ARCADE leverages some of these techniques, and it may take (with ARC) at least 1400 h to detect architectural changes for 100 revisions of Chromium. Therefore, lightweight techniques that do not rely on those steps would be more feasible for the development team to deploy for design review and document generation frequently. Even the techniques that used the AST or Graph from bytecode require string/identifier matching at the final stage. As an alternative to these, the *diff*[11] tool and the VCS APIs (such as GitPython[12]) provide code-change information as addition and deletion of strings with other useful information. In addition, a recent tool called PyDriller (2020) provides more structured information about involved classes/files and locations of code change. Processing this information with directory and file name would be a promising starting point of lightweight technique development for change detection and design decisions/requirements association.

9.1.2. Change instance detection for the advanced systems

In recent years, advanced platforms such as JPMS (Black, 2018) and Python modules are emerging that focuses on creating and configuring concrete modules to resolve various concerns (including security Ghorbani et al., 2019) related to advanced system development and deployment. Therefore, researchers should focus on more efficient architectural change detection metrics for systems designed by such advanced development frameworks. For example, for Object-oriented systems, existing metrics such as A2A (Garcia et al., 2013) and C2C (Le et al., 2015) extract high-level architecture (descriptive) considering methods, classes, and packages. However, JPMS defines and configures concrete modules (and various access rules) along with those entities. Thus, architecture extraction for JPMS systems requires different strategies and effectiveness measures. Even, change metrics for the prescriptive architectural models require defining the new dimensions of such systems.

9.1.3. Change prediction from natural language document

An empirical study found that more than 88% of projects document architecture in natural language. However, only one study (Mondal et al., 2019) explores possible change prediction from the natural language document deploying discriminating feature modeling technique which $F1$ score is 60%. Therefore, it is really important to conduct more investigations on this direction to develop reliable tools. In this regard, semantic meaning based prediction techniques using PLSA (Yan et al., 2016), RNN-LSTM (Sutskever et al., 2014) or BERT (Yu et al., 2019) model could be a starting point to explore.

9.1.4. Benchmark tool supporting various abstractions and views

Change detection techniques that focused on prescriptive architecture mostly covered high-level abstraction of software architecture. Arguably, the other two abstraction levels might not be promising for the prescriptive architecture because it does not represent operations of the internal components before the implementation. However, all three abstraction levels are valuable for descriptive architectures. In addition, multiple architectural views represent complementary information for a system (Kazman and Carriere, 1998). That said, change detection techniques at those abstraction levels and views are specific to projects' maintenance activities. Moreover, configurations, constraints, and common elements of documenting architecture are not rigorously considered for automatic change detection. Therefore, a benchmark tool with visual analytic support combining the existing techniques would be valuable for the developers and researchers for surfing from a single place. For example, EVA (Nam et al., 2018) is an excellent tool for recovering a descriptive architecture and detecting changes based on ACDC/MoJo (Tzerpos and Holt, 2000) and ARC (Garcia et al., 2011) techniques with visual support. We envision that exploring EVA or ARCADE (Schmitt Laser et al., 2020) tool would be a good starting point for adding supports for various views and other abstractions.

9.1.5. Semantic change information generation of the changed elements

We found a clear gap in producing the semantic meaning of change operations and values of the existing techniques based on which architectural change instances are properly defined. For instance, this semantic meaning may contain the *Consistency* and *Tracking* information (Schneidewind, 1997). However, this would be different for the three abstraction levels. It would also map and synchronize operations with the requirement/design engineering process. Furthermore, design decisions (and requirements) associativity is crucial for this purpose. A handful of studies focused on this direction but did not cover various aspects of architectural changes. For example, Bouwers et al. (2011) systematically define the CB metric that indicates changes in decomposing systems into components and in the number of uniform components. Le and Medvidovic (2016) defined six metrics such that; changes in these metrics indicate a change in the code design quality. We realize that researchers should explore more to define and validate the semantic meaning of the architectural change metrics.

Moreover, only a few studies focus on architectural change detection of legacy systems developed on functional and structural programming. Semantic change information generation of the changed code for them would be much more complicated. Thus rigorous investigations are essential considering the semantic relation for architectural change of these systems like Wang et al. (2019b) explored for the atomic changes. Please note that many studies are available for semantic view generation (Kazman and Carriere, 1998; Nam et al., 2018; Schmitt Laser et al., 2020; Bowman et al., 1999), whereas our suggestion is to look for semantic change metrics and their effectiveness measure (in this paper). However, architectural quality metrics (such as *Change Scenario Robustness*) proposed by Sehestedt et al. (2014) could be one of the baselines to start exploring in this direction. Some of the metrics' qualities defined by the 1061 IEEE Standard should also be focused (Schneidewind, 1997).

9.1.6. Better technique to extract model for architectural change detection

Our survey shows that change detection approaches are heavily dependent on defining architectural views (and models) and extracting architectural clusters. Because, view definition and cluster extraction are essential for architectural change detection if modules or design elements are not documented. However, several excellent architecture extraction tool exists (Garcia et al., 2013, 2011; Kazman and Carriere, 1998; Nam et al., 2018) such as EVA, and ARCADE based on several clustering techniques. Those can be modified and utilized for change instance detection. These steps are also required to generate a semantic architecture from the raw code elements. However, the existing approaches (Le

11 git-scm.com/docs/git-difftool.

12 gitpython.readthedocs.io/en/stable/.

et al., 2015; Garcia et al., 2013) for extracting various architectural models and clusters either perform moderately (in terms of accuracy) or require manual efforts (Bowman et al., 1999; Kazman and Carriere, 1998). Additionally, detecting changes based on generating clusters is highly unreliable because a few renaming operations in the next version would produce a drastically different (Nakamura and Basili, 2005) cluster. Moreover, validation of the outcome of the model extraction approaches is challenging, especially for large systems (Nakamura and Basili, 2005). Furthermore, most of the automated techniques extract modules that are not semantic (Cai et al., 2013). Therefore, future investigations towards better extraction of modules and validation focusing change instances can add value to the existing knowledge.

9.1.7. Versioning scheme focusing architectural maintenance

From the survey, we have realized that the existing system's versioning scheme is not strongly related to the extent of architectural change. Therefore, change detection techniques should be explored for developing a versioning scheme focusing on architecture-centric maintenance and development. Such a technique can emphasize to develop a database for systematically defining and tracking change instances in two types of architecture and three level of abstraction. This approach will also facilitate discovering maintenance knowledge-base accordingly.

9.2. List of open questions and current status for change classification

Some of the major open research questions for architectural change categorization in the literature are:

9.2.1. Message triad handling

From our survey, we observe that change categorization techniques are mainly dependent on analyzing developer discussions or commit messages. Many of them pruned commits having weak information (i.e., non-informative messages) from the experimental dataset. Moreover, empirical studies found that many change commit messages have ambiguous descriptions (Herzig et al., 2013). Therefore, rigorous focus is required to handle message triad — ambiguous messages, non-informative messages and messages containing multiple/tangled concepts for real-world applications of automated techniques. Overall, more study is required to cope with the following situations —

> *Non-informative Message (NI):* A message is non-informative if no particular information is presented about the specific reason for the change. An example message is — *some Pr Changes*. This message specifically cannot describe what type of change it is.
>
> *Ambiguous Message (AM):* A message is ambiguous if a long discussion is provided and can have multiple meanings or code change does not strongly reflect the description. For example, *update p2p logging* might indicate either perfective or preventive.
>
> *Message with tangled concepts:* A message might have multiple unrelated change information (called a tangled message). An example message is — *update p2p logging & fetch headers based on td*. It contains two different features — one is involved with logging, and another is involved in fetching headers information.

Code change relations at the architecture level might handle these situations. However, usual code properties have been explored, but most of them are not specifically architectural relations.

9.2.2. Benchmark dataset creation

After analyzing the existing studies, we feel the necessity of constructing a benchmark dataset containing architectural changes with proper annotations. The dataset should also contain groups of samples representing the abstraction levels. However, a few of the currently available datasets have small number of samples. Moreover, labeling and annotating change samples are the fundamental steps for creating a dataset. Although bug fixing can be directly annotated as corrective, according to the literature, labeling change category is ambiguous even by the developers (Mockus and Votta, 2000; Fu et al., 2015; Hindle et al., 2011). Nevertheless, Herzig et al. (2013) found that the developers misclassify 33.8% of the bug reports in the issue tracking system. To resolve this issue, involvement of experts on software architectures and experienced developers is required. Existing studies attempted to solve it by multiple authors as annotators or developers who change the code. However, developers often cannot come to a consensus while determining the change categories. Such disagreements would be more acute when two types of architectures and three levels of abstractions are analyzed at the same time (Rástočnỳ and Mlynčár, 2018). This could be a bottleneck for the progressive research works in this area. Further explorations involving the responsible architects and expert developers of the subject systems can be helpful towards resolving these concerns.

9.2.3. Slicing tangled changes

Existing studies reported the existence of tangled commits during software evolution (Wang et al., 2019b). A tangled commit contains two or more unrelated changes in the code-base (Hindle et al., 2011). An efficient technique must detect and separate tangled intentions in a single change task (or commit). Some techniques for tangled commit separation and multiple label detection exist only for local or atomic changes, however the performance of those techniques have not reached to a considerable level yet (Yan et al., 2016; Wang et al., 2019b). Besides, available studies did not focus on separating atomic and architectural change. Thus many analyses dependent on those techniques may not produce reliable consensus.

9.2.4. Efficient techniques for four types of changes

Only one study explored auto classifier for four types of architectural changes based on textual properties. However, the performance of the auto classifier is not promising (Mondal et al., 2019). Further studies should be done on exploring the feasibility of the existing commit classification algorithms for architectural change. Towards this direction, two existing studies can be considered as the closest to architectural change; one study deals with large commit classification (Hindle et al., 2009) and the other study is on tactic/design solution classification (Mirakhorli et al., 2012) which are subsets of architectural implementation/change in the code-base. These techniques could be leveraged in this area to some extent. Furthermore, open questions regarding which properties between source code and textual description (or combination of both) will produce a more efficient outcome (and what are those properties) remain unanswered.

Moreover, the outcome of the explored techniques for change classification are not decent even after excluding the poor samples. A more efficient technique would be reliable for applying change classification in various real-world scenarios. However, the deep neural word embedding model (Lai et al., 2015) is showing outstanding performance for various other domains. As such, adapting a deep neural model for classifying changes could be a good starting point. Furthermore, properties of architectural components are not explored explicitly to improve detecting a change type. Researchers should also focus in that direction.

9.2.5. Tool for classifying changes occurred at various granulates

Architectural change operations happen at various granulates. Developers need effective strategies for reviewing the changed code on these granulates. That said, we realize that relevance of the existing change classification techniques should be explored on all of these abstraction levels. Furthermore, it is essential to produce empirical data on the impact of change types and their categorization of these levels. Change classification should also focus on both descriptive and prescriptive types. The existing change classification techniques lag behind in classifying changes in these two types. We feel the necessity of further research in this direction. Researchers should emphasize on this direction.

9.2.6. Benchmark tool for evaluating classification techniques

We feel the necessity of a benchmark tool that can facilitate (Weka like) performance comparison of the existing techniques with various configurations and features. Such a tool will help in advancing research in this area.

9.2.7. Empirical study for design summary generation for change tasks

When a new release of a software project is issued, development teams produce a release note, usually included in the release package or uploaded on the project's website and internal log system for the development team. Such a release note summarizes the main changes that occurred in the software since the previous release, such as new features, bug fixes, restructuring the design components and changes to licenses under which the project is released. However, a number of works focused on generating typical release summary from historical change database, but those are not the architectural summary. To eradicate this limitation, we should conduct more empirical studies employing existing change detection and classification techniques for architectural design summary generation and architectural model synchronization like Rástočnỳ and Mlynčár (2018) study. Even we should enhance existing change detection and categorization techniques on the basis of the design summary generation constraints.

9.2.8. Automatic annotation of change topics

From our survey, we notice that annotating an architectural change instance to a specific or multiple categories is challenging and ambiguous. A few of the techniques attempted to develop a method for that purpose (Hindle et al., 2011), but those do not consider architectural changes explicitly. Therefore researchers can focus on automated techniques for annotating a sentence (into a relevant concept) within a change description for the particular type of change. We believe such a technique would also enhance the automatic design summary generation process.

10. Discussion

This section answers the three research questions that we have mentioned in the introduction on the basis of our survey.

Answer to RQ1: From the systematic review process of the existing studies, we found that architectural change detection techniques can be grouped by considering various attributes and concerns. The major attributes and concerns are which data is used as the input for the detection process, how the change metrics are associated with design decisions, how architecture and its change operations are defined, model and code element's properties are considered, software abstraction levels are supported, various views are considered, and types of architecture are adopted, and how the effectiveness of the metrics are measured. Moreover, the processes of extracting architectural models and document are dependent on external techniques. However, researchers have explored well enough almost all of the perspectives except extracting meaningful change metrics, information of changed elements of the systems and effectiveness measurement of the change metrics. Finally, the utilization of lightweight tools that are less dependent on human intervention and byte code generation for input data is not explored enough.

Answer to RQ2: From the analysis of the existing studies, we found that architectural change classification approaches can be analyzed by considering various attributes and concerns. The major attributes and concerns of the change classification approaches are which types of changes are focused, what are the variations in defining a change type, which types of features are considered, what annotation and validation processes are used, and how tangled purposes within a single task are handled. Also, they can be grouped by the types of learning models for constructing a classifier. However, only a few of the studies have focused on the challenges of the annotation and validation process, and detecting tangled changes. Other concerns such as effectiveness measures in the real-world scenarios are covered little by the literature (particularly focusing architectural change).

Answer to RQ3: We have also answered our third research question (**RQ3**) by discussing a number of future research possibilities in architectural change detection and classification. Section 9 contains this discussion.

11. Conclusion

In this paper, we present our investigation of the existing architectural (both prescriptive and descriptive) change detection and classification techniques. During the investigation, we compare the claims by the authors of those studies based on various aspects necessary to consider while working with architecture-centric software development. We found that most of the existing techniques require either manual intervention or input generation tools that many cases, are infeasible to process hundreds or thousands of change revisions in a single software release. Among the existing studies, AbuHassan and Alshayeb (2019) cover detecting changes of three types of prescriptive architectural models. In comparison, for the descriptive architecture, a study by Behnamghader et al. (2017) focused on more appropriate architectural abstraction. Yet, from our analysis of change detection techniques, we realize that the design decision associativity of the change elements and operations should be defined and validated for the design review process. Moreover, little focus has been given to defining semantic change metrics for non-object-oriented and legacy systems. We realize that a benchmark tool that can detect semantic architectural changes at various abstraction levels and views will add much value to the researchers and practitioners. Furthermore, most of the available detection metrics heavily rely on extracting architectural models or clustering program entities. For the descriptive architecture, ARCADE (Behnamghader et al., 2017; Schmitt Laser et al., 2020) is the most promising tool for architectural change analysis that deploys MoJo, MoJoFM, A2A, C2C, and ID-SD change metrics. However, the effectiveness of the majority of the metrics was not measured due to various challenges such as the availability of experts (Jansen et al., 2008). The question remains open on the reliability of the clustering-based change detection because a few renaming operations in the next version of a software source code would produce a drastically different cluster (Nakamura and Basili, 2005).

Among the existing change classification studies, we found that the techniques proposed by Hindle et al. (2009) and Mauczka et al. (2012) are more promising for software change classification. On the other hand, to handle tangled changes, the proposed classification model by Yan et al. (2016) is the most promising and adaptable to architectural change classification. Still, the

textual properties are the essential elements of the existing classification techniques. However, one of the major concerns that remain unresolved in this domain is when a change task description contains non-informative, ambiguous and tangled intentions. Future research in this direction can make a significant contribution to the architectural change review process. In this regard, we think that relations of the changed code and their operations should be explored for enhancing explicit architectural change categorization. We also feel that automated techniques should be developed for annotating change topics of a task description. In summary, our survey on both typical and architectural change classification shows that existing automatic techniques are not promising enough to use for real-world scenarios and reliable post analysis of causes of architectural change is not possible without manual intervention. There is also a lack of empirical data to construct an architectural change taxonomy, and further exploration in this direction would add much value to architectural change management. Overall, our survey will help the researchers in this field quickly identifying the existing tools and techniques and find the directions that are yet to explore and make a comparison in this field based on various perspectives of software architecture and maintenance.

CRediT authorship contribution statement

Amit Kumar Mondal: Conceptualization, Methodology, Writing – original draft, Investigation. **Kevin A. Schneider:** Writing – review & editing, Supervision. **Banani Roy:** Reviewing, Supervision. **Chanchal K. Roy:** Reviewing, Supervision.

Declaration of competing interest

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

Acknowledgments

We thank the unknown reviewers for their invaluable suggestions. This research is supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery grants, and by an NSERC Collaborative Research and Training Experience (CREATE) grant, and by two Canada First Research Excellence Fund (CFREF) grants coordinated by the Global Institute for Food Security (GIFS) and the Global Institute for Water Security (GIWS).

References

AbuHassan, Amjad, Alshayeb, Mohammad, 2019. A metrics suite for UML model stability. Softw. Syst. Model. 18 (1), 557–583.

Aghajani, Emad, Nagy, Csaba, Linares-Vásquez, Mario, Moreno, Laura, Bavota, Gabriele, Lanza, Michele, Shepherd, David C., 2020. Software documentation: the practitioners' perspective. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE). IEEE, pp. 590–601.

Ahmad, Aakash, Jamshidi, Pooyan, Pahl, Claus, 2014. Classification and comparison of architecture evolution reuse knowledge—a systematic review. J. Softw.: Evol. Process 26 (7), 654–691.

Alsolai, Hadeel, Roper, Marc, 2020. A systematic literature review of machine learning techniques for software maintainability prediction. Inf. Softw. Technol. 119, 106214.

Alves, Nicolli S.R., Mendes, Thiago S., de Mendonça, Manoel G., Spínola, Rodrigo O., Shull, Forrest, Seaman, Carolyn, 2016. Identification and management of technical debt: A systematic mapping study. Inf. Softw. Technol. 70, 100–121.

Arvanitou, Elvira Maria, Ampatzoglou, Apostolos, Tzouvalidis, Konstantinos, Chatzigeorgiou, Alexander, Avgeriou, Paris, Deligiannis, Ignatios, 2017. Assessing change proneness at the architecture level: An empirical validation. In: 2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW). IEEE, pp. 98–105.

Bachmann, Felix, Bass, Len, Clements, Paul, Garlan, David, Ivers, James, Little, M., Merson, Paulo, Nord, Robert, Stafford, Judith, 2010. Documenting Software Architectures: Views and Beyond, second ed. Addison-Wesley Professional.

Baldwin, Carliss Young, Clark, Kim B., 2000. Design Rules: The Power of Modularity, Vol. 1. MIT Press.

BCEL, 2020. jakarta.apache.org/bcel.

Behnamghader, Pooyan, Le, Duc Minh, Garcia, Joshua, Link, Daniel, Shahbazian, Arman, Medvidovic, Nenad, 2017. A large-scale study of architectural evolution in open-source software systems. Empir. Softw. Eng. 22 (3), 1146–1193.

ben Fadhel, Ameni, Kessentini, Marouane, Langer, Philip, Wimmer, Manuel, 2012. Search-based detection of high-level model changes. In: 2012 28th IEEE International Conference on Software Maintenance (ICSM). IEEE, pp. 212–221.

Bergersen, Gunnar R., Sjøberg, Dag I.K., Dybå, Tore, 2014. Construction and validation of an instrument for measuring programming skill. IEEE Trans. Softw. Eng. 40 (12), 1163–1184.

Bhat, Manoj, Shumaiev, Klym, Hohenstein, Uwe, Biesdorf, Andreas, Matthes, Florian, 2020. The evolution of architectural decision making as a key focus area of software architecture research: A semi-systematic literature study. In: 2020 IEEE International Conference on Software Architecture (ICSA). IEEE, pp. 69–80.

Bi, Tingting, Liang, Peng, Tang, Antony, Yang, Chen, 2018. A systematic mapping study on text analysis techniques in software architecture. J. Syst. Softw. 144, 533–558.

Bi, T., Xia, X., Lo, D., Grundy, J., Zimmermann, T., 2020. An empirical study of release note production and usage in practice. IEEE Trans. Softw. Eng. 1. http://dx.doi.org/10.1109/TSE.2020.3038881.

Black, Nate, 2018. Nicolai parlog on java 9 modules. IEEE Softw. (3), 101–104.

Blanco, Roi, Lioma, Christina, 2012. Graph-based term weighting for information retrieval. Inf. Retr. 54–92.

Bouwers, Eric, Correia, Jose Pedro, van Deursen, Arie, Visser, Joost, 2011. Quantifying the analyzability of software architectures. In: 2011 Ninth Working IEEE/IFIP Conference on Software Architecture. IEEE, pp. 83–92.

Bowman, Ivan T., Holt, Richard C., Brewster, Neil V., 1999. Linux as a case study: Its extracted software architecture. In: Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No. 99CB37002). IEEE, pp. 555–563.

Bruneliere, Hugo, Cabot, Jordi, Jouault, Frédéric, Madiot, Frédéric, 2010. MoDisco: a generic and extensible framework for model driven reverse engineering. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. pp. 173–174.

Buse, Raymond P.L., Zimmermann, Thomas, 2012. Information needs for software development analytics. In: 2012 34th International Conference on Software Engineering (ICSE). IEEE, pp. 987–996.

Cai, Yuanfang, Sullivan, Kevin, 2005. Simon: A tool for logical design space modeling and analysis. In: 20th IEEE/ACM International Conference on Automated Software Engineering.

Cai, Yuanfang, Sullivan, Kevin J., 2006. Modularity analysis of logical design models. In: Proc. of Automated Software Engineering. pp. 91–102.

Cai, Yuanfang, Wang, Hanfei, Wong, Sunny, Wang, Linzhang, 2013. Leveraging design rules to improve software architecture recovery. In: Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures. pp. 133–142.

Carriere, Jeromy, Kazman, Rick, Ozkaya, Ipek, 2010. A cost-benefit framework for making architectural decisions in a business context. In: 2010 ACM/IEEE 32nd International Conference on Software Engineering. 2, IEEE, pp. 149–157.

CDA, 2020. www.dependency-analyzer.org.

Chakroborti, Debasish, Schneider, Kevin A., Roy, Chanchal K., 2022. Backports: Change types, challenges and strategies. In: International Conference on Program Comprehension.

Chapin, Ned, Hale, Joanne E., Khan, Khaled Md, Ramil, Juan F., Tan, Wui-Gee, 2001. Types of software evolution and software maintenance. J. Softw. Maint. Evol.: Res. Pract. 13 (1), 3–30.

Chen, Yih-Fam, Gansner, Emden R., Koutsofios, Eleftherios, 1998. A C++ data model supporting reachability analysis and dead code detection. IEEE Trans. Softw. Eng. 24 (9), 682–694.

Cicchetti, Antonio, Ciccozzi, Federico, Leveque, Thomas, 2011. Supporting incremental synchronization in hybrid multi-view modelling. In: International Conference on Model Driven Engineering Languages and Systems. Springer, pp. 89–103.

Clements, Paul, Garlan, David, Little, Reed, Nord, Robert, Stafford, Judith, 2003. Documenting software architectures: views and beyond. In: 25th International Conference on Software Engineering, 2003. Proceedings. IEEE, pp. 740–741.

Codoban, M., Ragavan, S.S., Dig, D., Bailey, B., 2015. Software history under the lens: A study on why and how developers examine it. In: Proc. of the 2015 International Conference on Software Maintenance and Evolution. pp. 1–10.

Collard, Michael L., Kagdi, Huzefa H., Maletic, Jonathan I., 2003. An XML-based lightweight C++ fact extractor. In: 11th IEEE International Workshop on Program Comprehension, 2003. IEEE, pp. 134–143.

Corazza, Anna, Di Martino, Sergio, Maggio, Valerio, Scanniello, Giuseppe, 2011. Investigating the use of lexical information for software system clustering. In: 2011 15th European Conference on Software Maintenance and Reengineering. IEEE, pp. 35–44.

Cornelissen, Bas, Zaidman, Andy, Van Deursen, Arie, Moonen, Leon, Koschke, Rainer, 2009. A systematic survey of program comprehension through dynamic analysis. IEEE Trans. Softw. Eng. 35 (5), 684–702.

Dagpinar, Melis, Jahnke, Jens H., 2003. Predicting maintainability with object-oriented metrics-an empirical comparison. In: 10th Working Conference on Reverse Engineering, 2003. Proceedings. IEEE Computer Society, p. 155.

Dig, Danny, Comertoglu, Can, Marinov, Darko, Johnson, Ralph, 2006. Automated detection of refactorings in evolving components. In: European Conference on Object-Oriented Programming. Springer, pp. 404–428.

Ding, Wei, Liang, Peng, Tang, Antony, Van Vliet, Hans, 2015. Causes of architecture changes: An empirical study through the communication in OSS mailing lists. In: Software Engineering and Knowledge Engineering. pp. 403–408.

Ding, Wei, Liang, Peng, Tang, Antony, Van Vliet, Hans, Shahin, Mojtaba, 2014. How do open source communities document software architecture: An exploratory survey. In: 19th International Conference on Engineering of Complex Computer Systems. pp. 136–145.

Dong, Xinyi, Godfrey, Michael W., 2008. Identifying architectural change patterns in object-oriented systems. In: 2008 16th IEEE International Conference on Program Comprehension. IEEE, pp. 33–42.

Dragan, Natalia, Collard, Michael L., Hammad, Maen, Maletic, Jonathan I., 2011. Using stereotypes to help characterize commits. In: 2011 27th IEEE International Conference on Software Maintenance (ICSM). IEEE, pp. 520–523.

Durisic, Darko, Nilsson, Martin, Staron, Miroslaw, Hansson, Jörgen, 2013. Measuring the impact of changes to the complexity and coupling properties of automotive software systems. J. Syst. Softw. 86 (5), 1275–1293.

Durisic, Darko, Staron, Miroslaw, Nilsson, Martin, 2011. Measuring the size of changes in automotive software systems and their impact on product quality. In: Proceedings of the 12th International Conference on Product Focused Software Development and Process Improvement. pp. 10–13.

ECL, 2020. www.eclipse.org/epsilon/doc/ecl/.

EMF, 2020. www.eclipse.org/modeling/emf/.

Estublier, Jacky, Leblang, David, Hoek, André van der, Conradi, Reidar, Clemm, Geoffrey, Tichy, Walter, Wiborg-Weber, Darcy, 2005. Impact of software engineering research on the practice of software configuration management. ACM Trans. Softw. Eng. Methodol. (TOSEM) 14 (4), 383–430.

Esuli, Andrea, Sebastiani, Fabrizio, 2017. Sentiwordnet: A publicly available lexical resource for opinion mining. In: Proceedings of the Fifth International Conference on Language Resources and Evaluation.

Fluri, Beat, Gall, Harald C., 2006. Classifying change types for qualifying change couplings. In: 14th IEEE International Conference on Program Comprehension (ICPC'06). IEEE, pp. 35–45.

Fowler, M., 2003. Design-who needs an architect? IEEE Softw. 11–13.

Fu, Ying, Yan, Meng, Zhang, Xiaohong, Xu, Ling, Yang, Dan, Kymer, Jeffrey D., 2015. Automated classification of software change messages by semi-supervised latent Dirichlet allocation. Inf. Softw. Technol. 57, 369–377.

Garcia, Joshua, Ivkovic, Igor, Medvidovic, Nenad, 2013. A comparative analysis of software architecture recovery techniques. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 486–496.

Garcia, Joshua, Mirakhorli, Mehdi, Xiao, Lu, Zhao, Yutong, Mujhid, Ibrahim, Pham, Khoi, Okutan, Ahmet, Malek, Sam, Kazman, Rick, Cai, Yuanfang, et al., 2021. Constructing a shared infrastructure for software architecture analysis and maintenance. In: 2021 IEEE 18th International Conference on Software Architecture (ICSA). IEEE, pp. 150–161.

Garcia, Joshua, Popescu, Daniel, Mattmann, Chris, Medvidovic, Nenad, Cai, Yuanfang, 2011. Enhancing architectural recovery using concerns. In: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011). IEEE, pp. 552–555.

Garlan, David, Bachmann, Felix, Ivers, James, Stafford, Judith, Bass, Len, Clements, Paul, Merson, Paulo, 2010. Documenting Software Architectures: Views and Beyond, second ed. Addison-Wesley Professional.

Gharbi, Sirine, Mkaouer, Mohamed Wiem, Jenhani, Ilyes, Messaoud, Montassar Ben, 2019. On the classification of software change messages using multi-label active learning. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing. pp. 1760–1767.

Ghorbani, Negar, Garcia, Joshua, Malek, Sam, 2019. Detection and repair of architectural inconsistencies in java. In: Proceedings of the 41st International Conference on Software Engineering. IEEE Press, pp. 560–571.

Grundy, John, Hosking, John, 2000. High-level static and dynamic visualisation of software architectures. In: Proceeding 2000 IEEE International Symposium on Visual Languages. IEEE, pp. 5–12.

Gustafsson, Juha, Paakki, Jukka, Nenonen, Lilli, Verkamo, A. Inkeri, 2002. Architecture-centric software evolution by software metrics and design patterns. In: Proceedings of the Sixth European Conference on Software Maintenance and Reengineering. IEEE, pp. 108–115.

Haitzer, Thomas, Navarro, Elena, Zdun, Uwe, 2017. Reconciling software architecture and source code in support of software evolution. J. Syst. Softw. 123, 119–144.

Hammad, Maen, Collard, Michael L., Maletic, Jonathan I., 2009. Automatically identifying changes that impact code-to-design traceability. In: 2009 IEEE 17th International Conference on Program Comprehension. IEEE, pp. 20–29.

Hassan, Ahmed E., 2008. Automated classification of change messages in open source projects. In: Proceedings of the 2008 ACM Symposium on Applied Computing. pp. 837–841.

Hattori, Lile P., Lanza, Michele, 2008. On the nature of commits. In: 2008 23rd IEEE/ACM International Conference on Automated Software Engineering-Workshops. IEEE, pp. 63–71.

Herzig, Kim, Just, Sascha, Zeller, Andreas, 2013. It's not a bug, it's a feature: how misclassification impacts bug prediction. In: 2013 35th International Conference on Software Engineering (ICSE). IEEE, pp. 392–401.

Hewitt, Eben, 2019. Semantic Software Design: A New Theory and Practical Guide for Modern Architects. O'Reilly Media.

Hindle, Abram, Ernst, Neil A., Godfrey, Michael W., Mylopoulos, John, 2011. Automated topic naming to support cross-project analysis of software maintenance activities. In: Proceedings of the 8th Working Conference on Mining Software Repositories. pp. 163–172.

Hindle, Abram, German, Daniel M., Godfrey, Michael W., Holt, Richard C., 2009. Automatic classication of large changes into maintenance categories. In: 2009 IEEE 17th International Conference on Program Comprehension. IEEE, pp. 30–39.

Hindle, Abram, German, Daniel M., Holt, Ric, 2008. What do large commits tell us? A taxonomical study of large commits. In: Proceedings of the 2008 International Working Conference on Mining Software Repositories. pp. 99–108.

Hönel, Sebastian, Ericsson, Morgan, Löwe, Welf, Wingkvist, Anna, 2020. Using source code density to improve the accuracy of automatic commit classification into maintenance activities. J. Syst. Softw. 110673.

Jamshidi, P., Ghafari, M., Ahmad, A., Pahl, C., 2013. A framework for classifying and comparing architecture-centric software evolution research. In: Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering. pp. 305–314.

Jansen, Anton, Bosch, Jan, 2005. Software architecture as a set of architectural design decisions. In: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture. pp. 109–120.

Jansen, Anton, Bosch, Jan, Avgeriou, Paris, 2008. Documenting after the fact: Recovering architectural design decisions. J. Syst. Softw. 81 (4), 536–557.

Kazman, Rick, Carriere, S. Jeromy, 1998. View extraction and view fusion in architectural understanding. In: Proceedings. Fifth International Conference on Software Reuse (Cat. No. 98TB100203). IEEE, pp. 290–299.

Khan, Safoora Shakil, Greenwood, Phil, Garcia, Alessandro, Rashid, Awais, 2008. On the impact of evolving requirements-architecture dependencies: An exploratory study. In: International Conference on Advanced Information Systems Engineering. Springer, pp. 243–257.

Kim, Jungil, Lee, Eunjoo, 2014. The effect of IMPORT change in software change history. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing. pp. 1753–1754.

Kosenkov, Oleksandr, Unterkalmsteiner, Michael, Mendez, Daniel, Fucci, Davide, 2021. Vision for an artefact-based approach to regulatory requirements engineering. In: Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). pp. 1–6.

Kruchten, Philippe, 2004. An ontology of architectural design decisions in software intensive systems. In: 2nd Groningen Workshop on Software Variability. Citeseer, pp. 54–61.

Kurtanović, Zijad, Maalej, Walid, 2017. Automatically classifying functional and non-functional requirements using supervised machine learning. In: 2017 IEEE 25th International Requirements Engineering Conference (RE). IEEE, pp. 490–495.

Lai, Siwei, Xu, Liheng, Liu, Kang, Zhao, Jun, 2015. Recurrent convolutional neural networks for text classification. In: Twenty-Ninth AAAI Conference on Artificial Intelligence.

Le, Duc Minh, Behnamghader, Pooyan, Garcia, Joshua, Link, Daniel, Shahbazian, Arman, Medvidovic, Nenad, 2015. An empirical study of architectural change in open-source software systems. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. IEEE, pp. 235–245.

Le, Duc, Medvidovic, Nenad, 2016. Architectural-based speculative analysis to predict bugs in a software system. In: Proceedings of the 38th International Conference on Software Engineering Companion. pp. 807–810.

Lehman, Manny M., 1996. Laws of software evolution revisited. In: European Workshop on Software Process Technology. Springer, pp. 108–124.

Levin, Stanislav, Yehudai, Amiram, 2016. Using temporal and semantic developer-level information to predict maintenance activity profiles. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 463–467.

Levin, Stanislav, Yehudai, Amiram, 2017. Boosting automatic commit classification into maintenance activities by utilizing source code changes. In: Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering. pp. 97–106.

Li, Yi, Zhu, Chenguang, Rubin, Julia, Chechik, Marsha, 2017. Semantic slicing of software version histories. IEEE Trans. Softw. Eng. 44 (2), 182–201.

Lin, I.-H., Gustafson, David A., 1988. Classifying software maintenance. In: Proceedings. Conference on Software Maintenance, 1988. IEEE, pp. 241–247.

Linberg, Kurt R., 1999. Software developer perceptions about software project failure: a case study. J. Syst. Softw. 49 (2–3), 177–192.

Link, Daniel, Behnamghader, Pooyan, Moazeni, Ramin, Boehm, Barry, 2019. Recover and RELAX: Concern-oriented software architecture recovery for systems development and maintenance. In: 2019 IEEE/ACM International Conference on Software and System Processes (ICSSP). IEEE, pp. 64–73.

LLVM, 2020. llvm.org/.

Lutellier, Thibaud, Chollak, Devin, Garcia, Joshua, Tan, Lin, Rayside, Derek, Medvidovic, Nenad, Kroeger, Robert, 2015. Comparing software architecture recovery techniques using accurate dependencies. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 2. IEEE, pp. 69–78.

Ma, Haohai, Shao, Weizhong, Zhang, Lu, Ma, Zhiyi, Jiang, Yanbing, 2004. Applying OO metrics to assess UML meta-models. In: International Conference on the Unified Modeling Language. Springer, pp. 12–26.

Manadhata, Pratyusa K., Wing, Jeannette M., 2011. An attack surface metric. IEEE Trans. Softw. Eng. 37 (03), 371–386.

Mancoridis, Spiros, Mitchell, Brian S., Chen, Yihfarn, Gansner, Emden R., 1999. Bunch: A clustering tool for the recovery and maintenance of software system structures. In: Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'software Maintenance for Business Change'(Cat. No. 99CB36360). IEEE, pp. 50–59.

Mancoridis, Spiros, Mitchell, Brian S., Rorres, Chris, Chen, Y., Gansner, Emden R., 1998. Using automatic clustering to produce high-level system organizations of source code. In: Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No. 98TB100242). IEEE, pp. 45–52.

Maqbool, Onaiza, Babri, Haroon Atique, 2004. The weighted combined algorithm: A linkage algorithm for software clustering. In: Eighth European Conference on Software Maintenance and Reengineering, 2004. Proceedings. IEEE, pp. 15–24.

Maqbool, Onaiza, Babri, Haroon, 2007. Hierarchical clustering for software architecture recovery. IEEE Trans. Softw. Eng. 33 (11), 759–780.

Mariano, Richard V.R., dos Santos, Geanderson E., de Almeida, Markos V., Brandão, Wladmir C., 2019. Feature changes in source code for commit classification into maintenance activities. In: 2019 18th IEEE International Conference on Machine Learning and Applications (ICMLA). IEEE, pp. 515–518.

Mauczka, Andreas, Brosch, Florian, Schanes, Christian, Grechenig, Thomas, 2015. Dataset of developer-labeled commit messages. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. IEEE, pp. 490–493.

Mauczka, Andreas, Huber, Markus, Schanes, Christian, Schramm, Wolfgang, Bernhart, Mario, Grechenig, Thomas, 2012. Tracing your maintenance work–a cross-project validation of an automated classification dictionary for commit messages. In: International Conference on Fundamental Approaches to Software Engineering. Springer, pp. 301–315.

Mirakhorli, Mehdi, Shin, Yonghee, Cleland-Huang, Jane, Cinar, Murat, 2012. A tactic-centric approach for automating traceability of quality concerns. In: 2012 34th International Conference on Software Engineering (ICSE). IEEE, pp. 639–649.

Mo, Ran, Cai, Yuanfang, Kazman, Rick, Xiao, Lu, Feng, Qiong, 2019. Architecture anti-patterns: Automatically detectable violations of design principles. IEEE Trans. Softw. Eng..

Mockus, Audris, Votta, Lawrence G., 2000. Identifying reasons for software changes using historic databases. In: International Conference on Software Maintenance. pp. 120–130.

Mohagheghi, Parastoo, Conradi, Reidar, 2004. An empirical study of software change: origin, acceptance rate, and functionality vs. quality attributes. In: Proceedings. 2004 International Symposium on Empirical Software Engineering, 2004. IEEE, pp. 7–16.

Mondal, Amit Kumar, Roy, Banani, Schneider, Kevin A., 2019. An exploratory study on automatic architectural change analysis using natural language processing techniques. In: 2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM). IEEE. pp. 62–73.

Monschein, David, Mazkatli, Manar, Heinrich, Robert, Koziolek, Anne, 2021. Enabling consistency between software artefacts for software adaption and evolution. In: 2021 IEEE 18th International Conference on Software Architecture (ICSA). IEEE, pp. 1–12.

Montandon, Joao Eduardo, Silva, Luciana Lourdes, Valente, Marco Tulio, 2019. Identifying experts in software libraries and frameworks among github users. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). IEEE, pp. 276–287.

Myers, Christopher R., 2003. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. Phys. Rev. E 68 (4), 046116.

Nakamura, Taiga, Basili, Victor R., 2005. Metrics of software architecture changes based on structural distance. In: 11th IEEE International Software Metrics Symposium (METRICS'05). IEEE, p. 24.

Nam, Daye, Lee, Youn Kyu, Medvidovic, Nenad, 2018. Eva: A tool for visualizing software architectural evolution. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. pp. 53–56.

Nurwidyantoro, Arif, Shahin, Mojtaba, Chaudron, Michel, Hussain, Waqar, Perera, Harsha, Shams, Rifat Ara, Whittle, Jon, 2021. Towards a human values dashboard for software development: an exploratory study. In: Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). pp. 1–12.

Oreizy, Peyman, Medvidovic, Nenad, Taylor, Richard N., 1998. Architecture-based runtime software evolution. In: Proceedings of the 20th International Conference on Software Engineering. IEEE, pp. 177–186.

Ozkaya, Ipek, Wallin, Peter, Axelsson, Jakob, 2010. Architecture knowledge management during system evolution: observations from practitioners. In: Proceedings of the 2010 ICSE Workshop on Sharing and Reusing Architectural Knowledge. pp. 52–59.

Paixao, Matheus, Krinke, Jens, Han, DongGyun, Ragkhitwetsagul, Chaiyong, Harman, Mark, 2017. Are developers aware of the architectural impact of their changes? In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. pp. 95–105.

Paixao, Matheus, Krinke, Jens, Han, DongGyun, Ragkhitwetsagul, Chaiyong, Harman, Mark, 2019. The impact of code review on architectural changes. IEEE Trans. Softw. Eng..

PyDriller, 2020. github.com/ishepard/pydriller.

Ramage, Daniel, Hall, David, Nallapati, Ramesh, Manning, Christopher D., 2009. Labeled LDA: A supervised topic model for credit attribution in multi-labeled corpora. In: Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing. pp. 248–256.

Rasool, Ghulam, Fazal, Nancy, 2017. Evolution prediction and process support of OSS studies: a systematic mapping. Arab. J. Sci. Eng. 42 (8), 3465–3502.

Rástočnỳ, Karol, Mlynčár, Andrej, 2018. Automated change propagation from source code to sequence diagrams. In: International Conference on Current Trends in Theory and Practice of Informatics. Springer, pp. 168–179.

Roshandel, Roshanak, Hoek, André Van Der, Mikic-Rakic, Marija, Medvidovic, Nenad, 2004. Mae—a system model and environment for managing architectural evolution. ACM Trans. Softw. Eng. Methodol. (TOSEM) 13 (2), 240–276.

Russo, Barbara, Steff, Maximilian, 2014. What can changes tell about software processes? In: Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics. pp. 1–7.

Schmitt Laser, Marcelo, Medvidovic, Nenad, Le, Duc Minh, Garcia, Joshua, 2020. ARCADE: an extensible workbench for architecture recovery, change, and decay evaluation. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1546–1550.

Schneidewind, Norman, 1997. IEEE Standard For A Software Quality Metrics Methodology Revision And Reaffirmation. In: Proceedings of International Symposium on Software Engineering Standards. pp. 278–278.

Sehestedt, Stephan, Cheng, Chih-Hong, Bouwers, Eric, 2014. Towards quantitative metrics for architecture models. In: Proceedings of the WICSA 2014 Companion Volume. pp. 1–4.

Shahbazian, Arman, Lee, Youn Kyu, Le, Duc, Brun, Yuriy, Medvidovic, Nenad, 2018a. Recovering architectural design decisions. In: 2018 IEEE International Conference on Software Architecture (ICSA). IEEE, pp. 95–9509.

Shahbazian, Arman, Nam, Daye, Medvidovic, Nenad, 2018b. Toward predicting architectural significance of implementation issues. In: Proceedings of the 15th International Conference on Mining Software Repositories. ACM, pp. 215–219.

Silva, Marcelino Campos Oliveira, Valente, Marco Tulio, Terra, Ricardo, 2016. Does technical debt lead to the rejection of pull requests? In: In 12th Brazilian Symposium on Information Systems on Brazilian Symposium on Information Systems: Information Systems in the Cloud Computing Era. pp. 248–254.

Steff, Maximilian, Russo, Barbara, 2011. Measuring architectural change for defect estimation and localization. In: 2011 International Symposium on Empirical Software Engineering and Measurement. IEEE, pp. 225–234.

Sutskever, Ilya, Vinyals, Oriol, Le, Quoc V., 2014. Sequence to sequence learning with neural networks. In: Advances in Neural Information Processing Systems. pp. 3104–3112.

Swanson, E. Burton, 1976. The dimensions of maintenance. In: Proceedings of the 2nd International Conference on Software Engineering. IEEE Computer Society Press, pp. 492–497.

Tang, Antony, Lau, Man F., 2014. Software architecture review by association. J. Syst. Softw. 88, 87–101.

Taylor, Richard N., Medvidovic, Nenad, Dashofy, Eric, 2009. Software Architecture: Foundations, Theory, and Practice. Wiley.

Tzerpos, Vassilios, Holt, Richard C., 1999. Mojo: A distance metric for software clusterings. In: Sixth Working Conference on Reverse Engineering (Cat. No. PR00303). IEEE, pp. 187–193.
Tzerpos, Vassilios, Holt, Richard C., 2000. Accd: an algorithm for comprehension-driven clustering. In: Proceedings Seventh Working Conference on Reverse Engineering. IEEE, pp. 258–267.
Uchôa, Anderson, Barbosa, Caio, Coutinho, Daniel, Oizumi, Willian, Assunçao, Wesley K.G., Vergilio, Silvia Regina, Pereira, Juliana Alves, Oliveira, Anderson, Garcia, Alessandro, 2021. Predicting design impactful changes in modern code review: A large-scale empirical study. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR). IEEE, pp. 471–482.
Understand, 2020. www.scitools.com/.
Vasa, Rajesh, Schneider, J.-G., Woodward, Clinton, Cain, Andrew, 2005. Detecting structural changes in object oriented software systems. In: 2005 International Symposium on Empirical Software Engineering, 2005. IEEE, pp. 8–pp.
Verdecchia, Roberto, Kruchten, Philippe, Lago, Patricia, Malavolta, Ivano, 2021. Building and evaluating a theory of architectural technical debt in software-intensive systems. J. Syst. Softw. 176, 110925.
Wang, Song, Bansal, Chetan, Nagappan, Nachiappan, Philip, Adithya Abraham, 2019a. Leveraging change intents for characterizing and identifying large-review-effort changes. In: Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering. pp. 46–55.
Wang, Ying, Chen, Bihuan, Huang, Kaifeng, Shi, Bowen, Xu, Congying, Peng, Xin, Wu, Yijian, Liu, Yang, 2020. An empirical study of usages, updates and risks of third-party libraries in java projects. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 35–45.
Wang, Min, Lin, Zeqi, Zou, Yanzhen, Xie, Bing, 2019b. Cora: decomposing and describing tangled code changes for reviewer. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 1050–1061.
Wang, Tong, Wang, Dongdong, Zhou, Ying, Li, Bixin, 2019c. Software multiple-level change detection based on two-step mpat matching. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp. 4–14.
Wen, Zhihua, Tzerpos, Vassilios, 2004. An effectiveness measure for software clustering algorithms. In: Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004. IEEE, pp. 194–203.
Williams, Byron J., Carver, Jeffrey C., 2010. Characterizing software architecture changes: A systematic review. Inf. Softw. Technol. 31–51.
Williams, Byron J., Carver, Jeffrey C., 2014. Examination of the software architecture change characterization scheme using three empirical studies. Empir. Softw. Eng. 19 (3), 419–464.
Wimmer, Manuel, Moreno, Nathalie, Vallecillo, Antonio, 2012. Viewpoint co-evolution through coarse-grained changes and coupled transformations. In: International Conference on Modelling Techniques and Tools for Computer Performance Evaluation. Springer, pp. 336–352.
Wong, S., Cai, Y., Valetto, G., Simeonov, G., Sethi, K., 2009. Design rule hierarchies and parallelism in software development tasks. In: Proc. of Automated Software Engineering. p. 197.
Xing, Zhenchang, Stroulia, Eleni, 2007. Differencing logical UML models. Autom. Softw. Eng. 14 (2), 215–259.
Yan, Meng, Fu, Ying, Zhang, Xiaohong, Yang, Dan, Xu, Ling, Kymer, Jeffrey D., 2016. Automatically classifying software changes via discriminative topic model: Supporting multi-category and cross-project. J. Syst. Softw. 296–308.
Yu, Shanshan, Su, Jindian, Luo, Da, 2019. Improving bert-based text classification with auxiliary sentence and domain knowledge. IEEE Access 7, 176600–176612.
Zanjani, Motahareh Bahrami, Kagdi, Huzefa, Bird, Christian, 2016. Automatically recommending peer reviewers in modern code review. Trans. Softw. Eng. 530–543.
Zhang, Su, Zhang, Xinwen, Ou, Xinming, Chen, Liqun, Edwards, Nigel, Jin, Jing, 2015. Assessing attack surface with component-based package dependency. In: International Conference on Network and System Security. Springer, pp. 405–417.
Zwinkau, Andreas, 2019. Definitions of software architecture. beza1e1.tuxen.de/definitions_software_architecture.html.

Amit Kumar Mondal is a Ph.D. student in the Software Engineering Lab at the University of Saskatchewan. He completed his M.Sc. in Software Engineering from the Computer Science Department of the University of Saskatchewan, Canada by working under the supervision of Dr. Chanchal K. Roy and Dr. Kevin A. Schneider. He is also a faculty member of the Computer Science and Engineering Discipline, Khulna University, Bangladesh. He has been a reviewer of the IEEE Xplore journal. His primary research interests are Software Architecture, Software Maintenance and Evolution, Big Data Analytic and Automated Software Engineering.

Kevin A. Schneider is a Professor of Computer Science, Special Advisor ICT Research and Director of the Software Engineering Lab at the University of Saskatchewan. Dr. Schneider has previously been Department Head (Computer Science), Vice-Dean (Science) and Acting Chief Information Officer and Associate Vice-President Information and Communications Technology.

Before joining the University of Saskatchewan, Dr. Schneider was CEO and President of Legasys Corp., a software research and development company specializing in design recovery and automated software engineering. His research investigates models, notations and techniques that are designed to assist software project teams develop and evolve large, interactive and usable systems. He is particularly interested in approaches that encourage team creativity and collaboration.

Banani Roy is an Assistant Professor at the Department of Computer Science at the University of Saskatchewan. Dr. Roy closely works with the graduate, summer, and undergraduate students, and postdoctoral fellows in the Cloud-based Big Data Analytics for Crop Phenomics project (a.k.a P2IRC Project 3.1, a USask CFREF funded project) and Global Water Future Project (a.k.a. GWF project, second USask CFREF funded project). Led by Prof. Kevin Schneider, she was also able to secure a Compute Canada Resource Allocation Competition ($178,543.00) grant for the P2IRC project. She received her Ph.D. in Engineering Interactive Systems from the Queen's University in 2013 under the supervision of Prof. Nicholas Graham and Prof. Carl Gutwin, She worked as a faculty member at the Computer Science and Engineering Department at Khulna University of Engineering and Technology (KUET). She also received several awards including a Queen's Graduate Scholarship and the highly competitive Ontario Graduate Scholarship in Science and Technology (OGSST). Her research interests are Engineering Interactive Systems, Software Reengineering, Software Architecture, Big Data Analytics and Empirical Software Engineering.

Chanchal K. Roy is a Professor of Software Engineering/Computer Science at the University of Saskatchewan, Canada. While he has been working on a broad range of topics in Computer Science, his chief research interest is Software Engineering. In particular, he is interested in software maintenance and evolution, including clone detection and analysis, program analysis, reverse engineering, empirical software engineering and mining software repositories. He served or has been serving in the organizing and/or program committee of major software engineering conferences (e.g., ICSE, ICSME, SANER, ICPC, SCAM, CASCON, and IWSC). He has been a reviewer of major Computer Science journals including IEEE Transactions on Software Engineering, International Journal of Software Maintenance and Evolution, Science of Computer Programming, Journal of Information and Software Technology and so on. He received his Ph.D. at Queen's University, advised by James R. Cordy, in August 2009.