A Meta-Study of Software-Change Intentions

JACOB KRÜGER, Eindhoven University of Technology, Eindhoven, Netherlands
YI LI, Nanyang Technological University, Singapore, Singapore
KIRILL LOSSEV, University of Toronto, Toronto, Canada
CHENGUANG ZHU, The University of Texas at Austin, Austin, United States
MARSHA CHECHIK, University of Toronto, Toronto, Canada
THORSTEN BERGER, Ruhr-University Bochum, Bochum, Germany
JULIA RUBIN, The University of British Columbia, Vancouver, Canada

Every software system undergoes changes, for example, to add new features, fix bugs, or refactor code. The importance of understanding software changes has been widely recognized, resulting in various techniques and studies, for instance, on change-impact analysis or classifying developers' activities. Since changes are triggered by developers' intentions—something they plan or want to change in the system—many researchers have studied intentions behind changes. While there appears to be a consensus among software-engineering researchers and practitioners that knowing the intentions behind software changes is important, it is not clear how developers can actually benefit from this knowledge. In fact, there is no consolidated, recent overview of the state of the art on software-change intentions (SCIs) and their relevance for software engineering. We present a meta-study of 122 publications, which we used to derive a categorization of SCIs and to discuss motivations, evidence, and techniques relating to SCIs. Unfortunately, we found that individual pieces of research are often disconnected from each other, because a common understanding is missing. Similarly, some publications showcase the potential of knowing SCIs, but more substantial research to understand the practical benefits of knowing SCIs is needed. Our contributions can help researchers and practitioners improve their understanding of SCIs and how SCIs can aid software engineering tasks.

CCS Concepts: • **General and reference → Surveys and overviews**; • **Software and its engineering → Software evolution**; **Maintaining software;**

Additional Key Words and Phrases: Intentions, software evolution, change management, version control

Authors' Contact Information: Jacob Krüger, Eindhoven University of Technology, Eindhoven, Noord-Brabant, Netherlands; e-mail: j.kruger@tue.nl; Yi Li, Nanyang Technological University, Singapore, Singapore; e-mail: yi_li@ntu.edu.sg; Kirill Lossev, University of Toronto, Toronto, Ontario, Canada; e-mail: kirill.lossev@mail.utoronto.ca; Chenguang Zhu, The University of Texas at Austin, Austin, Texas, United States; e-mail: cgzhu@utexas.edu; Marsha Chechik, University of Toronto, Toronto, Ontario, Canada; e-mail: chechik@cs.toronto.edu; Thorsten Berger, Ruhr-University Bochum, Bochum, Nordrhein-Westfalen, Germany; e-mail: thorsten.berger@rub.de; Julia Rubin, The University of British Columbia, Vancouver, British Columbia, Canada; e-mail: mjulia@ece.ubc.ca.

ACM 0360-0300/2024/10-ART300
https://doi.org/10.1145/3661484

ACM Reference Format:
Jacob Krüger, Yi Li, Kirill Lossev, Chenguang Zhu, Marsha Chechik, Thorsten Berger, and Julia Rubin. 2024. A Meta-Study of Software-Change Intentions. *ACM Comput. Surv.* 56, 12, Article 300 (October 2024), 41 pages. https://doi.org/10.1145/3661484

1 Introduction

Software systems evolve rapidly, which is reflected in the many changes that developers apply to the codebase of their systems [40, 112, 119]. For instance, developers add new features, fix bugs, improve system performance, or optimize the structure of source code via refactorings. There seems to be consensus in software-engineering research that knowing such developer intentions behind a software change is beneficial, for instance, for managing software projects (e.g., for assigning resources to specific activities [102]), for creating training data (e.g., for automated program repair [99]), or for improving change histories (e.g., to transplant specific changes [100]). Still, developers' **software-change intentions (SCIs)**—specifying what they want(ed) to change in their system by modifying it—are rarely explicitly recorded. Consequently, many researchers rely on techniques to automatically recover SCIs, for instance, from commits [37, 41, 50, 53, 159].

Unfortunately, it is challenging to identify the different SCIs that may also be tangled within a single change and to untangle them [11, 20, 33, 45, 52, 152, 159, 163, 170]. This task is cumbersome and expensive, since the developers who implemented the changes typically use arbitrary natural-language descriptions to document changes (e.g., commit messages). Moreover, whether a description properly reflects on the change (e.g., a change described to fix a bug may also involve refactoring) and which descriptions refer to which SCIs (e.g., "optimization" versus "performance improvement") is often unclear. When we investigated the existing body of research, we noted that many publications referred to similar or even the same SCIs using different terms, operated on different levels of granularity, (re-)defined SCIs as they saw fit, and combined or intermixed orthogonal and overlapping categorizations of SCIs. This challenged our understanding of how different pieces of research were connected, what the actual benefits of using or understanding SCIs were, and how we could reuse or combine the existing research contributions.

So, despite extensive and very active research [5, 7, 37, 50, 80, 156], an important question remains: *What is a common ground for describing SCIs and what is the evidence that knowing SCIs is useful in practice?* Some researchers have classified different types and subsets of SCIs to varying degrees of abstraction (e.g., as maintenance activities or specific refactorings) [13, 43, 62, 64, 148, 166]. Unfortunately, the more systematic attempts (e.g., those based on literature surveys) of understanding SCIs are decades old, do not reflect on the benefits of knowing SCIs (e.g., they only derive or define a taxonomy for the purpose of having one), do not discuss the existing empirical evidence, and do not consider the problem of intermixed categories. Many publications do not even mention the term "intention," even though they are concerned with SCIs. For instance, a typical scenario is researchers being concerned with identifying bug-fixing (i.e., corrective [148]) software changes from commits, which can then be used for designing program-repair or fault-prediction techniques [71, 99, 140, 154]. The researchers typically refer to "bug fix" or "repair" changes, but essentially identify software changes with a corrective SCI. Since none of the previously proposed taxonomies has established itself as a common ground for describing SCIs, researchers and practitioners are building on whatever definitions of SCIs are most feasible for them. This, in turn, complicates building a common knowledge base, comparing research, and collecting reliable evidence on the usefulness of SCIs.

In this article, we present a meta-study in which we identified and analyzed a large body of research on SCIs to provide a detailed understanding of how knowledge on SCIs is used in research and what the actual evidence for benefiting from this knowledge is. To tackle this gap, we defined four research objectives (ROs) for our meta-study as follows:

RO$_1$ capturing the research on documenting, analyzing, and using SCIs;
RO$_2$ deriving a systematic categorization for describing SCIs;
RO$_3$ collecting empirical evidence on the usefulness of knowing SCIs; and
RO$_4$ comparing techniques that use and recover SCIs.

To identify relevant publications, we conducted a systematic literature review [73]. Our analysis of the resulting 122 publications is *qualitative*, investigating their actual content and contributions rather than providing publication statistics only. We provide in-depth insights into the usefulness of SCIs as a concept based on this extensive dataset of publications, which is available in a persistent open-access repository.[1]

Most of the publications we identified focus on techniques (50) or empirical studies (48) and use SCIs in a wide range of contexts (e.g., predicting maintenance activities, refactoring version histories). Building on a sample of the publications, we derived a systematic categorization to provide a structure for organizing SCIs. Unfortunately, we identified little evidence on the practical benefits of knowing SCIs, even though they are used in various techniques. During our meta-study, we experienced that understanding and structuring the existing research was challenging, a problem our categorization helps to tackle.

Our contributions in this article can guide researchers and practitioners in advancing techniques and studies on software evolution by providing a common ground on SCIs. More precisely, by capturing the state of the art (**RO$_1$**), we contribute a concise body of knowledge of the area that serves as a reference for others. Building on this body of knowledge, we derive a categorization for describing the notions of SCIs used in the literature (**RO$_2$**). This provides a single comprehensive overview of the SCIs used, exemplifies these SCIs, and eases communication as well as knowledge sharing. The empirical evidence (**RO$_3$**) indicates the potential impact knowing SCIs can have—but it also highlights the need for more in-depth studies. By comparing existing techniques (**RO$_4$**), we contribute an overview of how SCIs are used and what researchers or practitioners can build upon to advance software engineering in the future.

The remainder of this article is organized as follows: We describe the methodology of our meta-study in Section 2. In Section 3, we present the results of our literature search to capture the state of the art on SCIs (**RO$_1$**). Then, we address our three remaining research objectives in Section 4 (**RO$_2$**), Section 5 (**RO$_3$**), and Section 6 (**RO$_4$**), respectively. We discuss potential threats to the validity of our meta-study in Section 7 and its implications in Section 8. In Section 9, we summarize the related work before concluding this article in Section 10.

2 Methodology

In this section, we describe our methodology illustrated in Figure 1.

2.1 Initial Screening

Initially, we aimed to survey the literature to identify the current state of the art on intentions in software engineering in general. For this purpose, we performed an unstructured screening building on our prior knowledge of the area as well as automated searches using relevant keywords (e.g., "intent") in different search engines (e.g., DBLP and Google Scholar). During this screening, we organized the found publications into the three following categories:

(1) those that describe intention-related concepts in the broader context of software engineering (e.g., developers' knowledge of stakeholder intentions) [15, 60, 75, 85, 126, 127, 141, 143, 159];

[1] https://doi.org/10.5281/zenodo.10977570

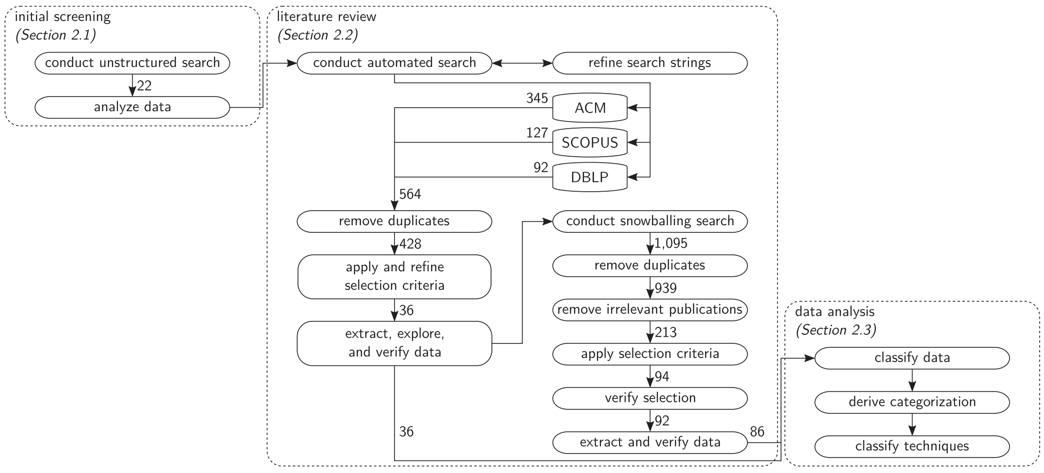


Fig. 1. Overview of our methodology for identifying and analyzing relevant publications. The numbers indicate the numbers of publications stemming from the previous step.

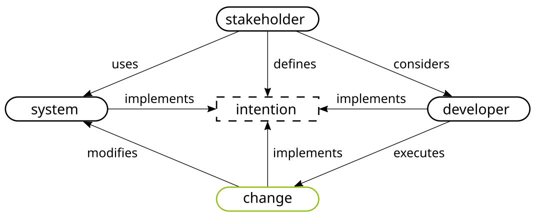


Fig. 2. High-level overview of the role of intentions in software engineering. We are primarily concerned with SCIs, which we highlight in green.

(2) those that use intentions as a concept to support established use cases (e.g., generating commit messages to specify SCIs) [9, 58, 59, 94, 97, 138, 139]; and
(3) those that report on or use classifications of intentions in software engineering (e.g., for labeling commits) [21, 25, 47, 91, 101, 148].[2]

This initial screening allowed us to achieve a high-level understanding of different uses of intentions in software-engineering publications.

Results. From our initial screening, we considered 22 publications and one website (referenced for each category defined above) as relevant and investigated them in detail. We found that these sources focus on one or more of four origins of intentions: those expressed by stakeholders of a system, those implemented by the developers, those behind changes (i.e., SCIs), and those implemented in the system. In Figure 2, we display these origins and their relations as a high-level overview.

Discussion. Intentions are a key notion in software engineering. Usually, intentions are defined by a stakeholder of a system who uses that system for a specific purpose, with those intentions typically being structured around requirements or features. A developer is then responsible for implementing such intentions by considering the stakeholder's descriptions and executing a corresponding change (e.g., modifying an asset). Consequently, a software change represents the actual

[2]https://www.parkersoftware.com/blog/the-4-software-maintenance-categories-and-what-they-mean-for-your-users/

implementation of the intention (correct or incorrect) in terms of modifications to the system. We can see that the notion of intentions relates to each step relevant for evolving a software system, but intentions are often expressed via different notations (e.g., requirements, features, assets, commits, revisions). Unfortunately, while this high-level understanding of intentions describes software evolution and provides a good intuition of the connections between the involved notations, it is far too coarse-grained to actually support researchers or practitioners.

Based on this insight, we decided to refine our analysis through an extensive meta-study to provide a foundation for our research vision [80]. To limit the scope of this meta-study and obtain a more concise overview, we chose to focus on SCIs. We decided to focus on SCIs, because a software change represents the actual implementation of an intention in a system—regardless of whether there have been miscommunication, mismatches, or errors in specifying this intention at any point. For instance, a software change that is intended to fix a bug represents the actual implementation of the developer's solution in terms of modifications to the system, and the intention typically originates from some stakeholder request (e.g., a bug report). However, the intended change may not align with the actual change; for instance, the change may involve other intentions of the developer (e.g., tangling refactoring) or may not fulfill its purpose (e.g., not fixing the bug). For software engineering, it is arguably important to understand SCIs as well as potential mismatches to their implementation, considering that they should represent stakeholder intentions. Consequently, we argue that the actual software changes provide the best and most relevant understanding for researchers and practitioners of the way (i.e., actual modifications to the source code) in which intentions are implemented in a software system.

2.2 Literature Review

To systematically elicit relevant publications for our meta-study, we followed the methodology of systematic literature reviews [73]. In the following, we report the individual steps of our methodology. We do not focus on providing statistics on the publications identified but instead on an in-depth meta-study of their actual content.

Selection Criteria. We defined selection criteria based on our insights from the initial screening. While testing different search strings and exploring the results of our automated search (explained shortly), we identified that these criteria were too broad, leading to the inclusion of many publications that were not relevant. We iteratively refined our selection criteria during discussions among all authors until we finally defined three **inclusion (IC)** and five **exclusion (EC)** criteria. Specifically, we included a publication if it satisfies IC_1, IC_2, and IC_3 as follows:

- IC_1 The publication defines types of changes/intentions from the perspective of developers (e.g., a taxonomy of intentions or maintenance activities, empirical studies classifying commits).
- IC_2 The publication uses types of changes/intentions from the perspective of developers in its actual contributions (e.g., not only to motivate a piece of work).
- IC_3 The publication is related to changes/intentions on source code; including other artifacts only if they are directly mapped to source code.

Please note that we did not assess how "fundamental" SCIs are to the research reported within a publication (e.g., identifying SCIs versus knowing a SCI to improve a technique). Consequently, we cover publications that are concerned with SCIs and also those that weakly relate to them. This is on purpose to provide a more complete overview of how different areas in software engineering are related to SCIs—and thereby to each other.

We excluded publications that fulfilled any of the following criteria:

- EC_1 Not written in English;
- EC_2 Not peer reviewed, such as dissertations, technical reports, or books;

EC$_3$ Fewer than three pages (e.g., keynotes, posters);
EC$_4$ Out of scope despite referring directly to intentions and software, for instance, the intention to use software tools, intent automata, or intentions behind software piracy; or
EC$_5$ Uses intention as a name for a concept unrelated to software changes (e.g., Android uses the name "intent" for an inter-component messaging system[3]).

Note that we did not employ a quality assessment of the publications we selected, since our research goal was broader than a comparison of empirical studies (the main purpose of a quality assessment) and, therefore, involved completely different types of research (e.g., techniques, empirical studies, methodologies) that can hardly be directly compared in terms of quality.

Automated Search. First, we performed an automated search. For this purpose, we built on the keywords we used during our initial screening to derive a number of search strings. We experimented with those search strings and explored the publications they returned (e.g., number, relevance). Based on these trial runs, we added synonyms and tested different combinations of operators (e.g., AND, OR). In the end, we decided to use two different search strings, which we executed on the ACM Digital Library Guide to Computing Literature, SCOPUS, and DBLP. We used these search engines, because they cover a variety of publishers, are (or can be) limited to the field of Computer Science, and allow bulk downloading of all returned publications. Specifically, our search strings were as follows:

(1) *intent* AND software* for ACM and SCOPUS
intent + software for DBLP
(2) *(change* OR commit*) AND (intent* OR maint* OR evol*) AND (classif* OR taxono* OR categor* OR recov* OR extract* OR generat* OR activit* OR label* OR reason* OR trace* OR detect* OR analy*)* for ACM and SCOPUS

DBLP does not allow for complex queries, which is why we did not employ the second search string on this engine. Furthermore, we limited SCOPUS to Computer Science, aiming to exclude unrelated research areas. With the first search string, we aimed to capture publications that directly refer to intentions in the context of software engineering. With the second search string, we built on our insights from our initial screening and trial runs to derive terms that researchers used in the context of SCIs. We employed the strings on titles, abstracts, and keywords for ACM and SCOPUS, whereas DBLP allows searching only over standard bibliographic data (e.g., titles).

Results of the Automated Search. The first author executed the automated searches, which returned 92 (DBLP), 153 (ACM), and 45 (SCOPUS) publications for the first search string as well as 192 (ACM) and 82 (SCOPUS) for the second one. Next, we consolidated the 564 results into a single BIB file. Using this file, we removed duplicates with JabRef's feature for importing new entries (460 results) and the feature for checking for duplicates of JabRef (459) and KBibTex (428). We put the remaining 428 publications into a spreadsheet that was available to all authors.

For these 428 publications, we employed our initial set of selection criteria on titles, abstracts, and then full texts if needed. Specifically, the first author checked every publication, while each other author checked a subset (each publication was reviewed by two authors). During this step, we determined that we could exclude a number of publications easily, but also had significant disagreements about some of the remaining ones. We found that this was mainly caused by different interpretations that we had about what exhibits a developer's intention at the source-code level. Some of the points of disagreement were "Are transformations code? What about grammars or models? Is implementing user intentions in the code in the scope of our study? What about architectural changes, commit messages, and comments?" With these experiences, we refined our selection criteria to derive the ones described above. Using these criteria, we re-iterated through

[3]https://developer.android.com/reference/android/content/Intent

all publications we did not yet exclude (i.e., that we included or for which we disagreed) and re-evaluated their scores. To resolve the remaining disagreements, one author who did not review the corresponding publication before checked it and provided a reasoning for the final decision. We repeatedly re-checked all publications during the snowballing and our data analysis (as explained shortly, we removed five more publications) to ensure that they were actually in the scope of our meta-study. In the end, we included 36 publications as relevant.

Snowballing. When investigating the 428 publications returned by the automated search, we found that some publications we knew to be relevant from our initial screening were missing—even if they matched our search strings. A prominent example for this case was the taxonomy of Swanson [148] that was not part of the 428 publications. This has likely been caused by the strong focus of our search strings and technical problems of search engines in the area of computer science [8, 79, 135]. We decided to perform backwards snowballing [167] on the 36 included publications to complement our automated search and mitigate such problems. Namely, we iterated through all 36 publications and extracted their references, resulting in 1,095 additional publications in a second spreadsheet.

Results of the Snowballing. The first author began with reducing the number of publications by removing duplicates (939) as well as those clearly not relevant (e.g., from the biology domain, theses, fewer than three pages) based on the author's expertise after analyzing the previous sample of publications (213). Afterwards, the first author applied the selection criteria on each remaining publication, indicating that 94 publications should be included. To verify that the exclusions and inclusions were reasonable, three other authors independently cross-checked random samples of 20 different publications each (i.e., we cross-checked 60 of 213 decisions). We identified disagreements in six cases, which we discussed among all authors. In the end, we found that only two of the included publications should instead be excluded, resulting in 92 publications.

The discussion revealed that the first author was too inclusive in his selection of publications during the snowballing. This insight enhances our confidence that the sample from the snowballing does not miss important publications. We removed six more publications during our data extraction and detailed analysis, resulting in 86 included publications.

During our analysis, we found that the additional publications confirmed our previous insights (e.g., we could match them to our categorization of SCIs and characterization of techniques), but did not reveal new major insights. We considered this as saturation and stopped after the first iteration of snowballing, ending up with a total of 122 publications as the basis for our meta-study.

Data Extraction. The first author of this article extracted the data for each publication. Then, the third author checked the extracted data to ensure their correctness. Both authors read through each publication and extracted data relevant for the respective type of research. Moreover, the second and fourth authors extracted additional data on the techniques we identified (**RO$_4$**). All authors met regularly to discuss and refine the extracted data (e.g., checking whether they were comprehensible and adding or removing relevant data entries). For each publication, we elicited the following data:

— *Bibliographic Data:* A collection of standard data about a publication, namely author names, title, publication year, and a link to the published version (typically based on a DOI).
— *Goal:* A one-sentence summary of the main research goal of the publication.
— *Type of Contribution:* The main contribution of the publication, representing one of the following categories (note that we assigned the dominant type of contribution, e.g., a publication proposing and empirically evaluating a technique is classified as technique):
  - Dataset—the publication presents a dataset of changes, together with some type of SCIs.
  - Empirical Study—the publication presents a study related to identifying SCIs or some software-change phenomena where SCIs may be useful to know.

  - Literature Review—the publication presents a literature review of software changes, including some notion of SCIs (e.g., to classify research or derive a taxonomy).
  - Methodology—the publication presents a methodology that incorporates SCIs.
  - Taxonomy Proposal—the publication presents a taxonomy (not derived from the literature) of changes or SCIs, which can refer to the problem space (e.g., perform maintenance) or to the solution space (e.g., remove a class), which we also recorded.
  - Technique—the publication presents a technique for identifying SCIs, or a technique that relies on some notion of SCIs (e.g., for verifying program behavior).
- *Benefits:* A brief description of the benefits of knowing SCIs as motivated by the authors. We were specifically interested in documented use cases for practitioners that go beyond just understanding the nature of changes, which is only useful for researchers (e.g., for classifying research that is related to the different types of changes).
- *Evidence:* A short note whether and what kind of evidence a publication provides on the usefulness of knowing SCIs.
- *Technique:* Provides further information about the intention-related technique if one is presented in the publication. Specifically, we recorded the aim of the technique, its input, output, the underlying technologies (e.g., static analysis, machine learning), and how SCIs are involved (e.g., to classify commits, as an intermediate entity).
- *Taxonomy:* A list of different SCIs or types of changes defined or used in the publication (e.g., corrective, bug fix, addition).

We documented these data in a shared spreadsheet to which all authors had access.

Reliability of the Results. We continuously discussed our progress and results in weekly meetings among all authors. Moreover, we employed multiple rounds of verification to ensure that the elicited publications and data were reliable. First, during the automated search, each publication was classified by two different authors, with the first author classifying all publications to obtain an overview understanding. We discussed all discrepancies in the decisions among at least five authors. Second, recall that we employed the snowballing, because we found that several publications we identified during the initial screening were missing in our dataset. During the snowballing, we relied on the first author to classify the publications, but performed cross-checks on 60 randomly selected publications.. We found that the first author was too inclusive for some publications, which resulted in further exclusions during our analysis. In summary, we conducted an extensive literature search with multiple rounds of verification, yielding 122 publications on various topics—which we argue is a reliable dataset for our meta-study.

2.3 Data Analysis

Next, we report the process of analyzing our dataset of publications.

Classifying the State of the Art (RO_1) and Empirical Evidence (RO_3). We provide an overview of how SCIs have been used in the 122 publications (Section 3), and of the empirical evidence knowing SCIs can have in practice (Section 5). To elicit such qualitative data, the first author iterated through each publication and extracted relevant statements based on an open-coding-like process. More specifically, the first author read through each publication to identify statements that relate to the relevant data fields. For example, the usage of SCIs within a publication was typically described in the introduction or under methodology. The benefits and empirical evidence were typically part of results or discussion sections, with summarizing statements reported in the abstract, introduction, or conclusion. Last, taxonomies used within the publications were often described within tables and methodologies, usually with an explanation of how they were created. For instance, a publication may reuse an existing taxonomy (e.g., citing the one by Swanson [148]), elicit it from a dataset, or define it ad hoc—which we assumed to be the case if no other explanation has been provided. By iterating through each publication and focusing on the sections most related to each data entry, the first author carefully extracted relevant statements (e.g., the taxonomy from a table). Afterwards, the third author cross-checked the data. Since this process yielded too much detailed data, we performed a data synthesis following an open-card-sorting-like method [175]. In this step, the first author identified common themes and topics in the data, for instance, that several publications motivated their work with predicting the efforts related to software changes. Then, the sixth author iterated through the data to see whether they were understandable, and cross-checked for individual publications whether the data were correct. Following this step, all authors agreed to the final structure and level of detail of our dataset. The first author re-iterated through each publication to update the dataset accordingly. During this step, he refined statements to be more descriptive, updated the contributions (for which we decided to adapt the classification above), and re-checked the remaining data. This also led to the inclusion of one more publication to the dataset, which we accidentally missed during the snowballing (it may have been removed by the automatic duplicate detection).

Deriving a Categorization of SCIs (RO$_2$). Inspired by the qualitative data-analysis techniques from grounded theory, namely open and axial coding [145], the fifth and seventh authors independently read 20 of the identified publications each (randomly chosen, distinct sets), looking for SCIs (e.g., a category of activities), their definitions (i.e., of the category), and concrete examples for these (i.e., individual activities). Then, the two authors met multiple times to discuss and unify the SCIs, grouping them into five main categories, roughly corresponding to *why* and *when* a certain *action* is performed; *what* objects it manipulates; and *who* will benefit from the action. As the final outcome, the two authors defined five orthogonal categories: *goals*, *actions*, *object*, *customer*, and *lifecycle phase*. To validate that this categorization would avoid the issues of previous taxonomies (e.g., overlapping categories, inconsistent terms), we discussed it among all authors and mapped other SCI-related terms we extracted before to the categories (cf. Section 4). During this process, we found that we could reassign most terms used in existing publications into one of our distinct categories without overlaps. Note that we did not aim to provide a unified categorization (i.e., a full-fledged taxonomy) that involves all terms from every publication, since this would involve many categories not related to SCIs and result in various levels of granularity. For instance, we did not map or integrate all sub-types of bugs (subsumed under corrective) or refactorings (subsumed under preventive: improving maintainability) mentioned within the publications into our categorization (cf. Figure 4) to avoid too many fine-grained types. Instead, we aimed to derive a categorization that can be used to properly describe SCIs at one level of detail, separating common orthogonal categories, allowing for extensions or refinements, and resulting in a comprehensive description of SCIs. Via these means, we aimed to ensure that our categories are truly orthogonal and avoid the typical problems we experienced and expressed the most relevant aspects of an SCI. Last, we noticed that our categories also cover the high-level intentions we identified from our initial screening of the literature (cf. Figure 2). Specifically, developers executing an SCI are considered, together with other stakeholders, as *customers*; the change itself is an *action*, the system is the *object*, and the underlying intention is the *goal*. That our categories cover all of these high-level concepts related to intentions improves our confidence that these are feasible for describing SCIs. We present our resulting categorization with discussions and examples for each SCI in Section 4.

Classifying SCI-related Techniques (RO$_4$). The second and fourth authors reviewed all 50 publications noted as proposing a technique related to SCIs to identify each technique's specific goal, inputs, outputs, and underlying approaches and how SCIs are used in it. Both authors discussed and compared their review results, until reaching consensus on how to describe and classify the techniques based on these data. Most importantly, they found that the techniques identify SCIs, use them to improve an existing technique, or involve them as intermediate results. Also, the authors could distinguish three primary underlying approaches, namely static-analysis, dynamic-analysis, and statistical (including classification and machine learning) approaches. To extract the data and classify the techniques into consistent categories, the authors relied on their expertise and the descriptions of SCIs within the techniques, provided by the publications. Last, the categorizations were reviewed by all of the authors. By comparing the insights and experiences of all analyses, we aimed to avoid redundancies, obtain a concise overview about the publications, and validate all our findings on SCIs. We present our discussion of techniques that involve SCIs in Section 6.

3 Understanding the Identified Publications (RO_1)

We now briefly summarize the results of our literature search to discuss the state of the art on SCIs—which we detail in the next sections. To this end, we display overviews of all 122 included publications in Table 1 (empirical studies), Table 2 (datasets, methodologies, literature reviews, taxonomy proposals), and Table 3 (techniques). For each publication, we summarize its core properties. These properties involve whether we identified the publication through the automated search or snowballing (source), its reference (two references imply journal extensions), and the type of contribution. Then, we display short descriptions of the goals, benefits, and evidence reported in a publication, which we discuss in Section 5. In the column taxonomy (discussed in Section 4), we specify whether a publication directly reuses, extends (e.g., with orthogonal categorizations or by integrating additional terms), synthesizes (from the literature), or proposes (i.e., not explicitly building on previous ones) a taxonomy. We considered a taxonomy within a publication to be proposed by the authors themselves if they did not explicitly cite a publication from which it was reused or derived. While we noticed that some publications used taxonomies similar or identical to existing ones, particularly the one by Swanson [148], the missing citations made it impossible to understand whether a taxonomy was reused from a specific publication or was an ad hoc proposal based on knowledge obtained from somewhere else (e.g., a course or website). For techniques (cf. Table 3), we also provide an indication on how SCIs and what underlying approaches are used within each technique, which we discuss in more detail in Section 6.

In Figure 3, we provide a yearly overview of when the publications have been published, at what venues (abbreviated labels), and what they contribute (colors and/or borders). We can see that we included several rather old papers, such as the taxonomy proposed by Swanson in 1976. However, most publications in our dataset have been published between 2003 and 2017, and we also identified more recent publications. Considering our search strategy (i.e., automated search in 2021, backwards snowballing), the distribution of publications is reasonable and covers a long period of software-engineering research that relates to SCIs. Similarly, we can see that the publications appeared at a range of venues similar to the other literature reviews on software changes that we identified (cf. Section 9); with well-established software engineering and evolution venues occurring more frequently, for instance, the International Conference on Software Maintenance and Evolution (20, 1988–2016), **International Conference on Mining Software Repositories (MSR)** (9, 2005–2020), the International Conference on Software Engineering (8, 1976–2019), the Journal of Systems and Software: Evolution and Process (8, 1990-2013), or the IEEE Transactions on Software Engineering (6, 1995–2014). While the distribution over time indicates the continuous interest in SCIs, the venues clearly highlight that SCIs are relevant to a broad range of research topics. Last, we can see that publications aiming to synthesize or specify certain types of SCIs (literature reviews, taxonomies) occur frequently throughout the years—but recent reviews are missing. This highlights the continuous interest in research for constructing a common foundation for describing software changes and SCIs to build upon, which has not yet been achieved (cf. Section 9). Overall, most publications either contribute a technique (50) or an empirical study (48) related

Table 1. Overview of the 48 Publications Classified as Empirical Study (ES) in Our Meta-study

source	reference	contribution	goal	benefits	evidence	taxonomy
S, S	[1, 2]	ES	comprehension	project monitoring	practice	[148]
S	[10]	ES	predicting	effort estimation	–	❍
S	[11]	ES	predicting	bug prediction	–	❍
S	[12]	ES	predicting	bug prediction	–	❍
S	[14]	ES	comprehension	project monitoring	–	[148]
S	[22]	ES	predicting	measuring maintainability	–	[148]+
S	[23]	ES	comprehension	comprehend software evolution	–	❍
S	[35]	ES	comprehension	comprehend software evolution	–	[148]+
S	[36]	ES	labeling	project monitoring	–	❍
S	[38]	ES	predicting	bug prediction	–	❍
S	[42]	ES	comprehension	comprehend software evolution	–	[148]+
S	[46]	ES	predicting	bug prediction	–	❍
S	[47]	ES	comprehension	comprehend software evolution	–	[148]+
S	[49]	ES	comprehension	comprehend software evolution	–	❍
S	[51]	ES	labeling	identifying misclassification	–	[148]+
S	[52]	ES	untangling	improve version history	–	❍
S	[63]	ES	predicting	effort estimation	–	[148]+
S	[67]	ES	predicting	change prediction	–	❍
S	[69]	ES	comprehension	comprehend software evolution	–	❍
S	[70]	ES	comprehension	comprehend software evolution	survey	❍
S	[87]	ES	predicting	effort estimation	–	[148]+
S	[93]	ES	comprehension	comprehend software evolution	survey	[148]
S	[106]	ES	comprehension	comprehend software evolution	–	❍
A	[104]	ES	visualization	ensuring intention fulfillment	–	–
A	[107]	ES	comprehension	project monitoring	–	[148]+
S	[108]	ES	predicting	bug prediction	practice	❍
S	[109]	ES	comprehension	comprehend software evolution	–	[148]+
S	[110, 111]	ES	specification	safe evolution templates	–	❍
S	[113]	ES	comprehension	comprehend software evolution	survey	❍
S	[117]	ES	specification	compare evolution patterns	–	❍
S	[114]	ES	comprehension	comprehend software evolution	–	❍
A	[115]	ES	comprehension	comprehend software evolution	–	❍
S	[116]	ES	comprehension	comprehend software evolution	–	❍
S	[121]	ES	predicting	maintenance activities	–	[148]+
S	[122]	ES	predicting	bug prediction	–	❍
S	[123]	ES	comprehension	comprehend software evolution	–	[148]+
S	[129]	ES	comprehension	ensuring consistency between artifacts	payoff	❍
A	[132]	ES	comprehension	comprehend software evolution	–	[148]+
S	[134]	ES	comprehension	compare research	–	[148]
S	[140]	ES	comprehension	bug prediction	–	❍
S	[142]	ES	comprehension	comprehend software evolution	survey	[148]+
S	[151]	ES	comprehension	comprehend software evolution	survey	❍
S	[155]	ES	comprehension	comprehend software evolution	–	●
A	[157]	ES	comprehension	comprehend software evolution	–	❍
S	[158]	ES	comprehension	comprehend software evolution	–	❍
S	[173]	ES	comprehension	comprehend software evolution	survey	❍

A: automated search, S: snowballing
[xx]+: extends reference, ❍: proposes own taxonomy, ●: synthesizes from multiple publications

to SCIs. Furthermore, 12 publications propose a novel taxonomy of SCIs, 6 contribute a literature review, 5 provide a methodology for analyzing software changes, and 1 presents a dataset.

Types of Contributions. We can see that researchers have been concerned with, or used, SCIs to achieve various research contributions. Now, we exemplify some of these contributions. Herzig and Zeller [52] investigated five Java projects empirically to understand to what extent bug-fixing commits involve tangled changes and propose a predictor to help untangle them (cf. Table 1). We classified their contribution to be primarily empirical and to have the motivated goal of untangling SCIs, which can help improve version histories. Unfortunately, the publication does not provide

Table 2. Overview of the One Dataset (DS), Five Methodology (ME), Six Literature Review (LR), and 12 Taxonomy Proposal (TP) Publications in Our Meta-study

source	reference	contribution	goal	benefits	evidence	taxonomy
S	[118]	DS	comprehension	comprehend software evolution	–	❍
S	[19]	ME	comprehension	comprehend software evolution	practice	[148]+
A	[18]	ME	comprehension	problem identification	–	[148]
S	[68]	ME	comprehension	comprehend software evolution	–	❍
S	[128]	ME	comprehension	project monitoring	practice	[148]
A	[136]	ME	comprehension	problem identification	–	❍
A	[13]	LR	taxonomy	defining change properties	–	●
S	[62]	LR	taxonomy	comparing evolution research	–	●
S	[64]	LR	taxonomy	comparing MSR research	–	●
A	[88]	LR	taxonomy	comparing CIA research	–	●
S	[130]	LR	taxonomy	formalizing maintenance	–	●
S	[166]	LR	taxonomy	comparing research	–	●
S	[6]	TP	taxonomy	problem identification	–	❍
S	[24]	TP	taxonomy	comprehend software evolution	–	[148]+
S	[29]	TP	taxonomy	comprehend software evolution	–	[148]
A	[43]	TP	taxonomy	comprehend software evolution	–	❍
S	[54]	TP	taxonomy	comprehend software evolution	–	[148]+
S	[74]	TP	taxonomy	comparing research	–	❍
A	[89]	TP	taxonomy	improving CIA research	–	❍
S	[95]	TP	taxonomy	refine taxonomy	–	[148]+
S	[105]	TP	taxonomy	comparing research	–	❍
S	[148]	TP	taxonomy	comprehend software evolution	–	❍
A	[161]	TP	taxonomy	effort estimation	–	❍
S	[162]	TP	taxonomy	comparing research	–	[148]+

A: automated search, S: snowballing
[xx]+: extends reference, ❍: proposes own taxonomy, ●: synthesizes from multiple publications

evidence of how useful it is to know the tangled SCIs or to untangle them (this was out of scope for that publication). In terms of a taxonomy, Herzig and Zeller rely on the notion of bug-fixing SCIs (corrective according to Swanson and our categorization). Going into another direction, Benestad et al. [13] conducted a literature review on research that is concerned with understanding software evolution by analyzing individual changes (cf. Table 2). The authors have the goal of understanding and essentially providing a taxonomy of change attributes (which partially include SCIs) that have been used in research; and this work is highly similar to our own research (we compare both publications in more detail in Section 9). Since Benestad et al. contribute a literature review, they do not provide own empirical evidence on the benefits of knowing SCIs but synthesize their taxonomy from the literature. Sun et al. [146] propose a technique for improving change-impact analysis using a taxonomy of change types (cf. Table 3). While they evaluate their technique and show its potential benefits, the evidence that the technique works is purely research driven—in contrast to the practical evidence regarding the benefits of knowing SCIs we are interested in. Sun et al. define their own taxonomy of SCIs that is structured around adding, removing, or modifying different entities in the source code. We argue that all of such publications build on the connecting concept of SCIs. Not surprisingly for software engineering, SCIs are mostly considered when developing or improving a technique and when conducting empirical studies.

Table 3. Overview of the 50 Publications Classified as Technique (TE) in Our Meta-study

source	reference	contribution	goal	benefits	evidence	SCI usage	approaches	taxonomy
S	[17]	TE	predicting	effort estimation	correlations	▼	classification	–
S	[20]	TE	labeling	comprehend software evolution	–	◆	static	❍
A	[26]	TE	predicting	ensuring consistency between artifacts	–	▲	static	–
A	[28]	TE	transplantation	avoiding errors	correctness	◆	static	–
S, A	[30, 31]	TE	labeling	comprehend software evolution	–	▲	static	❍
S	[32]	TE	labeling	comprehend software evolution	–	◆	static + classification	❍
S	[33]	TE	labeling	project monitoring	–	▲	classification	[148]
A	[34]	TE	labeling	comprehend software evolution	–	▲	static	[32]
S	[39]	TE	labeling	comprehend software evolution	–	◆	static	❍
A	[41]	TE	comprehension	comprehend software evolution	–	▲	static + dynamic	❍
A	[44]	TE	mining	refine concern mining	correctness	▲	static	❍
S	[45]	TE	labeling	project monitoring	–	▲	classification	❍
S	[48]	TE	untangling	improve version histories	–	◆	static	–
A	[53]	TE	labeling	filter changes	–	▲	classification	[148]+
A	[55]	TE	labeling	project monitoring	–	▲	classification	[148]
A	[56]	TE	verification	ensuring intention fulfillment	feedback	▼	static	–
A	[59]	TE	verification	ensuring intention fulfillment	–	▼	static	–
S	[61]	TE	visualization	ensuring intention fulfillment	–	◆	static	–
S	[66]	TE	labeling	comprehend software evolution	–	▲	static	❍
S	[71]	TE	predicting	bug prediction	–	◆	static	❍
S	[72]	TE	untangling	improve version histories	–	◆	static	❍
S	[84]	TE	predicting	improving CIA	–	◆	static	❍
S	[90]	TE	predicting	maintenance activities	–	▲	ML/statistics	[148]
A	[91]	TE	labeling	project monitoring	–	▲	classification	[148]
A	[94]	TE	transplantation	operational intentions	correctness	▼	static	❍
A	[98]	TE	verification	ensuring intention fulfillment	–	▼	dynamic	–
A	[99]	TE	specification	compare evolution patterns	–	▼	static	[117]
A	[100]	TE	untangling	improve version histories	–	▼	static	❍
A	[102]	TE	labeling	project monitoring	–	▲	classification	[148]
A	[103]	TE	visualization	ensuring intention fulfillment	–	▲	static	–
S	[124]	TE	verification	ensuring intention fulfillment	usability	▼	static	–
A	[125]	TE	labeling	problem identification	–	▲	static	❍
S	[133]	TE	specification	safe evolution templates	–	◆	static	❍
A	[137]	TE	predicting	reducing effort	correctness	▼	dynamic	[148]
A	[139]	TE	programming	enabling domain experts	–	▼	code generation	–
S	[144]	TE	predicting	bug prediction	–	◆	static + classification	❍
S	[146]	TE	predicting	improving CIA	–	◆	static	❍
S	[147]	TE	predicting	improving CIA	–	◆	static	❍
S	[149]	TE	predicting	breaking changes	–	◆	static + classification	❍
S	[152]	TE	untangling	comprehend software evolution	–	◆	static	❍
S	[153]	TE	untangling	comprehend software evolution	–	▼	static	❍
S	[154]	TE	labeling	bug fix transplantation	–	◆	ML/statistics	❍
S	[159]	TE	labeling	comprehend software evolution	–	◆	static	❍
A, A	[163, 164]	TE	labeling	effort estimation	correctness	◆	classification	❍
A	[165]	TE	labeling	comprehend software evolution	–	◆	static	❍
S	[168]	TE	labeling	bug prediction	–	◆	static	❍
S	[170]	TE	labeling	project monitoring	–	▲	classification	[148]
A	[172]	TE	verification	ensuring intention fulfillment	understandability	▼	dynamic	–

A: automated search, S: snowballing
[xx]+: extends reference, ❍: proposes own taxonomy, ●: synthesizes from multiple publications
▲: identify SCIs, ▼: use SCIs to improve other techniques, ◆: SCIs as an intermediate result

Goals and Benefits. Next, we summarize the goals and benefits described in each publication. We can see in the tables that these vary widely. For instance, some publications motivate the ability to predict or label SCIs as a primary goal, which may help, for example, estimate maintenance efforts [10, 17], predict bugs [11, 12], or improve change-impact analysis [146, 147]. The broad range of contributions results in a variety of different goals and presumed benefits. As a consequence, it can be challenging to identify and synthesize commonalities between the publications, particularly since they also build on different taxonomies. In Section 5, we give a more detailed analysis of the publications' goals, benefits, and empirical evidence—aiming to provide an overview that helps clarify how SCIs have been used and how the different pieces of research are connected.

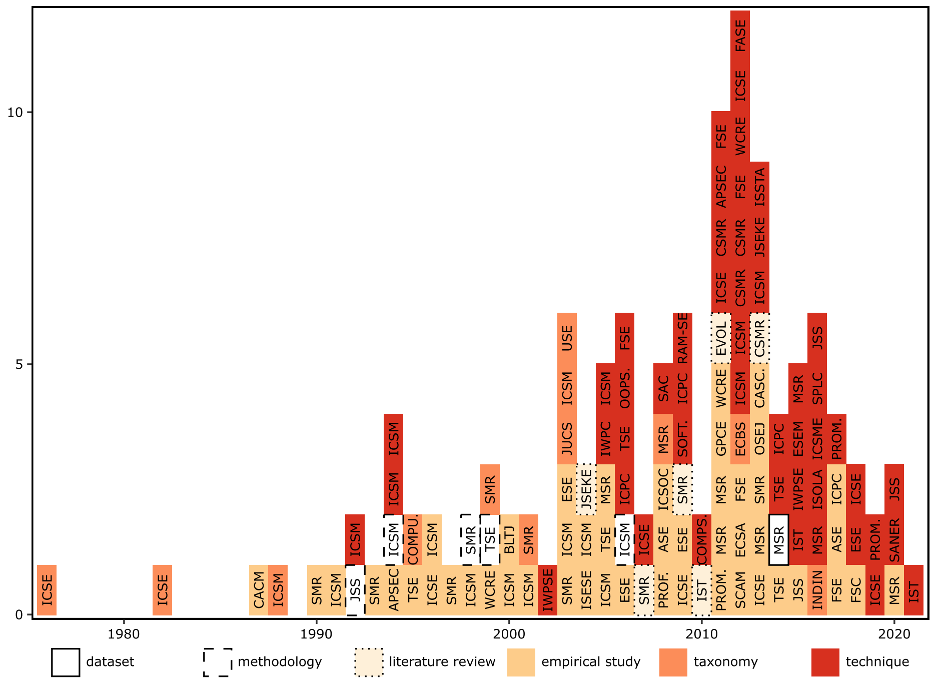


Fig. 3. Overview of the paper distribution in our dataset.

Taxonomies. Regarding the taxonomies related to SCIs that have been used in the publications, we can see that these stem from different sources. Most researchers rely on the taxonomy of maintenance activities proposed by Swanson or one of its extensions, or they extend it themselves. It was sometimes unclear whether a taxonomy was directly based on Swanson's work or stemmed from the terms being common knowledge or phrases (e.g., researchers referring to corrective changes).

When extracting the data, we checked whether the authors directly referred to their taxonomy as stemming from another work (e.g., Swanson), for instance, by stating so in the text or by putting a reference in the respective table. So some other taxonomies may have been based on a previous taxonomy without the authors being aware of it. Still, the overall picture of most publications extending or proposing a taxonomy as needed remains. This is also caused by the various contributions and goals of the publications, which are often concerned with more fine-grained SCIs (e.g., specific refactorings [159]) or require different categorizations. For instance, Hindle et al. [54] are concerned with empirically analyzing large commits (cf. Table 2). They extend Swanson's taxonomy, add a more fine-grained layer on the commit level (thereby mixing different levels of abstraction, e.g., bug fix on SCI level versus module add on implementation level), and further classifying commits based on size (i.e., "large").

Unfortunately, even the literature reviews that derive their taxonomies from various publications do not provide a feasible unification of SCIs (cf. Section 9). Primarily, these publications elicit different categorizations for changes that involve SCIs to some extent, but they do not tackle the problem of separating different categories of SCIs. When analyzing the publications, the missing consensus based on which we could understand SCIs and map them to different concepts made it challenging to compare the publications or identify their connections—leading to confusion and multiple iterations within our methodology (e.g., when deciding what is still in scope or to ensure the data quality). We address this problem by providing a systematically derived categorization of SCIs that clearly distinguishes different SCIs. So our categorization in Section 4 provides a foundation to facilitate the understanding, unify the description, and clarify connections of research on SCIs.

RO_1: State of the Art on SCIs

To provide an overview of how SCIs are used and studied in software-engineering research, we systematically selected 122 relevant publications. Our analysis of the publications indicates the following:

- *Most of these publications contribute either a technique (50) or an empirical study (48), with the rest contributing taxonomy proposals, literature reviews, methodologies, or a dataset.*
- *SCIs are used for a broad range of goals (e.g., comprehending software evolution, predicting maintenance activities, verifying the correctness of changes) aiming to achieve many benefits (e.g., improving automated analysis techniques, facilitating project monitoring).*
- *The taxonomies used to describe SCIs are often adapted as required, resulting in intermixing of synonymous, overlapping, or orthogonal categories.*

4 Moving Toward a Taxonomy (RO_2)

Most researchers build on their own definitions or taxonomies of SCIs, sometimes being inspired by or extending other works. In particular, the taxonomy of Swanson [148] is regularly reused and extended. However, the actual problem is that the extensions are often arbitrary and non-systematic. Some publications simply add categories as they see fit, resulting in incoherent, vague, and overlapping SCIs that act on different levels of granularity (e.g., high-level developer intentions versus code modifications versus specific activity types such as refactorings). There have been several previous attempts at describing SCIs more systematically by synthesizing categories through literature reviews [13, 62, 64, 88, 130, 166]. Unfortunately, as we discuss in Section 9, these works do not focus on the SCIs themselves but on related techniques (e.g., change-impact analysis) or various change attributes—which do not lead to a coherent understanding of SCIs. As a result, it was often challenging during our analysis to understand the relations between different taxonomies and identify which SCIs are related, highlighting the absence of a common understanding of SCIs.

4.1 Categorizing SCIs

During our analysis (cf. Section 2.3), we discovered that many of the SCIs were overlapping and that their aspects were named inconsistently. To move toward a language and taxonomy for specifying the key properties of an SCI, we separated overlapping SCIs while applying a uniform naming. We display our resulting categorization of SCIs in Figure 4, with solid boxes indicating the higher-level categories and dashed boxes exemplifying terms that are part of these categories and may yield more fine-grained categorizations (e.g., sub-categories of bug fix or refactoring SCIs). Our categorization contains five top-level categories: *goals*, *actions*, *objects*, *customer*, and *lifecycle phase*, which we describe in more detail in the following. Together, they specify and formalize a change by defining when (*lifecycle phase*), what (*objects*) parts of a system have been changed, how (*actions*), for what purpose (*goal*) of a specific actor (*customer*) and can be thought of as a language for specifying the key properties of an SCI. While in practice, developers and researchers may use a subset of these categories to describe the parts of a change that are relevant for them, missing information may lead to misuses, such as causing misunderstandings regarding the goals of a change

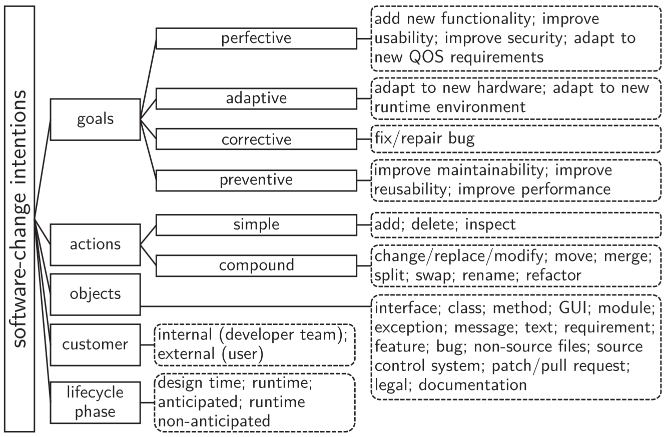

software-change intentions
goals
perfective
add new functionality; improve usability; improve security; adapt to new QOS requirements
adaptive
adapt to new hardware; adapt to new runtime environment
corrective
fix/repair bug
preventive
improve maintainability; improve reusability; improve performance
actions
simple
add; delete; inspect
compound
change/replace/modify; move; merge; split; swap; rename; refactor
objects
interface; class; method; GUI; module; exception; message; text; requirement; feature; bug; non-source files; source control system; patch/pull request; legal; documentation
customer
internal (developer team); external (user)
lifecycle phase
design time; runtime; anticipated; runtime non-anticipated

Fig. 4. Our categorization of SCIs.

or the customers for which it was implemented. Note again that our categorization in Figure 4 is not intended to be complete. Instead, as we argue in Section 4.2, by focusing on the most important properties of an SCI, we produce an extensible foundation for describing and comparing research.

Goals. This category is most commonly discussed in existing publications, building directly on the taxonomy by Swanson [148]. It describes the purpose for which developers perform a change. We divide goals into four sub-categories:

— *Perfective* changes (a.k.a. enhancements) focus on the evolution of requirements and features that exist in a software system. Such changes include, for instance, adding new functionality requested by the users, improving the usability or security of a system, or implementing new quality of service requirements.

— *Adaptive* changes are important when the environment of a system changes, which can be caused by changes to, for example, the operating system, hardware, or dependencies. Adaptive changes can also include changes reflected by organizational policies and rules.

— *Corrective* changes address errors and faults in a system. These changes usually originate from bug reports that were created by users or from internal reviews done within an organization.

— *Preventive* changes help developers improve their system and prevent its deterioration, so that it can function for a longer period of time. These changes include, for instance, improving software understandability and maintainability, addressing the accumulated technical debt, eliminating performance bottlenecks, or making parts of the system reusable.

Using these four categories, we can distinguish the different goals a SCI can have.

Actions. This category describes the concrete activities developers perform to achieve their goals. We further split actions into two sub-categories:

— *Simple* actions include adding, deleting, and inspecting an element of a system (e.g., lines of code, modules, interfaces, and classes).

— *Compound* actions are higher-level operations performed by combining multiple simple actions, such as changing, moving, or renaming a set of elements (i.e., deleting and adding); merging, splitting, or swapping multiple elements (i.e., inspecting, deleting, adding); or performing a refactoring using a combination of these actions.

These two categories are well established for specifying actions and operations and serve the same purpose in our taxonomy.

Objects. This category describes the elements that are manipulated by an action. Due to the complexity of modern software systems, the objects include a wide range of software artifact, such as interfaces, classes, methods, and exceptions; GUI elements; requirements, features, and tests; source and non-source files; documentation; or source control systems and patch/pull requests. Due to this wide range, it is challenging to define a useful sub-categorization that would be applicable for every type of research on SCIs.

Customer. With this category, we cover the individual or entity due to which an SCI has been initiated. Essentially, we can distinguish between internal (i.e., development team) or external (e.g., end user, legal body) customers. Still, a more fine-grained categorization of customers may be needed for some research on SCIs.

Lifecycle Phase. Our last category specifies at what time an SCI is performed. In the literature, we identified different examples that relate to this category, for instance, SCIs that occur at design time or at runtime—and that may be anticipated or not [161]. So, this category provides an understanding of the relation of an SCI to the lifecycle of its respective system.

Example for Categorizing Changes. We argue that these five categories are well suited to describe the most important properties of SCIs that cannot be easily covered by metrics. As an example of using the categories for describing changes, let us assume a stakeholder finds and reports a bug in a system that a developer is trying to fix—connecting to the concepts related to high-level intentions we sketch in Figure 2. Now, the developer implements a change, for instance, in a separate fork of the system, and creates a pull request. Following our categories, they could specify within the pull request that they implemented a corrective change (*goal*) due to the stakeholder's request (*customer*), modifying a conditional statement (*compound action*) to a code file (*object*) while the system is already operational (*lifecycle phase*). In a second pull request (or within the same one), the developer may add new (*simple action*) test cases (*object*) to ensure that other developers (*customer*) can ensure the system's future operation (*lifecycle*) through preventive maintenance (*goal*). Of course, the developers of the system have to agree on what level of granularity they document this information, what details they cover, and whether they want to document all of these categories. However, we argue that these five categories cover the most relevant pieces of information for describing an SCI, which can help document these more reliably and consistently, thereby also allowing to design new techniques to support developers.

4.2 Comparison to Existing Taxonomies

Our categorization is not a synthesis of all taxonomies we identified in the 122 publications. Instead, we aimed to define a concise categorization for specifying SCIs, providing an extensible (e.g., more fine-grained sub-categories) foundation for describing and comparing research. We argue that such a concise overview is much more helpful than a more detailed, but still incomplete, taxonomy that involves too many fine-grained levels. Moreover, our categorization represents the most important properties of an SCI and thereby covers the relevant properties of existing taxonomies.

To further exemplify our categorization, we summarize the 19 publications that extend or adapt the taxonomy of Swanson [148] in Table 4. Please note that Swanson's taxonomy itself is subsumed by the category *goal* in our categorization. In Table 4, each bullet refers to a taxonomy defined in the respective publication, for instance, Hindle et al. [53] extend Swanson's taxonomy with "feature addition" as well as "non-functional" SCIs and complement it with another taxonomy for large changes that involves SCIs, such as "legal," "module management," and "maintenance." This is an example for the inconsistencies of taxonomies that are used within and between different publications. Concretely, the extension of "non-functional" in the first taxonomy overlaps and is inconsistent with the other four SCIs: It covers legal aspects (e.g., copyright), source control system management (e.g., branching, tagging), and code clean-up. While the former two are not

Table 4. Overview of the Taxonomies We Identified as Extending the One of Swanson [148] (i.e., Those Marked as "[148]+" in Previous Tables)

ref	contr	taxonomy
Swanson [148]	TP	• adaptive; corrective; perfective
[22]	ES	• adaptive; corrective; perfective; preventive
[35]	ES	• adaptive; corrective; new application
[42]	ES	• adaptive; corrective; perfective; preventive
[47]	ES	• corrective; forward; management; re-engineering • tiny; small; medium; large
[51]	ES	• adaptive; corrective; documentation; perfective; refactoring; other
[63]	ES	• adaptive; corrective; perfective; preventive • introduction/deletion of module; change of interface, control flow, data declarations, data, or assignment statement
[87]	ES	• corrective; enhancive (adaptive, perfective, preventive)
[107]	ES	• adaptive; corrective; inspection; perfective
[109]	ES	• adaptive; perfective (functional; quality attributes); preventive
[121]	ES	• adaptive; non-urgent corrective; perfective; preventive; urgent-corrective
[123]	ES	• adaptive; corrective; inspection; perfective • delete; insert; modify
[132]	ES	• adaptive; corrective; enhancement; perfective; preventive
[142]	ES	• adaptive; corrective; perfective; preventive • small; medium; large
[19]	ME	• adapative; corrective; enhancement; preventive
[24]	TP	• adaptive; corrective; enhancive; groomative; performance; preventive; reductive
[54]	TP	• adaptive; corrective; implementation; non-functional; perfective • branch; bug fix; build; clean up; cross; data; debug; documentation; external; feature add; indentation; initialization; internationalization; legal; maintenance; merge; module add, move, remove; platform specific; source control; refactoring; rename; testing; token replace; versioning
[95]	TP	• adaptive; corrective; documentation; pretty printing; retrenchment; retrieving
[162]	TP	• adaptive; code leverage; corrective; perfective; reuse
[53]	TE	• adaptive; corrective; feature addition; non-functional; perfective • feature addition; legal; maintenance; meta-program; module management; non-functional source code changes; SCS management

A line starting with a bullet (•) indicates a taxonomy used in the respective publication; for instance, Hattori and Lanza [47] distinguish between maintenance activities (e.g., "corrective") and change size (e.g., "tiny"). SCIs in parentheses are sub-categories of the SCI directly before the parentheses; for instance, Lee and Jefferson [87] summarize "adaptive," "perfective," and "preventive" as "enhancive."

related to actual code changes, code clean-up is covered as the goal of an SCI (i.e., preventive). The second taxonomy keeps some of the SCIs (e.g., "feature addition"), while also summarizing (e.g., "maintenance") and splitting (e.g., "legal") others. As a consequence, it was challenging for us to understand the relations between SCIs within and between publications.

We argue that our categorization can help researchers improve the comprehensibility and comparability of their research. Particularly, note that even though the taxonomies in Table 4 are all based on the one of Swanson, almost each one adds a highly individual naming for different SCIs, ignores some SCIs, or introduces concepts unrelated to SCIs. Our categorization can help resolve such inconsistencies, since its high-level categories provide a common ground for specifying the context and coverage of SCIs in a publication. For instance, "enhancement" [132], "feature addition" [53], and "feature add" [54] all refer to extending a system with novel functionality and thus are covered by the categories goal (i.e., perfective: adding new functionality), action (i.e., simple: add), and objects (i.e., feature). Furthermore, our categorization resolves inconsistencies, such as that concepts (e.g., "branch," "bug fix," "module add" [54]) are part of the same taxonomy—even though they are not describing the same abstractions or do not relate to SCIs. For example, we removed "branch," since it does not relate to a change of the source code and separated "bug fix" (goal) from "module add" (action and object). Consequently, we removed concepts, such as size measures [47, 142], "inspection" [123], "branch" [53, 54], or "urgent" [121], since these do not describe SCIs (e.g., changes to the actual source code). Overall, our categorization covers the most important properties to specify SCIs, which helps distinguish SCIs and make the corresponding research more comparable.

RO$_2$: Categorization of SCIs

To define a categorization of SCIs, we performed open and axial coding on 40 publications, which we randomly selected from our sample. The main benefits of the resulting categorization are as follows:

— *It provides a systematic view of concepts related to SCIs, defining five orthogonal categories (goals, actions, objects, customer, lifecycle phase).*
— *It includes examples of concepts in each category, easing communication and facilitating a shared understanding of SCIs.*
— *It defines a common level of abstraction and allows extensions with more orthogonal categories and/or more fine-grained levels.*

5 Assessing the Benefits (RO$_3$)

The 122 publications we reviewed had different goals and benefits. Not surprisingly, the overarching theme of all of these is to support or facilitate software evolution, which is why there are various overlaps and strong connections between the publications. In contrast, we can see in Tables 1, 2, and 3 that only a small number of the publications (20) provide any form of real-world evidence on the claimed benefits of knowing SCIs. We remark again that this does not mean that the contributions in the publications have not been evaluated by some other means (e.g., comparisons against other techniques or a dataset). Still, a transfer into practice, for instance, as a field experiment, case study, tool, or any other practical evaluation, to study the benefits for actual software engineers has rarely been reported. In this section, we first discuss the goals and benefits claimed in the reviewed publications before summarizing the available evidence to support these claimed benefits of knowing SCIs.

5.1 Goals and Benefits

The most frequently mentioned goals of researching SCIs in the publications are as follows: comprehending software evolution (29), labeling changes with SCIs (22), predicting some software-evolution phenomenon (e.g., bugs) (20), and providing a taxonomy for understanding software changes (18). Moreover, SCIs have been considered in more specific, highly interesting research directions, such as verifying software (5), transplanting code (2), or even programming (1). In the following, we discuss the individual goals and benefits the authors claim knowing or using SCIs could have with respect to such goals. Since the general idea (i.e., knowing SCIs) and overarching goal (i.e., supporting software evolution) are identical, there are many similarities between the publications.

Comprehension (29). This goal assumes that knowing SCIs helps developers comprehend changes more easily. In contrast to understanding low-level code changes (i.e., program comprehension), a developer comprehends the higher-level SCI (e.g., what is the purpose of the change and what are the underlying assumptions)—with research showing that developers consider such more abstract knowledge as more important [77, 78, 131]. For example, Greevy et al. [41] propose a technique that allows us to analyze the evolution of a system to help developers identify feature changes and refactorings more easily. However, this goal also involves many publications that are concerned with fundamental research questions only; helping researchers provide a better comprehension of software evolution without direct practical impact. For instance, Hattori and Lanza [47] investigated 72,351 commits from nine projects to comprehend what constitutes small or large commits, which of these involve most developer activities, and whether large commits are concerned with code management. To address particularly the second question, the authors adapted the classification of Swanson [148] to comprehend different developer activities. Overall, the main benefits claimed for this goal are as follows: improving the comprehension of software evolution, monitoring projects (e.g., analyzing workload distributions [1]), predicting bugs (e.g., comprehending which changes before a corrective one introduced the bug [140]), comparing research (e.g., comparing efforts for different SCIs [134]), and identifying software-evolution problems (e.g., checking whether quality attributes are still fulfilled [136]). Note that how these benefits are achieved also depends on the type of contribution of a publication. For instance, methodologies propose how developers can comprehend software changes, empirical studies provide data for this purpose, and techniques present supportive tools. Last, we want to remark that this goal of comprehending evolution is very broad, spanning very different research directions. We decided not to split this goal further, because (1) the boundaries between more fine-grained goals were often vague and (2) the authors of the papers themselves regularly mentioned this high-level goal with respect to knowing SCIs. Please note that the goal and benefits of the actual work, which may only build on SCIs, are often involving additional goals. By not splitting up this goal further, we aimed to contribute a concise overview that does not involve many overlapping and interconnected sub-goals.

Labeling (23). This goal builds on the same assumption as the previous one, namely that knowing SCIs is helpful. However, research on this goal is only concerned with labeling changes according to some taxonomy of SCIs, but the results are not directly used for any other purpose (e.g., for comprehending software evolution). The results of such a labeling can help researchers or practitioners achieve the other goals. It is not surprising that most publications related to this goal propose techniques (19). For instance, Tian et al. [154] propose a technique for identifying bug-fixing (i.e., goal: corrective) changes and compare their technique against other such labeling techniques. There is no follow up use of the SCI labels in the publication itself, but it is motivated to help with transplanting corrective changes into other code locations or even projects (e.g., for code transplantation and automatic program repair). To motivate the labeling of software changes, several benefits are discussed that strongly relate to other goals and their benefits. For example, labeling changes with SCIs can help researchers and practitioners improve project monitoring (e.g., allowing to add missing labels to changes [45] or identify mislabeled changes [51]), comprehending software evolution, filtering changes based on their SCIs to facilitate analyses [53], transplanting bug fixes, estimating maintenance efforts (e.g., identifying changes causing large efforts during reviews [163]), or predicting bugs (e.g., linking changes and bug reports [168]).

Predicting (22). Various researchers have aimed to use SCIs to enable or improve predictions on evolving software, which is sometimes closely related to labeling. Typically, such works aim to use SCIs to improve an existing analysis (e.g., change-impact analysis [146]) or build on a set of changes labeled with SCIs to predict occurrences of a phenomenon or the same SCI (e.g., corrective changes are used to identify bug-introducing changes to then predict future bugs [71]). For instance, Tang et al. [149] proposed a machine learning–based technique that allows to predict whether a corrective change may break regression testing. Dagenais and Robillard [26] designed a technique that aims to keep software documentation in sync with source code changes by predicting whether documentation must be updated after a change. Most of the mentioned benefits of knowing SCIs for predictions are highly intuitive, such as predicting types of changes and their economic impact on projects [10], predicting bugs to prevent these [46], predicting (breaking) changes to identify problems [67], predicting updates to other artifacts (e.g., documentation), or predicting software quality to ensure the system's maintainability [22]. Please note that the goals of labeling and predicting are closely connected and interrelated, for instance, SCIs may be labeled with a technique and then used for some prediction. We refer to labeling if the primary focus of a publication is to identify (or "predict") SCIs, whereas our goal of predicting typically refers to publications that use such labels to predict some other phenomenon. For the previous example, this goal would typically be predicting and not labeling (e.g., because an existing technique was reused to label SCIs and these were the underlying idea for the prediction).

Taxonomy (18). Several publications have been concerned with deriving a taxonomy or similar classification of software evolution that involves SCIs (cf. Section 9 for a detailed comparison to our own meta-study). Such publications are primarily focused on researchers, indicating that understanding the properties and SCIs of changes can help them to study a certain problem domain. For instance, the researchers argue that using their taxonomies, and thus SCIs, can help compare research (e.g., on software evolution [62], mining software repositories [64], or change impact analysis [88]) and formalize software evolution (e.g., defining an ontology of software maintenance [130]). Other benefits mentioned are closely related to these two, and to the benefits of other goals, such as identifying problems, comprehending software evolution, improving techniques, or supporting effort estimations.

Untangling (6). Several researchers experienced that changes can involve multiple, tangled SCIs, complicating automated analyses and program comprehension. This resulted in the research goal of untangling different SCIs that are part of a single change, aiming to improve version histories and support the comprehension of software evolution. For instance, Kirinuki et al. [72] propose to derive templates of changes (i.e., specifying SCIs) from past changes to warn developers if they commit tangled SCIs. In a similar direction, Matsuda et al. [100] propose a technique to automatically separate refactoring SCIs (i.e., goal: perfective) and Hayashi et al. [48] propose refactorings to enable developers to untangle and reorder changes based on their SCIs.

Specification (5), Verification (5), and Visualization (3). We describe these three goals together, because they build on the same underlying benefits: analyzing evolution patterns and ensuring that software evolution is save. For instance, Sampaio et al. [133] propose and formalize evolution templates for product lines, which represent different SCIs developers may have. Ensuring these templates during the evolution of a product line would allow to guarantee that products not affected by the changed code keep their behavior. Among others, Hou and Hoover [56] build on the same idea and essentially propose to use the Structural Constraint Language to specify constraints of the code as conditions—particularly non-functional design intentions that should always be fulfilled. During software evolution, it can be checked whether a software change still fulfills these conditions. Such research essentially adopts the concept of SCIs in the context of software verification. Similarly, other researchers proposed visualizations, primarily to check whether changes still fulfill underlying intentions of the code. For instance, Jackson and Ladd [61] proposed Semantic Diff as a tool to summarize SCIs and allow a developer to compare these against their original SCI (e.g., whether a change identified as corrective was intended to be corrective).

Mining (1), Programming (1), and Transplantation (2). While these three goals are strongly connected to some of the previous ones, they are different and highly interesting in terms of the research proposed. Specifically, they rely on the concept of SCIs for implementing or improving a concrete technique. Hashimoto and Mori [44] propose a concrete technical improvement for concern mining by considering SCIs. Dhaliwal et al. [28], propose to group changes with the same SCI (e.g., a specific goal, such as corrective or perfective) to transplant them safely and Lillack et al. [94] define SCIs as operational concepts for transplanting changes. In both cases, avoiding errors is the claimed main benefit. Finally, Simonyi et al. [139] propose to use intentions as the primary concern for implementing software systems to enable domain experts to design software. So, we argue that these techniques are taking more concrete steps of integrating SCIs into the actual engineering and evolution of software than most other publications we identified.

Discussion. Overall, we can see that the different goals and benefits are, not surprisingly, closely related. However, most of the contributions are more intended to support researchers instead of practitioners. For instance, labeling changes with SCIs and taxonomies of SCIs are used to compare research and enable certain analyses. The claimed benefits for practitioners are rarely supported by actual practical evidence. In contrast, few publications are investigating how to incorporate SCIs into software engineering to directly benefit practitioners, for instance, by improving code transplantation or program verification. Note that this does not mean that these publications do not have practical applicability, but their concrete usability seems more abstract and is rarely shown. One recurring problem in this regard are the completely different paths, notations, and levels of detail at which researchers consider SCIs. For example, some explicitly refer to SCIs while others refer to various sub-categories; and the same terms in two publications can refer to different SCIs. We relied heavily on our understanding of the research area as well as our classification to classify and structure the goals and benefits. For instance, for labeling changes, our classification provides a common understanding of what is labeled (e.g., goals) or what may be relevant to fully describe an SCI (i.e., what goal is fulfilled by what actions on what objects for what customer in what lifecycle phase). In fact, our classification directly addresses the most often mentioned goals and benefits by improving our comprehension of software evolution (research). So, we consider our classification as a helpful means for researching, communicating, and using SCIs—thereby avoiding confusion and illustrating connections between individual works.

5.2 Evidence

As discussed, we identified various intuitive goals and benefits claimed to motivate the use of SCIs in software engineering. However, many of these goals are only substantiated by showing that a technique works as intended, by improving such a technique compared to other techniques, or by sketching how a problem could be tackled with the consequent knowledge of SCIs. What is often missing is an actual transfer into practice to study whether these goals and benefits are relevant to practitioners. In the following, we summarize and discuss the 20 pieces of practical evidence we found on the benefits of knowing SCIs in software engineering. Note that these pieces of evidence are often either highly specific or too vague, which is why we can only provide a general intuition on the benefits of knowing SCIs.

Practices (5). Some publications report on case studies conducted in companies or describe a company using categorizations of SCIs for project monitoring, change management, or effort estimations [1, 2, 19]. Unfortunately, the researchers do not provide insights on the concrete benefits of having this particular knowledge. Similarly, Mockus and Weiss [108] report that a methodology involving SCIs to assess the risks of software changes is applied in a company to inform developers about risks, but actual empirical data are missing. Rombach et al. [128] report on using SCIs for measuring maintenance activities at **Software Engineering Laboratory (SEL)**—a joint venture of NASA, University of Maryland, Computer Sciences Corporation—including a concrete template for specifying SCIs based on the taxonomy of Swanson [148]. The authors use this information to understand the effort distributions within the company, and express that such information helped the SEL comprehend its software development and maintenance. While missing actual data, we argue that the companies must have seen a benefit in knowing SCIs to implement such practices.

Surveys (6) and Correlations (1). Several researchers conducted surveys to understand specific types of SCIs. Yet, we were unable to identify any that directly investigated the benefits of knowing different SCIs. However, the following surveys indicate that certain SCIs impact developers' perceptions, so knowing SCIs would have practical benefits (e.g., supporting developers' program comprehension, debiasing misconceptions, providing tools for untangling SCIs in one change).

Kim et al. [70] performed a survey with Microsoft developers, asking them for their definitions and perceptions of what constitutes a refactoring. The findings indicate that the developers had various definitions of refactoring that are not in line with the original one (i.e., that refactoring SCIs are often tangled with other goals, namely corrective or perfective ones). Moreover, the developers argue that refactorings are associated with various risks (e.g., breaking changes, merge conflicts) and benefits (e.g., improved readability, fewer bugs) that only partly correspond to this SCI. This indicates that developers could benefit from more clearly separating between different SCIs when committing changes, thereby mitigating risks (e.g., refactorings should not cause breaking changes). Researchers can build on these insights to study whether knowing the SCI of a change mitigates the risks identified or makes developers more aware of their actual causes.

Tao et al. [151] report a survey with interviews involving 180 Microsoft developers to research the importance of comprehending software changes. Their results indicate that the question of the rationale behind a code change (i.e., the SCI) is perceived to be the most important piece of information—but also the one that is easy to recover. However, the difficulty rating builds on the assumption that the description of the change (e.g., commit message) is available. Some issues the participants mention to challenge the comprehension of SCIs involve low quality of the description, tangled changes, and missing links to additional metadata.

Lientz et al. [93] conducted a survey to study software maintenance and enhancement, for which they received responses from 69 organizations. They built on the taxonomy of Swanson to elicit how often each type of maintenance activity occurs in practice. The findings of Lientz et al. support the assumption that software maintenance is the most expensive activity for organizations. Moreover, organizations seem to be aware of the different goals of SCIs and particularly perfective SCIs (new functionalities) seem to cause most management problems. Consequently, knowing specific SCIs could, in fact, support organizations in improving their project management by adapting their processes accordingly (e.g., identifying and solving concrete problems associated with an SCI). Similarly, Nosek and Palvia [113] replicated a previous survey on software maintenance and obtained essentially the same results, indicating, for instance, that demands for perfective SCIs, software changes, and management support cause problems. A direct analysis of the relations of SCIs and consequent problems in software projects (i.e., technical and management) could provide more detailed insights and practical evidence. Other surveys [142, 173] in this direction focus on studying the costs or activities of software maintenance based on taxonomies of SCIs, indicating that the distinction of SCIs may help practitioners improve their comprehension. Similarly, Briand and Basili [17] aim to predict the effort of software changes by using correlations to show that the SCI is an indicator for such efforts—thus, together with the previous surveys, indicating that knowing SCIs may benefit effort estimations.

Correctness (5). Dhaliwal et al. [28] use metrics to find changes that belong together (i.e., representing one SCI) to facilitate code transplantation in product lines. They conduct a study with automatic and developer-guided transplantations, yielding 76% and 94% fewer failures, respectively. In the same direction, Lillack et al. [94] define six SCIs as concrete operators that automate the integration of code changes between different variants of a product line. Conducting a user study with 12 developers, they show that the operational SCIs result in fewer integration errors (e.g., seven compared to 17.5 for a standard merge tool). Hashimoto and Mori [44] propose a technique for locating concerns in version histories, relating changes and considering SCIs to improve their technique. The authors conduct a simulation study that indicates a precision from 53.3% to 71.9%. Silva et al. [137] describe a technique for identifying code locations where perfective software changes should be executed. Evaluating their technique on four case studies, Silva et al. find that the precision of suggesting locations for perfective changes ranges from 77% to 100%. Last, Wang et al. [163, 164] report on using SCIs to identify changes that require large reviewing efforts. They evaluate their technique based on four projects, with the results indicating that considering SCIs improves the technique's performance by up to 19% (7.4% on average). These findings represent arguably the most interesting and most reliable empirical evidence in our dataset. More precisely, it seems that knowing what changes belong to the same SCI or even operationalizing SCIs can reduce the number of bugs, and thus costs, of integrating or transplanting code changes.

Payoff (1). Rostkowycz et al. [129] describe experiences of re-documenting a software system in an organization. Any iteration of re-documentation was triggered by a number of components of the system having been changed and reaching an "error free" status of the system. The documentation included the intentions of the software components and of the individual software changes. While the precise impact of knowing SCIs is not reported, this knowledge was still part of the documentation, and the authors report that the re-documentation payed off after 1.5 years. Replicating such a study with a focus on SCIs only would help understand the actual benefits of knowing SCIs; for instance, by introducing explicit documentation for SCIs based on our taxonomy or studying projects that have a related taxonomy of SCIs for issues.

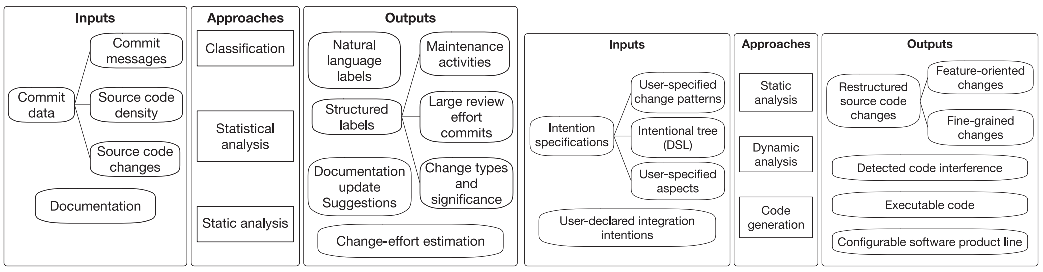
Feedback (1), Usability (1), and Understandability (1). Hou and Hoover [56] collaborated with industrial developers to evaluate their technique for specifying constraints for changes as intentions. Unfortunately, the authors do not report an empirical study with these developers, they only state that their collaborators perceived the technique as helpful. Qi et al. [124] build on the same idea, defining software-change contracts to ensure that the SCI behind a change is fulfilled. The authors asked two students to write different contracts and report primarily on the usability of the tool from the students' perspectives, indicating that both were quite successful in using the corresponding tool (i.e., they could write 52 contracts for 57 changes). Finally, Yi et al. [172] extend the previous paper, reporting a user study of 16 students who had to modify, comprehend, and write change contracts. Overall, the results indicate that the students could easily understand change contracts (86–100% correct responses). The results of the first two studies are only anecdotal but hint in the same direction as the user study reported in the third publication: Specifying SCIs to verify changes seems to be an interesting and intuitive way of checking software changes.

Discussion. In summary, these pieces of evidence are not providing a good understanding of whether knowing SCIs really helps practitioners. Consequently, our impression remains that much of the existing research in this area is oriented toward other researchers. We argue that more empirical studies are required that investigate the impact and potential benefits that knowing SCIs can have for practitioners. For instance, consider the idea of untangling SCIs. While it is intuitive that tangled SCIs could be refactored, for example, to facilitate cherry-picking and improve other techniques, the actual benefits of these techniques for practitioners have not been explicitly investigated. Unfortunately, it is challenging to collect reliable empirical evidence on the benefits of knowing SCIs in practice, particularly since there has been no common understanding of their properties. Our meta-study and classification can help researchers design empirical studies more systematically to improve our confidence that knowing (certain categories of) SCIs is relevant for practice. Moreover, our classification helped us structure our comparison of the publications' insights, providing guidance for analyzing their different perspectives and levels of abstractions. Still, further research to investigate whether our classification provides a common ground that helps researchers and practitioners in their work is needed.

RO$_3$: Reported Benefits and Evidence

To understand the usefulness of knowing SCIs, we analyzed the reported benefits and the supporting evidence provided in our sample of 122 publications. Our insights are as follows:

- *A primary benefit of knowing SCIs is being able to comprehend, classify, and compare software-evolution research.*
- *Collecting empirical evidence on the benefits of knowing SCIs in practice is difficult.*
- *Comparing between the publications and composing empirical evidence reported in them is challenging, since they are defined at different levels of abstraction.*



(a) Identify developers' SCIs. (b) Use SCIs to improve other techniques.

Fig. 5. Our classification of SCI-based techniques.

6 Analyzing the Techniques (RO$_4$)

Recall that we have classified 50 publications as techniques concerned with SCIs in Table 3, which identify SCIs (▲), use SCIs to improve other techniques (▼), or have SCIs as an intermediate result (◆). Next, we describe the first two types of techniques in more detail. In Figure 5, we illustrate our classification of the techniques according to their input and output artifacts, underlying approach, as well as the way SCIs are used. Specifically, these techniques build on the concept of SCIs in two ways: (i) *identifying developers' SCIs from software development artifacts* (cf. Figure 5(a)) or (ii) *using SCIs to improve the effectiveness of other techniques* (cf. Figure 5(b)).

Techniques of the former type rely on various software development artifacts as inputs to determine developers' SCIs, for instance, commit messages [53, 107, 132], source code changes [34, 41, 91], source code density [55], or documentation [26]. The outputs of these techniques are SCIs represented using natural language labels [41, 53, 91, 132], structured labels (e.g., based on the category *goals* from our categorization) [55, 107], change types as well as change significance [34], and documentation edits [26]. Most commonly, the underlying approaches build on static analysis [99] or statistical analysis (including data mining and machine learning), such as clustering [107], classification [163, 164], and keyword frequency analysis [132].

Techniques of the second type use SCIs as inputs. There, SCIs are typically represented in the form of user-specified change patterns [99], user-specified aspects in an aspect-oriented languages [98], intentional trees defined using domain-specific languages [139], or user-declared integration SCIs for forked product variants [94]. These SCIs are then used to (re-)structure source code changes, for example, producing more fine-grained changes [100] or refactoring code changes to be feature oriented [30]. SCI specifications can also be used to detect unanticipated code interferences between different aspects [98], to automatically generate executable code [139], and to reverse engineer product variants into configurable software product lines [94].

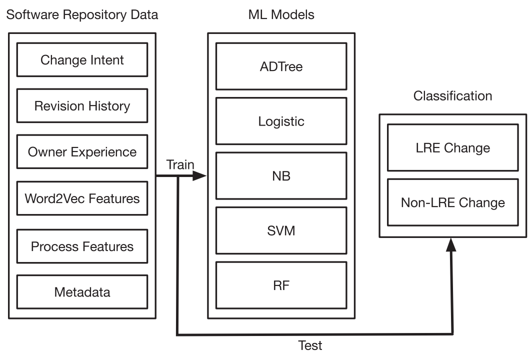
In the following, we walk through some representative SCI-based techniques according to their underlying approaches. Specifically, the second and fourth authors picked these examples based on their overview understanding of which and how many techniques rely on which underlying approaches. We provide an overview of the mapping between techniques and approaches in Table 3.

6.1 Statistical Analysis and Machine Learning

Approaches related to statistical analyses, especially machine learning, are typically used to identify SCIs. The inputs to these techniques are software artifacts, such as commit messages, commit metadata (e.g., source code density [55]), and source code changes. The outputs are the identified SCIs, usually in the form of natural language labels or structured labels.

SCI	description	heuristics
Bug Fix	Changes are made to fix bugs	1. The commit message contains keywords: "bug" or "fix" AND 2. The commit message does not contain keywords: "test case" or "unit test"
Resource	Changes are made to update non-source code resources, configurations, or documents	1. The commit message contains keywords: "conf" or "license" or "legal" OR 2. If no keyword is matched in step 1, the changed files do not involve any source/test files
Feature	Changes are made to implement new or update existing features	1. The commit message contains keywords: "update" or "add" or "new" or "create" or "add" or "implement feature" OR 2. Changes in the 'Other' category that contain keywords: "enable" or "add" or "update" or "implement" or "improve"
Test	Changes are made to add new or update existing test cases	1. The commit message contains the keyword: "test" OR 2. The changed files contain only test files or resource files
Refactor	Changes are made to refactor existing code	The commit message contains the keyword: "refactor"
Merge	Changes are made to merge branches	The commit message contains keywords: "merge" or "merging" or "integrate" or "integrated" or "integrated"
Deprecate	Changes are made to remove deprecated code	The commit message contains keywords: "deprecat" or "delete" or "clean up"
Auto	Changes that are committed by automated accounts or bots	The change is submitted by automated accounts or bots
Others	Changes that are not in any of the above categories	-

(a) Heuristics for categorizing SCI.



(b) The overview of the LRE change classification.

Fig. 6. An example of the machine learning technique proposed by Wang et al. [164].

For example, Wang et al. [163, 164] applied machine learning-based classification techniques, including Alternating Decision Tree, Logistic Regression, Naive Bayes, Support Vector Machine, and Random Forest, to classify changes according to their SCIs. The ultimate goal of this work was to label commits according to their estimated review efforts—**large-review-effort (LRE)** changes typically take more time, more reviewers, and more iterations to resolve all suggestions made by reviewers. The authors found a strong correlation between the SCI with the review effort of the changes, meaning that changes with certain SCIs are more likely to take more review iterations. Therefore, they proposed to identify SCIs as an intermediate result, which is then used to facilitate the identification of LRE changes. In Figure 6(b), we illustrate the workflow of the technique by Wang et al. The input to the machine learning models is the change metadata, such as the revision history of a file, committers' experiences, and the SCIs identified through an *intent* *analysis*. The trained machine learning models are then used to classify commits into either LRE or non-LRE changes.

The intent analysis heuristically identifies SCIs by searching for keywords in commit messages (e.g., "fix," "refactor," and "feature"). We summarize the nine types of SCIs, their descriptions, and the heuristics used to automate the classification process in Figure 6(a). For example, the commit messages of changes related to "Test" usually contain a keyword "test," and the changed files contain only test files or resource files. These heuristics were initially summarized manually and subsequently refined using a feedback-driven technique. If the accuracy of some heuristics is lower than 80% on a test sample, then they are refined by adding new heuristics or adjusting existing ones. One interesting finding from this study is that software changes are unevenly distributed regarding SCIs. Changes with some SCIs, such as "Feature" and "Refactor," have a higher probability of being LRE changes. Through extensive experimentation using different machine learning models, the authors found that among the examined classifiers, Logistic Regression and Random Forest achieve good AUC scores [16], which confirms the feasibility of identifying LRE changes by using machine learning algorithms and SCIs.

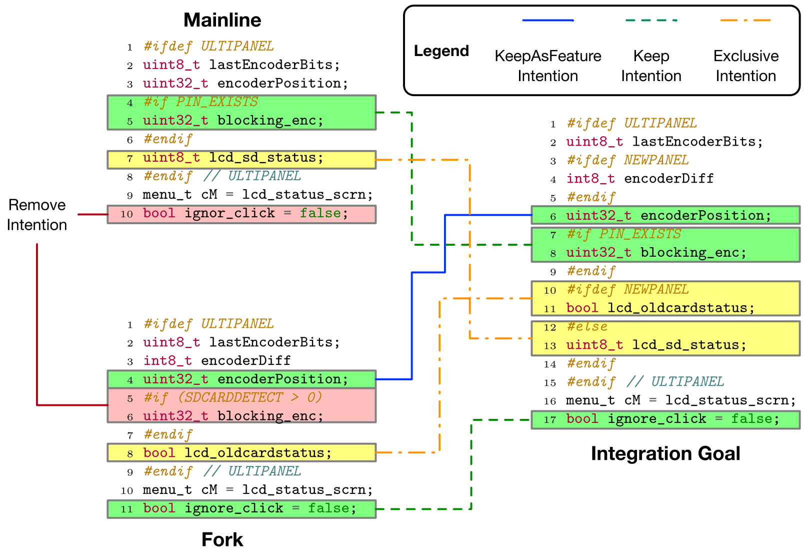
Other works of this type follow similar ideas, and many of them produce SCIs as their outputs. Hindle et al. [53] proposed a machine learning technique that automatically classifies commits into SCIs based on the commit messages and author identities. The training set was built on the commit history data of nine open source projects. The features used for classification include word distribution, author identity, and module/file types. Multiple machine learning algorithms were used, including Nearest Neighbor, Naive Bayes, Support Vector Machine, tree-based learners, and rule-based learners. Levin and Yehudai [91] proposed a technique that automatically classifies commits into SCIs using source code changes and commit messages. The training set was built on top of the version histories of a set of popular Java repositories on GitHub. The classification was based on keyword frequency, with the goal of creating a model of high accuracy and Kappa value. The machine learning models used include Random Forest, Gradient Boosting Machine, and J48. Hönel et al. [55] proposed an automatic commit classification technique based on source code density measure. The training set was built using 359K commits, where 1,149 commits had SCI labels (i.e., adaptive, corrective, and perfective). The features used for classification included code change-related features, such as the number of added and deleted files in a commit or code density.

6.2 Static Analysis

Static analysis approaches are generally used to identify developers' SCIs, elicit change types, and refine existing techniques with the help of SCIs. The input to these techniques includes source code changes, binary file changes, user-specified change patterns, and forked product variants. The output includes the identified code changes, change types, and program entities affected by changes, depending on the goal of the technique.

For example, Lillack et al. [94] proposed to leverage user-provided *integration intentions* to alleviate the challenges in integrating software variants. Software variant integration is the process of building a configurable software product line [81, 120] from a number of software variants created through cloning—copying existing code and adapting it to new requirements by implementing new or modifying existing features. Variant integration is a challenging task, which requires a good understanding of all the variants, their differences as well as how they are aligned, and making design decisions on what to keep and what to remove [3, 27, 76, 82, 83, 94].

In Figure 7, we show an example that illustrates the variant integration technique called INCLINE. The inputs are two software variants shown on the left-hand side, namely "Mainline" and "Fork," and the desired output is on the right-hand side, namely "Integration Goal." Each code excerpt offers some configuration options implemented with preprocessor directives, such as `#if`



Mainline

```
1  #ifdef ULTIPANEL
2  uint8_t lastEncoderBits;
3  uint32_t encoderPosition;
4  #if PIN_EXISTS
5  uint32_t blocking_enc;
6  #endif
7  uint8_t lcd_sd_status;
8  #endif // ULTIPANEL
9  menu_t cM = lcd_status_scrn;
10 bool ignor_click = false;
```

Fork

```
1  #ifdef ULTIPANEL
2  uint8_t lastEncoderBits;
3  int8_t encoderDiff
4  uint32_t encoderPosition;
5  #if (SDCARDDETECT > 0)
6  uint32_t blocking_enc;
7  #endif
8  bool lcd_oldcardstatus;
9  #endif // ULTIPANEL
10 menu_t cM = lcd_status_scrn;
11 bool ignore_click = false;
```

Integration Goal

```
1  #ifdef ULTIPANEL
2  uint8_t lastEncoderBits;
3  #ifdef NEWPANEL
4  int8_t encoderDiff
5  #endif
6  uint32_t encoderPosition;
7  #if PIN_EXISTS
8  uint32_t blocking_enc;
9  #endif
10 #ifdef NEWPANEL
11 bool lcd_oldcardstatus;
12 #else
13 uint8_t lcd_sd_status;
14 #endif
15 #endif // ULTIPANEL
16 menu_t cM = lcd_status_scrn;
17 bool ignore_click = false;
```

Fig. 7. Software variant integration with developer specified SCIs [94].

or #ifdef. Normally, to derive the integrated version, developers have to explore different edit options, undo and redo changes, and iterate through this process several times. INCLINE semi-automates this process by allowing developers to specify their high-level integration decisions as SCIs, which frees them from low-level error-prone editing work. The integration SCIs used in Figure 7 include (1) *KeepAsFeature*, which specifies that Line 4 of "Fork" should be kept in the integrated code as a feature (Line 6); (2) *Keep*, which specifies that Lines 4 and 5 of "Mainline" should be kept; (3) *Exclusive*, which specifies that Line 7 of "Mainline" and Line 8 of "Fork" should be kept mutually exclusive and configurable by an #if-#else-#endif structure; and (4) *Remove*, which specifies that Line 10 of "Mainline" should be removed (due to a typo). With INCLINE, developers can easily experiment with different integration SCIs and the low-level editing work is carried out automatically. The generation of variant integration is supported by an automated transformation of the **abstraction syntax trees (ASTs)**. Specifically, each user-defined integration SCI corresponds to a partial functions transforming ASTs. These functions are applied on the ASTs following a certain order to properly handle interactions between SCIs.

Martinez et al. [99] proposed an AST-based technique that automatically finds change pattern instances (i.e., SCIs) in a codebase given a user-written change pattern specification. The technique represents versioning changes of a commit and change patterns at the AST level. It accepts as input a commit and a list of user-provided change patterns (e.g., a change of an if-condition expression, an addition of a method declaration), parses the AST difference in the commit, and determines whether the difference matches any pattern. Rayside et al. [125] proposed a change impact analysis technique that detects which part of a Java code base is affected by a change of JDK or third-party libraries—focusing on adaptive SCIs. The technique detects changes by analyzing the difference of bytecode before and after the change and then identifies the affected code entities by building a dependency graph and analyzing the propagation path of the changes. Dintzner et al. [31] proposed FEVER, a heuristic-based technique to extract changes in variability models, assets, and mappings. It accepts as input a set of commits and outputs an instance of its change model covering the given commit range. From the initial set of commits, it first analyzes every commit, identifies the changed code entities and the changes, and creates the relationships between the entities and the changes. Finally, it consolidates the change relationship information over time by keeping track of relationships spreading beyond single commits (i.e., an SCI). Matsuda et al. [100] proposed a technique to reorder and regroup changes, based on a commit policy (i.e., fine-grained or coarse-grained). It reorganizes commit histories to better conform to a specified policy regarding specific SCIs. For example, for a fine-grained policy, different types of refactoring changes should be committed separately while for a coarse-grained policy, refactoring should be separate from behavior-changing changes. Dagenais and Robillard [26] proposed AdDoc, which mines code patterns based on a set of rules and templates. Using code changes, AdDoc generates documentation of the SCI and recommends the documentation update to developers. It also reports violations of the patterns as the code and the documentation evolve.

6.3 Dynamic Analysis

Dynamic analysis techniques are generally applied to identify developers' SCIs, which are then used to improve other techniques. The inputs to these techniques include source code changes and sometimes user-provided specifications.

For example, Yi et al. [172] proposed a *change contract language* to formally describe intended behavioral as well as structural changes across program versions (i.e., goals and actions in our taxonomy). The formal semantics of the change contract language are based on the **Java Modeling Language (JML)** [86], and the language focuses on capturing the intended behavioral changes and their semantic effects. Essentially, a change contract specifies how the post-conditions of the same method in two consecutive versions should relate to each other under certain preconditions. The authors also developed tool support for the language, which enables both test generation to witness contract violation, and automated repair of certain tests that are broken due to program changes.

In Figure 8(b), we display an example of a change contract written for the `execute` method of Apache Ant. Lines 4–13 are JML-style annotations with extra keywords, such as "`change_behavior`," "`requires`," and "`when_signaled`." The change contract in Figure 8(b) pragmatically expresses the verbal description given in the bug report in Figure 8(a). On the high level, the change contract captures both the observed symptom (i.e., failing with an error message, "Use of the extension element [...]") and the necessary condition to reproduce the symptom (i.e., "broken on JDK 7 when a `SecurityManager` is set").

Such a contract is useful for dynamically validating code changes against developer's SCIs. To perform contract validation, random test generation techniques are used to first generate a set of relevant tests, which execute the target method and satisfy the precondition given in the contract. Then, the tests are executed on both the old and the updated versions, instrumented with proper checking code. If a violation is found during the test run, then it is reported to the developer as evidence.

Greevy et al. [41] proposed a technique for Smalltalk programs that summarizes changes to determine whether software entities (e.g., classes) that participate in the implementation of a feature become obsolete, whether new entities are added to the implementation of a feature, and whether code is refactored. It takes multiple versions of a system as input and uses dynamic analysis to extract traces by executing a feature for each version. By comparing and analyzing traces of different versions, the technique captures and summarizes the evolution of a system as SCIs. Marot and Wuyts [98] proposed a technique that detects aspect interference in aspect-oriented programs. To use the technique, developers first semantically annotate the advices, called *compositional intentions*, with their intended compositions. Then, during runtime, if a compositional SCI is violated, then an error is triggered with an explanation of the violation, with the goal of giving feedback to the developers that the program execution is in conflict with their SCI.

Bug 51668 - <junitreport> broken on JDK 7 when a SecurityManager is set

Fails with: "Use of the extension element 'redirect' is not allowed when the secure processing feature is set to true."

It turns out to apply to any environment in which there is a system security manager set.

JDK 7's TransformerFactoryImpl constructor introduced:

```
if (System.getSecurityManager() != null) {
    _isSecureMode = true; _isNotSecureProcessing = false;
}
```

which conflicts with <redirect:write>.

(a) A Bugzilla report for software Ant.

```
// file: XMLResultAggregator.jml
package org.apache.tools.ant.taskdefs.optional.junit;
public class XMLResultAggregator extends Task implements XMLConstants {
  /*@ changed_behavior
    @ requires System.getSecurityManager() != null &&
    @   System.getProperty("java.runtime.version").startsWith("1.7") &&
    @   getDestinationFile().exists() == false;
    @ when_signaled (BuildException e) e.getMessage().contains(
    @   "Use of the extension element 'redirect' is not allowed " +
    @   "when the secure processing feature is set to true." );
    @ signals (BuildException e) false;
    @ ensures getDestinationFile ().exists();
    @*/
  public void execute() throws BuildException;
}
```

(b) A change contract corresponding to the bug report in Figure 8a.

Fig. 8. An example for the dynamic analysis technique by Yi et al. [172].

6.4 Emerging Techniques and LLMs

Emerging AI technologies based on **Large Language Models (LLMs)** [174] have significantly impacted numerous domains, including software engineering [57, 96]. Many early attempts have been made to "reinvent" analyses and tools with LLMs for software-engineering tasks, such as code generation [160], test generation [150], fault localization [65], and program repair [169]. LLM-based techniques emerge as multifaceted and versatile tools to better support software developers based on their access to a vast collection of natural language and code data. But, to date, we are not aware of any peer-reviewed publication on identifying SCIs based on LLMs. Therefore, we exclude such techniques from this analysis.

Nevertheless, LLM-based techniques have the potential to simplify existing SCI-based software analysis techniques. For example, for effort estimation, instead of identifying SCIs first [163], new techniques have been proposed to rely on context-aware language models, such as BERT, to estimate efforts required for software maintenance tasks [4]. This type of techniques can potentially provide an end-to-end solution to many SCI-related software analysis tasks. Yet, recent studies [57, 96] also pointed out that LLM-based techniques, in their current state, have limitations, which may hinder their adoption in some software-engineering scenarios. For example, while LLMs are trained on massive amounts of data, their generalizability across different tasks remains a big challenge. When applied on domains that are outside the scope of training, LLM-based techniques may not perform consistently well. Moreover, the lack of interpretability makes the understanding of the decision-making process of LLM-based techniques difficult. Many studies [92, 171] have shown that it is possible to manipulate model decisions with malicious attacks. Therefore, we believe that identifying SCIs still holds merit even with the presence of LLMs, because an orthogonal angle toward developers' intentions may enhance the interpretability and trustworthiness of the LLM-generated results, leading to better adoption of LLM-based techniques.

RO$_4$: Techniques Related to SCIs

To understand how techniques build on SCIs, we analyzed the 50 publications from our sample that used or proposed them. From the analysis, we learned the following:

— *The goals of the techniques are either identifying SCIs or using them as a concept to improve analysis results.*
— *The techniques rely on three underlying concepts: machine learning, static analysis, and dynamic analysis.*
— *Yet, the range of the techniques is surprisingly large, because they are based on different levels of abstractions of SCIs (e.g., "bug fix" versus "keep" versus contracts).*

7 Threats to Validity

Internal Validity. The internal validity is concerned with how we conducted our meta-study and potential biases we may have introduced. Since we had to extract and interpret natural-language descriptions of other authors, it is possible that we misinterpreted their statements. Consequently, we may have extracted data incorrectly, which may have biased our data analysis. To mitigate this problem, we iteratively refined our analysis process and extracted data based on numerous discussions with all authors of this article. Moreover, we cross-checked the extracted data repeatedly, for instance, when reading on details in individual publications to address our research objectives. To categorize our data, we relied on open-card-sorting-like methods and agreement between the authors to mitigate misinterpretations. Based on this process, the agreement within our dataset, and the affirmation of all authors involved, we argue that threats to the internal validity are mitigated.

External Validity. The external validity is concerned with how well we can apply our findings in the broader context, and thus their general reliability. Since our meta-study synthesizes from 122 publications, we argue that we considered a relevant set of publications to ensure the reliability of our results. We combined an automated with a snowballing search to mitigate potential problems with search engines and to improve the completeness of our dateset. As stop criterion, we used saturation, which means that the new publications we found during the snowballing did not provide fundamentally new insights. Still, since the area of software evolution and maintenance is enormous, it is simply not possible to cover all relevant publications. To tackle this problem, we considered only peer-reviewed publications. So, we may have missed some relevant data for our meta-study, but based on the comparison to related reviews and surveys (cf. Section 9) as well as our expertise in the area, we argue that this threat is limited. Additionally, we publish our data in a persistent open-access repository[1] to allow other researchers to validate, replicate, and extend our study.

8 Implications

After summarizing the key insights with respect to each of our research objectives, we now briefly discuss the consequent implications that researchers and practitioners can derive from our meta-study. Specifically, we consider the following five important implications:

SCIs are an underlying concept of software engineering. Our meta-study revealed that many different research directions are concerned with or at least related to SCIs. While some connections are rather loose, we argue that almost all software-engineering research is at some point connecting to changes of a software system, and thus SCIs. More specifically, software changes are what results in a new system and its modifications, which is why we can consider them key within software-engineering research and practice. For this reason, we argue that SCIs are an important concept to connect different research directions and practices, which are often reported disjointedly despite their connections. So, we think that SCIs may serve as a common connector between research areas in software engineering and can guide techniques as well as practices that combine and integrate such areas [80].

Using a unified classification helps comprehend and compare research. During our meta-study, we found it challenging to fully understand all pieces of research and their connections due to the varying terminologies used. We argue that the lack of a unified terminology makes it very challenging for researchers and practitioners to fully grasp relationships between areas. Consequently, we hope that our classification is a helpful means and stepping stone for overcoming this challenge. In particular, we would recommend that researchers concerned with software changes try to use a common classification or explain why this is not possible. By referring to or extending an existing terminology, they can help others understand the context of their work and its connections to other areas.

The practical evidence on the usefulness of SCIs is limited. We see a continuous interest in (documenting) SCIs in research and practice. Despite this interest, there is limited empirical evidence whether knowing SCIs is helpful in practice. Among others, this lack of evidence makes it challenging to identify the right levels of abstraction for documenting SCIs and how to document them. For this reason, we argue that future studies are needed to elicit more evidence and provide an in-depth understanding of the usefulness of SCIs. The evidence and insights collected via our meta-study are a helpful means for this purpose, and already hint at the relevance of SCIs for research and practice.

Documenting the most important properties of SCIs can help developers. While the empirical evidence is scarce, we argue that it helps developers to agree on a common documentation template for their changes—building on the concept of SCIs to document particularly important and more abstract information. In fact, when sketching our vision of using SCIs in software engineering [80], we found that there are templates on GitHub that cover some of our SCI categories, particularly the goal. Combined with the existing evidence and argued practical benefits, we consider it helpful for developers to document SCIs. Of course, our categorization can support designing a template covering the most important aspects, but the developers have to agree on what details are important to them and how to ensure that the documentation is correct as well as maintained.

Developers and researchers can build on various techniques to work with SCIs. We identified techniques for achieving different goals via three underlying approaches. These insights highlight what has been studied in the past and what are potential new research directions. Even more importantly, researchers and practitioners who want to study or adopt SCIs do not have to start from scratch. Instead, they can build on an extensive body of research and technologies on SCIs. For example, practitioners may want to reuse a technique to automatically label changes according to the involved SCIs, which can warn them about potentially violated SCIs.

We hope that these implications help practitioners and researchers reflect on SCIs, and thereby scope new research as well as practical improvements in the future.

9 Related Work

Next, we provide an overview of the related work. For this purpose, we focus on discussing and comparing the literature reviews we identified during our search (cf. Table 2), since these represent the closest research to our own. We provide a synthesized overview of all six reviews in Table 5. As we can see, the overlap of our meta-study with these reviews is rather low. However, this is not very surprising, since the reviews focus on different research goals (e.g., change-impact analysis,

Table 5. Comparison of the Literature Reviews from Table 2 to Our Meta-study (122 Publications)

authors	period	publications covered			research goal	commonalities	differences
		#	# overlap	% we cover			
Benestad et al. [13]	1993–2007	34	15	44.12	identify measurement goals and attributes used in change-based studies	some attributes partly overlap with SCIs	attributes also span data and metrics; no analysis of SCIs, evidence, or techniques
Jamshidi et al. [62]	1995–2011	60	0	0.00	identify types of architecture evolution with respective formalisms, reasonings, run-time aspects, and tools	types of evolution partly overlap with SCIs	focus on architecture changes and automation; no analysis of evidence, goals, or benefits of SCIs
Kagdi et al. [64]	1996–2006	80	4	5.00	identify the repositories used in, purpose of, methodologies for, and evaluations used in mining papers	software changes that partly overlap with SCIs as a subset of the proposed taxonomy	no deeper analysis of software changes; no analysis of techniques, evidence, or benefits of knowing SCIs
Lehnert [88]	1991–2011	160	N/A	N/A	identify techniques for change-impact analysis classifying the artifacts analyzed, inputs used, changes supported, and algorithms	classification of software changes is a subset of the category *actions* in our work	no abstraction of SCIs or analysis of their benefits, evidence, and techniques
Ruiz et al. [130]	1998–2002	13	1	7.69	formalize an ontology of software maintenance	some SCIs are partly represented in the ontology	no analysis of SCIs
Williams and Carver [166]	1976–2008	130	14	10.77	identify attributes of software change taxonomies as part of studying architecture evolution and impact analysis	subsets of SCIs partly included in change taxonomies	no analysis of SCIs

N/A: paper states 160 included publications, but the online list linked in the paper is not available anymore.
The overlap specifies how many papers of each literature review we cover, too.

architecture evolution) that inherently lead to other papers being relevant. We also noticed that we often included publications by the same authors in our meta-study in which these focused more on SCIs, such as the journal extension of Germán [36] whose original conference paper was included in the review by Kagdi et al. [64]. Also, our literature search covered over 10 more recent years, which logically leads to many more recent publications in our study. Note that we assume that we cover the most relevant data from the reviews' primary studies, because we analyzed the reviews themselves. Last, the distribution of venues (e.g., most publications being published at the International Conference on Software Maintenance and Evolution) between our meta-study aligns very well with the reviews, like the one by Williams and Carver [166]. All of this improves our confidence in our selection of publications, even though none of the previous reviews has attempted to address our research objectives.

Benestad et al. [13] report a literature review on change-based studies. Their goal is to summarize the state of the art and identify future challenges for researchers. For this purpose, the authors summarized the goals of the included studies and identified 43 change attributes that they mapped into a conceptual model. Since their goals and extracted data partly overlap with ours, this literature review is arguably the closest one to our own meta-study. Unfortunately, the literature review of Benestad et al. has the same limitations of unclear categories and missing evidence we described as a limitation for understanding SCIs in the beginning of this article. Concretely, while Benestad et al. elicit change attributes, they also intermix categories. For instance, they define attributes, such as activity (partially mapping to goals in our taxonomy), maintenance type (again, partially mapping to goals in our taxonomy), change size (partially mapping to actions in our taxonomy), change interval, code quality, developer ID, status, or tool use. These attributes can be used to describe a change, but there is no common connection between them (e.g., they are related to SCIs, statistical metrics, meta-data), and the benefits of knowing them are unclear. We improved considerably on this work by (1) focusing on a concrete and related set of change attributes (i.e., those related to SCIs instead of statistics of changes), (2) synthesizing these SCIs into distinct categories to clarify their scope, (3) discussing the actual empirical evidence on the use of knowing SCIs (i.e., not only their goals), (4) analyzing the techniques used in the context of SCIs (not covered by Benestad et al.), and (5) considering more recent work (i.e., more than a decade of new research).

Consequently, our contributions and insights extend and complement those of Benestad et al., providing a more focused and recent understanding of SCIs for researchers and practitioners.

The other literature reviews and surveys focus on a specific type of change or certain techniques, which are partially related to or can involve SCIs. For this reason, we included them into our analyses, while they have only few relations to our own meta-study. Concretely, Jamshidi et al. [62] studied 60 publications to provide a taxonomy for classifying architecture-centric software evolution research. Their sub-categories "need for evolution" (e.g., corrective) and "means of evolution" (e.g., refactoring) intermix different SCIs that relate to goals in our taxonomy—whereas their other categories are not fitting for SCIs (e.g., "UML specification"). The remainder of the review focuses on formalizing, reasoning about, and tools for architectural changes, constructing a framework that specifies the relations between these points, requirements, system models, and the system's execution. We have not been concerned with these points in our meta-study. For instance, we actively decided to include models only if they have been directly connected to code changes (cf. Section 2.2).

Kagdi et al. [64] surveyed 80 publications to evaluate how well their proposed taxonomy helps classify research on mining software repositories. Software changes are only a subset in this taxonomy and do match only to the category *actions* in our categorization of SCIs. The remaining review focuses on aspects, such as the repositories covered, mining methodologies and their evaluations, or goals of the studies. None of these have been objectives of our meta-study. Similarly, Lehnert [88] reviewed 160 publications to provide a taxonomy for describing research on change-impact analysis. For this purpose, Lehnert summarized properties like the techniques proposed, their required input, or algorithms used. As we found while comparing this review to our meta-study, change-impact analysis rarely considers SCIs, but focuses primarily on identifying whether one arbitrary change causes other changes. Consequently, some categories in the taxonomy fit the category *actions* of our categorization, but deeper insights or actual SCIs are missing from the review by Lehnert. Ruiz et al. [130] proposed an ontology of software maintenance by surveying 13 publications and industry standards. However, this ontology focuses on a high-level conceptual model of software maintenance (involving, e.g., "resource," "process management," and "support"), and thus scratches only a marginal part of SCIs. Also, Ruiz et al. did not aim to provide an overview of the research area, but picked a known selection of publications that were feasible for their own work. Last, Williams and Carver [166] reviewed 130 publications to provide a taxonomy for characterizing architectural changes, with similar differences to our work as the review by Jamshidi et al. Specifically, Williams and Carver did not analyze SCIs beyond eliciting a subset of categories that partly overlap with our categorization. In contrast to all these works, we focus on actual software changes, the SCIs behind them, and other research objectives (e.g., empirical evidence) and provide a more recent overview of the research area.

10 Conclusion

In this article, we have reported the results of an extensive meta-study in which we analyzed 122 publications related to SCIs. We elicited these publications based on the methodology of systematic literature reviews [73], combining an automated and a snowballing search. Using open-coding-like, open-card-sorting-like, and axial-coding methods, we extracted and analyzed data from these publications to (**RO$_1$**) capture the research on SCIs, (**RO$_2$**) derive a classification of SCIs, (**RO$_3$**), collect empirical evidence on the benefits of knowing SCIs, and (**RO$_4$**) comparing techniques related to SCIs. Our key contributions with respect to these research objectives are as follows:

RO$_1$ We found that most publications on SCIs contribute techniques (50) and empirical studies (48), employing SCIs in a wide range of contexts (e.g., predicting maintenance activities, verifying changes) for various goals (e.g., improving techniques, monitoring projects) and often define or adapt taxonomies non-systematically (i.e., as needed).

- **RO$_2$** We propose a classification that provides a concise overview of SCIs by defining five orthogonal categories (i.e., goals, actions, objects, customer, lifecycle phase) and including concrete examples from existing publications—serving as a common ground for communicating, understanding, and extending research on SCIs.
- **RO$_3$** We identified that knowing SCIs can serve several benefits (e.g., comparing research), but eliciting reliable empirical evidence on these benefits for practitioners is challenging, and comparing this evidence between publications is hampered by the different understandings that are established and that we aimed to align with our classification.
- **RO$_4$** We provide an overview of techniques related to SCIs, which indicates that these techniques either aim to identify SCIs or use them to improve an established analysis, build on three underlying technologies (i.e., machine learning, static analysis, dynamic analysis), and span a variety of abstractions of SCIs—with our classification and overview serving as a common ground to understand and extend such techniques.

Our insights underpin the value of having a concise classification of SCIs that enables researchers and practitioners to understand the existing body of knowledge of software evolution research. Particularly, our overview of the existing publications highlights the potential knowing SCIs could have, and the need for eliciting empirical evidence that this is the case. Such empirical evidence is a consequent next step for future work, in which we also want to explore how to compare and integrate different techniques to facilitate research on software evolution and SCIs. Our insights into the goals, benefits, and evidence of using SCIs have motivated us to define a research agenda on using SCIs to move toward controlled software evolution that can help developers avoid errors and improve the comprehensibility of software changes [80].

References

[1] Alain Abran and Hong Nguyenkim. 1991. Analysis of maintenance work categories through measurement. In *ICSM*. IEEE.

[2] Alain Abran and Hong Nguyenkim. 1993. Measurement of the maintenance process from a demand-based perspective. *J. Softw. Maint. Res. Pract.* 5, 2 (1993), 63–90.

[3] Jonas Åkesson, Sebastian Nilsson, Jacob Krüger, and Thorsten Berger. 2019. Migrating the android apo-games into an annotation-based software product line. In *SPLC*. ACM.

[4] Mohammed Alhamed and Tim Storer. 2022. Evaluation of context-aware language models and experts for effort estimation of software maintenance issues. In *ICSME*. IEEE.

[5] Eman A. Al Omar, Jiaqian Liu, Kenneth Addo, Mohamed W. Mkaouer, Christian Newman, Ali Ouni, and Zhe Yu. 2022. On the documentation of refactoring types. *Autom. Softw. Eng.* 29, 9 (2022), 1–40.

[6] Robert S. Arnold and Donald A. Parker. 1982. The dimensions of healthy maintenance. In *ICSE*. IEEE.

[7] Wesley K. G. Assunção, Jacob Krüger, Sébastien Mosser, and Sofiane Selaoui. 2023. How Do microservices evolve? An empirical analysis of changes in open-source microservice repositories. *J. Syst. Softw.* 204, 111788 (2023), 1–14.

[8] Muhammad A. Babar and He Zhang. 2009. Systematic literature reviews in software engineering: Preliminary results from interviews with researchers. In *ESEM*. IEEE.

[9] Thar Baker, Michael Mackay, Martin Randles, and Azzelarabe Taleb-Bendiab. 2013. Intention-oriented programming support for runtime adaptive autonomic cloud-based applications. *Comput. Electr. Eng.* 39, 7 (2013), 2400–2412.

[10] Victor Basili, Lionel Briand, Steven Condon, Yong-Mi Kim, Walcélio L. Melo, and Jon D. Valen. 1996. Understanding and predicting the process of software maintenance releases. In *ICSE*. IEEE.

[11] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. 2012. When does a refactoring induce bugs? An empirical study. In *SCAM*. IEEE.

[12] Robert M. Bell, Thomas J. Ostrand, and Elaine J. Weyuker. 2011. Does measuring code change improve fault prediction? In *PROMISE*. ACM.

[13] Hans Christian Benestad, Bente Anda, and Erik Arisholm. 2009. Understanding software maintenance and evolution by analyzing individual changes: A literature review. *J. Softw. Maint. Res. Pract.* 21, 6 (2009), 349–378.

[14] Susan Bergin and John G. Keating. 2003. A case study on the adaptive maintenance of an internet application. *J. Softw. Maint. Res. Pract.* 15, 4 (2003), 254–264.

[15] Ted Biggerstaff, Bharat Mitbander, and Dallas Webster. 1993. The concept assignment problem in program understanding. In *WCRE*. IEEE.

[16] Andrew P. Bradley. 1997. The use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern Recogn.* 30, 7 (1997), 1145–1159.
[17] Lionel C. Briand and Victor R. Basili. 1992. A classification procedure for the effective management of changes during the maintenance process. In *ICSM*. IEEE.
[18] Lionel C. Briand, Victor R. Basili, Yong-Mi Kim, and Donald R. Squier. 1994. A change analysis process to characterize software maintenance projects. In *ICSM*. IEEE.
[19] Lionel C. Briand, Yong-Mi Kim, Walcélio L. Melo, Carolyn B. Seaman, and Victor R. Basili. 1998. Q-MOPP: Qualitative evaluation of maintenance organizations, processes and products. *J. Softw. Maint. Res. Pract.* 10, 4 (1998), 249–278.
[20] Aline Brito, André C. Hora, and Marco Tulio Valente. 2020. Refactoring graphs: Assessing refactoring over time. In *SANER*. IEEE.
[21] Panuchart Bunyakiati and Chadarat Phipathananunth. 2017. Cherry-picking of code commits in long-running, multi-release software. In *FSE*. ACM.
[22] Elizabeth Burd and Malcolm Munro. 1999. An initial approach towards measuring and characterising software evolution. In *WCRE*. IEEE.
[23] Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo da Silva Sousa, Rafael Maiani de Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. 2017. Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects. In *ESEC/FSE*. ACM.
[24] Ned Chapin, Joanne E. Hale, Khaled Md. Khan, Juan F. Ramil, and Wui-Gee Tan. 2001. Types of software evolution and software maintenance. *J. Softw. Maint. Evol. Res. Pract.* 13, 1 (2001), 3–30.
[25] Luis Cortés-Coy, Mario Linares-Vásquez, Jairo Aponte, and Denys Poshyvanyk. 2014. On automatically generating commit messages via summarization of source code changes. In *SCAM*. IEEE.
[26] Barthélémy Dagenais and Martin P. Robillard. 2014. Using traceability links to recommend adaptive changes for documentation evolution. *IEEE Trans. Softw. Eng.* 40, 11 (2014), 1126–1146.
[27] Jamel Debbiche, Oskar Lignell, Jacob Krüger, and Thorsten Berger. 2019. Migrating Java-based apo-games into a composition-based software product line. In *SPLC*. ACM.
[28] Tejinder Dhaliwal, Foutse Khomh, Ying Zou, and Ahmed E. Hassan. 2012. Recovering commit dependencies for selective code integration in software product lines. In *ICSM*. IEEE.
[29] Márcio Greyck Batista Dias, Nicolas Anquetil, and Káthia Marçal de Oliveira. 2003. Organizing the knowledge used in software maintenance. *J. Univers. Comput. Sci.* 9, 7 (2003), 641–658.
[30] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. 2016. FEVER: Extracting feature-oriented changes from commits. In *MSR*. ACM.
[31] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. 2018. FEVER: An approach to analyze feature-oriented changes and artefact co-evolution in highly configurable systems. *Empir. Softw. Eng.* 23, 2 (2018), 905–952.
[32] Beat Fluri and Harald C. Gall. 2006. Classifying change types for qualifying change couplings. In *ICPC*. IEEE.
[33] Ying Fu, Meng Yan, Xiaohong Zhang, Ling Xu, Dan Yang, and Jeffrey D. Kymer. 2015. Automated classification of software change messages by semi-supervised latent dirichlet allocation. *Inf. Softw. Technol.* 57 (2015), 369–377.
[34] Harald C. Gall, Beat Fluri, and Martin Pinzger. 2009. Change analysis with evolizer and ChangeDistiller. *IEEE Softw.* 26, 6 (2009), 26–33.
[35] David Gefen and Scott L. Schneberger. 1996. The non-homogeneous maintenance periods: A case study of software modifications. In *ICSM*. IEEE.
[36] Daniel M. Germán. 2006. An empirical study of fine-grained software modifications. *Empir. Softw. Eng.* 11, 3 (2006), 369–393.
[37] Lobna Ghadhab, Ilyes Jenhani, Mohamed W. Mkaouer, and Montassar Ben Messaoud. 2021. Augmenting commit classification by using fine-grained source code changes and a pre-trained deep neural language model. *Inf. Softw. Technol.* 135, 106566 (2021), 1–13.
[38] Emanuel Giger, Martin Pinzger, and Harald C. Gall. 2011. Comparing fine-grained source code changes and code churn for bug prediction. In *MSR*. ACM.
[39] Carsten Görg and Peter Weißgerber. 2005. Detecting and visualizing refactorings from software archives. In *IWPC*. IEEE.
[40] Georgios Gousios. 2013. The GHTorent dataset and tool suite. In *MSR*. IEEE.
[41] Orla Greevy, Stéphane Ducasse, and Tudor Gîrba. 2005. Analyzing feature traces to incorporate the semantics of change in software evolution analysis. In *ICSM*. IEEE.
[42] Anita Gupta, Reidar Conradi, Forrest Shull, Daniela S. Cruzes, Christopher Ackermann, Harald Rønneberg, and Einar Landre. 2008. Experience report on the effect of software development characteristics on change distribution. In *PROFES*. Springer.
[43] Jens Gustavsson. 2003. A classification of unanticipated runtime software changes in Java. In *ICSM*. IEEE.

[44] Masatomo Hashimoto and Akira Mori. 2012. Enhancing history-based concern mining with fine-grained change analysis. In *CSMR*. IEEE.

[45] Ahmed E. Hassan. 2008. Automated classification of change messages in open source projects. In *SAC*. ACM.

[46] Ahmed E. Hassan. 2009. Predicting faults using the complexity of code changes. In *ICSE*. IEEE.

[47] Lile P. Hattori and Michele Lanza. 2008. On the nature of commits. In *ASE-W*. IEEE.

[48] Shinpei Hayashi, Takayuki Omori, Teruyoshi Zenmyo, Katsuhisa Maruyama, and Motoshi Saeki. 2012. Refactoring edit history of source code. In *ICSM*. IEEE.

[49] Wolfgang Heider, Michael Vierhauser, Daniela Lettner, and Paul Grünbacher. 2012. A case study on the evolution of a component-based product line. In *WICSA/ECSA*. IEEE.

[50] Tjaša Heričko, Saša Brdnik, and Boštjan Šumak. 2022. Commit classification into maintenance activities using aggregated semantic word embeddings of software change messages. In *SQAMIA*. CEUR-WS.org.

[51] Kim Herzig, Sascha Just, and Andreas Zeller. 2013. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *ICSE*. IEEE.

[52] Kim Herzig and Andreas Zeller. 2013. The impact of tangled code changes. In *MSR*. IEEE.

[53] Abram Hindle, Daniel M. German, Michael W. Godfrey, and Richard C. Holt. 2009. Automatic classification of large changes into maintenance categories. In *ICPC*. IEEE.

[54] Abram Hindle, Daniel M. Germán, and Richard C. Holt. 2008. What do large commits tell us? A taxonomical study of large commits. In *MSR*. ACM.

[55] Sebastian Hönel, Morgan Ericsson, Welf Löwe, and Anna Wingkvist. 2020. Using source code density to improve the accuracy of automatic commit classification into maintenance activities. *J. Syst. Softw.* 168, 110673 (2020), 1–19.

[56] Daqing Hou and H. James Hoover. 2006. Using SCL to specify and check design intent in source code. *IEEE Trans. Softw. Eng.* 32, 6 (2006), 404–423.

[57] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *arXiv preprint* arXiv:2308.10620 (2024).

[58] Yuan Huang, Qiaoyang Zheng, Xiangping Chen, Yingfei Xiong, Zhiyong Liu, and Xiaonan Luo. 2017. Mining version control system for automatically generating commit comment. In *ESEM*. IEEE.

[59] Marieke Huisman, Herbert Bos, Sjaak Brinkkemper, Arie van Deursen, Jan Groote, Patricia Lago, Jaco van de Pol, and Eelco Visser. 2016. Software that meets its intent. In *ISoLA*. Springer.

[60] Ayelet Israeli and Dror G. Feitelson. 2009. *Characterizing Software Maintenance Categories Using the Linux Kernel.* Technical Report 2009–10. The Hebrew University of Jerusalem.

[61] Daniel Jackson and David A. Ladd. 1994. Semantic diff: A tool for summarizing the effects of modifications. In *ICSM*. IEEE.

[62] Pooyan Jamshidi, Mohammad Ghafari, Aakash Ahmad, and Claus Pahl. 2013. A framework for classifying and comparing architecture-centric software evolution research. In *CSMR*. IEEE.

[63] Magne Jørgensen. 1995. Experience with the accuracy of software maintenance task effort prediction models. *IEEE Trans. Softw. Eng.* 21, 8 (1995), 674–681.

[64] Huzefa H. Kagdi, Michael L. Collard, and Jonathan I. Maletic. 2007. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Res. Pract.* 19, 2 (2007), 77–131.

[65] Sungmin Kang, Gabin An, and Shin Yoo. 2023. A preliminary evaluation of LLM-based fault localization. *arXiv preprint* arXiv:2308.05487 (2023).

[66] David Kawrykow and Martin P. Robillard. 2011. Non-essential changes in version histories. In *ICSE*. ACM.

[67] Chris F. Kemerer and Sandra Slaughter. 1997. Determinants of software maintenance profiles: An empirical investigation. *J. Softw. Maint. Res. Pract.* 9, 4 (1997), 235–251.

[68] Chris F. Kemerer and Sandra Slaughter. 1999. An empirical approach to studying software evolution. *IEEE Trans. Softw. Eng.* 25, 4 (1999), 493–509.

[69] Miryung Kim, Dongxiang Cai, and Sunghun Kim. 2011. An empirical investigation into the role of api-level refactorings during software evolution. In *ICSE*. ACM.

[70] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2014. An empirical study of refactoring: Challenges and benefits at microsoft. *IEEE Trans. Softw. Eng.* 40, 7 (2014), 633–649.

[71] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. 2007. Predicting faults from cached history. In *ICSE*. IEEE.

[72] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. 2014. Hey! are you committing tangled changes? In *ICPC*. ACM.

[73] Barbara A. Kitchenham, David Budgen, and O. Pearl Brereton. 2015. *Evidence-Based Software Engineering and Systematic Reviews.* CRC.

[74] Barbara A. Kitchenham, Guilherme H. Travassos, Anneliese von Mayrhauser, Frank Niessink, Norman F. Schneidewind, Janice Singer, Shingo Takada, Risto Vehvilainen, and Hongji Yang. 1999. Towards an ontology of software maintenance. *J. Softw. Maint. Res. Pract.* 11, 6 (1999), 365–389.

[75] Amy Ko, Robert DeLine, and Gina Venolia. 2007. Information needs in collocated software development teams. In *ICSE*. IEEE.

[76] Jacob Krüger. 2021. *Understanding the Re-Engineering of Variant-Rich Systems: An Empirical Work on Economics, Knowledge, Traceability, and Practices*. Ph.D. Dissertation. Otto-von-Guericke University Magdeburg.

[77] Jacob Krüger and Regina Hebig. 2020. What developers (care to) recall: An interview survey on smaller systems. In *ICSME*. IEEE.

[78] Jacob Krüger and Regina Hebig. 2023. To memorize or to document: A survey of developers' views on knowledge availability. In *PROFES*. Springer.

[79] Jacob Krüger, Christian Lausberger, Ivonne von Nostitz-Wallwitz, Gunter Saake, and Thomas Leich. 2020. Search. review. repeat? An empirical study of threats to replicating SLR searches. *Empir. Softw. Eng.* 25, 1 (2020), 627–677.

[80] Jacob Krüger, Yi Li, Chenguang Zhu, Marsha Chechik, Thorsten Berger, and Julia Rubin. 2023. A vision on intentions in software engineering. In *ESEC/FSE*. ACM.

[81] Jacob Krüger, Wardah Mahmood, and Thorsten Berger. 2020. Promote-pl: A round-trip engineering process model for adopting and evolving product lines. In *SPLC*. ACM.

[82] Jacob Krüger, Alex Mikulinski, Sandro Schulze, Thomas Leich, and Gunter Saake. 2023. DSDGen: Extracting documentation to comprehend fork merges. In *SPLC*. ACM.

[83] Elias Kuiter, Jacob Krüger, Sebastian Krieter, Thomas Leich, and Gunter Saake. 2018. Getting rid of clone-and-own: Moving to a software product line for temperature monitoring. In *SPLC*. ACM.

[84] David Chenho Kung, Jerry Gao, Pei Hsia, F. Wen, Yasufumi Toyoshima, and Cris Chen. 1994. Change impact identification in object oriented software maintenance. In *ICSM*.

[85] Thomas D. LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining mental models: A study of developer work habits. In *ICSE*. ACM.

[86] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. 2006. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Softw. Eng. Notes* 31, 3 (2006), 1–38.

[87] Min-Gu Lee and Theresa L. Jefferson. 2005. An empirical study of software maintenance of a web-based java application. In *ICSM*. IEEE.

[88] Steffen Lehnert. 2011. A taxonomy for software change impact analysis. In *EVOL/IWPSE*. ACM.

[89] Steffen Lehnert, Qurat-ul-ann Farooq, and Matthias Riebisch. 2012. A taxonomy of change types and its application in software evolution. In *ECBS*. IEEE.

[90] Stanislav Levin and Amiram Yehudai. 2016. Using temporal and semantic developer-level information to predict maintenance activity profiles. In *ICSME*. IEEE.

[91] Stanislav Levin and Amiram Yehudai. 2017. Boosting automatic commit classification into maintenance activities by utilizing source code changes. In *PROMISE*. ACM.

[92] Jia Li, Zhuo Li, Huangzhao Zhang, Ge Li, Zhi Jin, Xing Hu, and Xin Xia. 2022. Poison attack and defense on deep source code processing models. *arXiv preprint* arXiv:2210.17029 (2022).

[93] Bennet P. Lientz, E. Burton Swanson, and G. E. Tompkins. 1978. Characteristics of applications software maintenance. *Commun. ACM* 21, 6 (1978), 466–471.

[94] Max Lillack, Ştefan Stănciulescu, Wilhelm Hedman, Thorsten Berger, and Andrzej Wąsowski. 2019. Intention-based integration of software variants. In *ICSE*. IEEE.

[95] Ie-Hong Lin and David A. Gustafson. 1988. Classifying software maintenance. In *ICSM*. IEEE.

[96] David Lo. 2023. Trustworthy and synergistic artificial intelligence for software engineering: Vision and roadmaps. *arXiv preprint* arXiv:2309.04142 (2023).

[97] Walid Maalej. 2010. *Intention-Based Integration of Software Engineering Tools*. Ph.D. Dissertation. Technical University of Munich.

[98] Antoine Marot and Roel Wuyts. 2009. Detecting unanticipated aspect interferences at runtime with compositional intentions. In *RAM-SE*. ACM.

[99] Matias Martinez, Laurence Duchien, and Martin Monperrus. 2013. Automatically extracting instances of code change patterns with AST analysis. In *ICSM*. IEEE.

[100] Jumpei Matsuda, Shinpei Hayashi, and Motoshi Saeki. 2015. Hierarchical categorization of edit operations for separately committing large refactoring results. In *IWPSE*. ACM.

[101] Andreas Mauczka, Florian Brosch, Christian Schanes, and Thomas Grechenig. 2015. Dataset of developer-labeled commit messages. In *MSR*. IEEE.

[102] Andreas Mauczka, Markus Huber, Christian Schanes, Wolfgang Schramm, Mario Bernhart, and Thomas Grechenig. 2012. Tracing your maintenance work—A cross-project validation of an automated classification dictionary for commit messages. In *FASE*. Springer.

[103] Kim Mens, Tom Mens, and Michel Wermelinger. 2002. Supporting software evolution with intentional software views. In *IWPSE*. IEEE.

[104] Kim Mens, Bernard Poll, and Sebastián González. 2003. Using intentional source-code views to aid software maintenance. In *ICSM*. IEEE.

[105] Tom Mens, Jim Buckley, Matthias Zenger, and Awais Rashid. 2003. Towards a taxonomy of software evolution. In *USE*. EPFL.

[106] Omar Meqdadi, Nouh Alhindawi, Michael L. Collard, and Jonathan I. Maletic. 2013. Towards understanding large-scale adaptive changes from version histories. In *ICSM*. IEEE.

[107] Audris Mockus and Lawrence G. Votta. 2000. Identifying reasons for software changes using historic databases. In *ICSM*. IEEE.

[108] Audris Mockus and David M. Weiss. 2000. Predicting risk of software changes. *Bell Labs Tech. J.* 5, 2 (2000), 169–180.

[109] Parastoo Mohagheghi and Reidar Conradi. 2004. An empirical study of software change: Origin, acceptance rate, and functionality vs. quality attributes. In *ISESE*. IEEE.

[110] Laís Neves, Paulo Borba, Vander Alves, Lucinéia Turnes, Leopoldo Teixeira, Demóstenes Sena, and Uirá Kulesza. 2015. Safe evolution templates for software product lines. *J. Syst. Softw.* 106 (2015), 42–58.

[111] Laís Neves, Leopoldo Teixeira, Demóstenes Sena, Vander Alves, Uirá Kulesza, and Paulo Borba. 2011. Investigating the safe evolution of software product lines. In *GPCE*. ACM.

[112] Sebastian Nielebock, Paul Blockhaus, Jacob Krüger, and Frank Ortmeier. 2021. AndroidCompass: A dataset of android compatibility checks in code repositories. In *MSR*. IEEE.

[113] John T. Nosek and Prashant Palvia. 1990. Software maintenance management: Changes in the last decade. *J. Softw. Maint. Res. Pract.* 2, 3 (1990), 157–174.

[114] Matheus Paixão, Jens Krinke, DongGyun Han, Chaiyong Ragkhitwetsagul, and Mark Harman. 2017. Are developers aware of the architectural impact of their changes? In *ASE*. IEEE.

[115] Matheus Paixão, Anderson G. Uchôa, Ana Carla Bibiano, Daniel Oliveira, Alessandro Garcia, Jens Krinke, and Emilio Arvonio. 2020. Behind the intents: An in-depth empirical study on software refactoring in modern code review. In *MSR*. ACM.

[116] Fabio Palomba, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2017. An exploratory study on the relationship between changes and refactoring. In *ICPC*. IEEE.

[117] Kai Pan, Sunghun Kim, and E. James Whitehead Jr. 2009. Toward an understanding of bug fix patterns. *Empir. Softw. Eng.* 14, 3 (2009), 286–315.

[118] Leonardo Teixeira Passos and Krzysztof Czarnecki. 2014. A dataset of feature additions and feature removals from the linux kernel. In *MSR*. ACM.

[119] Antoine Pietri, Diomidis Spinellis, and Stefano Zacchiroli. 2020. The software heritage graph dataset: Large-scale analysis of public software development history. In *MSR*. ACM.

[120] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering*. Springer.

[121] Macario Polo, Mario Piattini, and Francisco Ruiz. 2001. Using code metrics to predict maintenance of legacy programs: A case study. In *ICSM*. IEEE.

[122] Daryl Posnett, Abram Hindle, and Premkumar T. Devanbu. 2011. Got issues? Do new features and code improvements affect defects? In *WCRE*. IEEE.

[123] Ranjith Purushothaman and Dewayne E. Perry. 2005. Toward understanding the rhetoric of small source code changes. *IEEE Trans. Softw. Eng.* 31, 6 (2005), 511–526.

[124] Dawei Qi, Jooyong Yi, and Abhik Roychoudhury. 2012. Software change contracts. In *FSE*. ACM.

[125] Derek Rayside, Scott Kerr, and Kostas Kontogiannis. 1998. Change and adaptive maintenance detection in Java software systems. In *WCRE*. IEEE.

[126] Ralf Reussner, Michael Goedicke, Wilhelm Hasselbring, Birgit Vogel-Heuser, Jan Keim, and Lukas Märtin (Eds.). 2019. *Managed Software Evolution*. Springer.

[127] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. 2012. How do professional developers comprehend software? In *ICSE*. IEEE.

[128] H. Dieter Rombach, Bradford T. Ulery, and Jon D. Valett. 1992. Toward full life cycle control: Adding maintenance measurement to the SEL. *J. Syst. Softw.* 18, 2 (1992), 125–138.

[129] Alexander J. Rostkowycz, Václav Rajlich, and Andrian Marcus. 2004. A case study on the long-term effects of software redocumentation. In *ICSM*. IEEE.

[130] Francisco Ruiz, Aurora Vizcaíno Barceló, Mario Piattini, and Félix García. 2004. An ontology for the management of software maintenance projects. *Int. J. Softw. Eng. Knowl. Eng.* 14, 3 (2004), 323–349.

[131] Samar Saeed, Shahrzad Sheikholeslami, Jacob Krüger, and Regina Hebig. 2023. What data scientists (care to) recall. In *PROFES*. Springer.

[132] Munish Saini and Kuljit Kaur Chahal. 2018. Change profile analysis of open-source software systems to understand their evolutionary behavior. *Front. Comput. Sci.* 12, 12 (2018), 1105–1124.

[133] Gabriela Sampaio, Paulo Borba, and Leopoldo Teixeira. 2016. Partially safe evolution of software product lines. In *SPLC*. ACM.

[134] Stephen R. Schach, Bo Jin, Liguo Yu, Gillian Z. Heller, and A. Jefferson Offutt. 2003. Determining the distribution of maintenance categories: Survey versus measurement. *Empir. Softw. Eng.* 8, 4 (2003), 351–365.

[135] Yusra Shakeel, Jacob Krüger, Ivonne von Nostitz-Wallwitz, Christian Lausberger, Gabriel C. Durand, Gunter Saake, and Thomas Leich. 2018. (Automated) literature analysis - threats and experiences. In *SE4Science*. ACM.

[136] Yaqian Shen and Nazim H. Madhavji. 2006. ESDM—A method for developing evolutionary scenarios for analysing the impact of historical changes on architectural elements. In *ICSM*. IEEE.

[137] Luciana Lourdes Silva, Klérisson Vinícius Ribeiro Paixão, Sandra de Amo, and Marcelo de Almeida Maia. 2011. On the use of execution trace alignment for driving perfective changes. In *CSMR*. IEEE.

[138] Charles Simonyi. 1995. *The Death of Computer Languages, the Birth of Intentional Programming*. Technical Report MSR-TR-95-52. Microsoft Research.

[139] Charles Simonyi, Magnus Christerson, and Shane Clifford. 2006. Intentional software. In *OOPSLA*. ACM.

[140] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? In *MSR*. ACM.

[141] Sarocha Sothornprapakorn, Shinpei Hayashi, and Motoshi Saeki. 2018. Visualizing a tangled change for supporting its decomposition and commit construction. In *COMPSAC*. IEEE.

[142] Maria J. C. Sousa and Helena M. Moreira. 1998. A survey on the software maintenance process. In *ICSM*. IEEE.

[143] Klaas-Jan Stol and Brian Fitzgerald. 2018. The ABC of software engineering research. *ACM Trans. Softw. Eng. Methodol.* 27, 3 (2018), 11:1–51.

[144] Maximilian Störzer, Barbara G. Ryder, Xiaoxia Ren, and Frank Tip. 2006. Finding failure-inducing changes in Java programs using change classification. In *FSE*. ACM.

[145] Anselm Strauss and Juliet Corbin. 1998. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Sage.

[146] Xiaobing Sun, Bixin Li, Chuanqi Tao, Wanzhi Wen, and Sai Zhang. 2010. Change impact analysis based on a taxonomy of change types. In *COMPSAC*. IEEE.

[147] Xiaobing Sun, Bixin Li, Wanzhi Wen, and Sai Zhang. 2013. Analyzing impact rules of different change types to support change impact analysis. *Int. J. Softw. Eng. Knowl. Eng.* 23, 3 (2013), 259–288.

[148] Burton Swanson. 1976. The dimensions of maintenance. In *ICSE*. ACM.

[149] Xinye Tang, Song Wang, and Ke Mao. 2015. Will this bug-fixing change break regression testing? In *ESEM*. IEEE.

[150] Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. 2023. ChatGPT vs SBST: A comparative assessment of unit test suite generation. *IEEE Trans. Softw. Eng.* Early Access (2023), 1–19.

[151] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. 2012. How do software engineers understand code changes? An exploratory study in industry. In *FSE*. ACM.

[152] Yida Tao and Sunghun Kim. 2015. Partitioning composite code changes to facilitate code review. In *MSR*. IEEE.

[153] Sirinut Thangthumachit, Shinpei Hayashi, and Motoshi Saeki. 2011. Understanding source code differences by separating refactoring effects. In *APSEC*. IEEE.

[154] Yuan Tian, Julia Lawall, and David Lo. 2012. Identifying linux bug fixing patches. In *ICSE*. IEEE.

[155] Pitamber Tiwari, Wei Li, Raouf Alomainy, and Bingyang Wei. 2013. An empirical study of different types of changes in the eclipse project. *Open Softw. Eng. J.* 7 (2013).

[156] Alexander Trautsch, Johannes Erbel, Steffen Herbold, and Jens Grabowski. 2023. What really changes when developers intend to improve their source code: A commit-level study of static metric value and static analysis warning changes. *Empir. Softw. Eng.* 28, 30 (2023), 1–40.

[157] Martin Treiber, Hong Linh Truong, and Schahram Dustdar. 2008. On analyzing evolutionary changes of web services. In *ICSOC*. Springer.

[158] Nikolaos Tsantalis, Victor Guana, Eleni Stroulia, and Abram Hindle. 2013. A multidimensional empirical study on refactoring activity. In *CASCON*. IBM.

[159] Nikolaos Tsantalis, Matin Mansouri, Laleh Mousavi Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In *ICSE*. ACM.

[160] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *CHI*. ACM.

[161] Birgit Vogel-Heuser, Thomas Simon, Jens Folmer, Robert Heinrich, Kiana Rostami, and Ralf H. Reussner. 2016. Towards a common classification of changes for information and automated production systems as precondition for maintenance effort estimation. In *INDIN*. IEEE.

[162] Anneliese von Mayrhauser and A. Marie Vans. 1995. Program comprehension during software maintenance and evolution. *Computer* 28, 8 (1995), 44–55.

[163] Song Wang, Chetan Bansal, and Nachiappan Nagappan. 2021. Large-scale intent analysis for identifying large-review-effort code changes. *Inf. Softw. Technol.* 130, 106408 (2021), 1–15.

[164] Song Wang, Chetan Bansal, Nachiappan Nagappan, and Adithya Abraham Philip. 2019. Leveraging change intents for characterizing and identifying large-review-effort changes. In *PROMISE*. ACM.

[165] Shaowei Wang, David Lo, and Lingxiao Jiang. 2013. Understanding widespread changes: A taxonomic study. In *CSMR*. IEEE.

[166] Byron J. Williams and Jeffrey C. Carver. 2010. Characterizing software architecture changes: A systematic review. *Inf. Softw. Technol.* 52, 1 (2010), 31–51.

[167] Claes Wohlin. 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *EASE*. ACM.

[168] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. 2011. ReLink: Recovering links between bugs and changes. In *ESEC/FSE*. ACM.

[169] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *ICSE*. IEEE.

[170] Meng Yan, Ying Fu, Xiaohong Zhang, Dan Yang, Ling Xu, and Jeffrey D. Kymer. 2016. Automatically classifying software changes via discriminative topic model: Supporting multi-category and cross-project. *J. Syst. Softw.* 113 (2016), 296–308.

[171] Zhou Yang, Bowen Xu, Jie M. Zhang, Hong J. Kang, Jieke Shi, Junda He, and David Lo. 2024. Stealthy backdoor attack for code models. *IEEE Trans. Softw. Eng.* Early Access 50, 4 (2024), 721–741.

[172] Jooyong Yi, Dawei Qi, Shin Hwei Tan, and Abhik Roychoudhury. 2013. Expressing and checking intended changes via software change contracts. In *ISSTA*. ACM.

[173] Stephen W. L. Yip and Tom Lam. 1994. A software maintenance survey. In *APSEC*. IEEE.

[174] Wayne X. Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2023. A survey of large language models. *arXiv preprint* arXiv:2303.18223 (2023).

[175] Thomas Zimmermann. 2016. Card-sorting: From text to themes. In *Perspectives on Data Science for Software Engineering*. Elsevier.

Received 15 June 2022; revised 1 November 2023; accepted 2 April 2024