A Systematic Survey of Just-In-Time Software Defect Prediction

YUNHUA ZHAO, CUNY Graduate Center, USA
KOSTADIN DAMEVSKI, Virginia Commonwealth University, USA
HUI CHEN*†, CUNY Brooklyn College, USA

Recent years have experienced sustained focus in research on Software Defect Prediction (SDP) that aims to predict the likelihood of software defects. Moreover, with the increased interest in continuous deployment, a variant of SDP called Just-in-Time Software Defect Prediction (JIT-SDP) is focusing on predicting whether each incremental software change is defective. JIT-SDP is unique in that it consists of two interconnected data streams, one consisting of the arrivals of software changes stemming from design and implementation, and the other the (defective or clean) labels of software changes resulting from quality assurance processes.

We present a systematic survey of 67 JIT-SDP studies with the objective to help researchers advance the state-of-the-art in JIT-SDP and practitioners become familiar with recent progress. We summarize best practices in each phase of JIT-SDP workflow, carry out a meta-analysis of prior studies, and suggest future research directions. Our meta-analysis of JIT-SDP studies indicates, among other findings, that the predictive performance correlates with change defect ratio, suggesting that JIT-SDP is most performant in projects that experience relatively high defect ratios. Future research directions for JIT-SDP include situating each technique into its application domain, reliability-aware JIT-SDP, and user-centered JIT-SDP.

CCS Concepts: • **Software and its engineering → Software defect analysis**; **Risk management**; • **Computing methodologies →** *Cross-validation*; *Learning settings*; *Learning paradigms.*

Additional Key Words and Phrases: software defect prediction, release software defect prediction, just-in-time software defect prediction, change-level software defect prediction, machine learning, searching-based algorithms, software change metrics, change defect density

ACM Reference Format:
Yunhua Zhao, Kostadin Damevski, and Hui Chen. 2022. A Systematic Survey of Just-In-Time Software Defect Prediction. *ACM Comput. Surv.* Vol, Num, Article 0 (2022), 36 pages. https://doi.org/xx.xxxx/xxyyzza.aabbcde

1 INTRODUCTION

Software Defect Prediction (SDP) aims to predict the likelihood of defects in software artifacts, typically, in source code elements of different granularity (e.g., methods, classes, files, components). SDP has long been a topic of interest among researchers and industry practitioners alike, since knowing the predicted defect likelihood can help improve software quality and reduce its cost. For

*The corresponding author
†Also with CUNY Graduate Center, Department of Computer Science.

Authors' addresses: Yunhua Zhao, Department of Computer Science, CUNY Graduate Center, 365 5th Avenue, New York, NY, USA, 10016, yzhao5@gradcenter.cuny.edu; Kostadin Damevski, Department of Computer Science, Virginia Commonwealth University, 401 West Main Street, Richmond, VA, USA, 23284, damevski@acm.org; Hui Chen, Department of Computer & Information Science, CUNY Brooklyn College, 2900 Bedford Avenue, Brooklyn, NY, USA, 11210, huichen@acm.org.

0360-0300/2022/0-ART0 $15.00
https://doi.org/xx.xxxx/xxyyzza.aabbcde

instance, developers can use SDP for prioritizing maintenance tasks, planning activities to reduce technical debt, estimating cost and needed resources for project Quality Assurance (QA) efforts, and improving the overall development process by understanding systemic factors that contribute to defect occurrences.

Modern software development is collaborative and agile, with popular trends like continuous delivery and continuous deployment that aim at building, fixing, and releasing software with greater speed and frequency. As a result of this growing trend, recent years have witnessed activities in a new sub-area of SDP, Just-in-Time Software Defect Prediction (JIT-SDP), which aims to predict the defect likelihood of each software change. JIT-SDP has introduced significant new requirements beyond ordinary SDP, e.g., streaming data [7, 68], label verification latency [7, 69], which make the prior work in SDP inappropriate.

This article is a systematic literature survey of Just-In-Time Software Defect Prediction (JIT-SDP). Our goal is a comprehensive overview of the state-of-the-art in JIT-SDP, including data sources and data preparation, independent and dependent variables, modeling techniques, performance evaluation, and a discussion of the trends and gaps in the literature. To our knowledge, this is the first comprehensive survey specifically on JIT-SDP. More specifically, the objectives of this survey are, 1) to help researchers gain a comprehensive understanding about typical concerns and current techniques at different stages of JIT-SDP workflow that would allow researchers entering this field to get up to speed quickly; 2) to help researchers identify gaps and opportunities for future studies in this area; and 3) to benefit practitioners in selecting and tailoring JIT-SDP models to their QA needs.

We identify prior surveys on SDP and list them in Table 1. These SDP surveys have focused on a variety of aspects of the SDP problem, including the specific definition of the problem (e.g., predicting a probability or binary value), selected features, data granularity, training and test data sets, model design and evaluation metrics. While some of the surveys mention JIT-SDP, they focus only on the difference in data type (i.e., JIT-SDP uses software changes), but fail to provide coverage of the more nuanced aspects of the problem. For instance, JIT-SDP introduces a label identification latency stemming from the fact that it takes time for developers to identify defects, which, in turn, changes certain past software changesets from clean to defect-inducing.

Table 1. Summary of Software-Defect Prediction Surveys

No.	Authors	Duration	Survey Coverage # of Articles Surveyed	Survey Topic
SV1	Li et al. [41]	2000–2018	49	Unsupervised SDP
SV2	Li et al. [43]	2014–2017	70	Comprehensive
SV3	Hosseini et al. [26]	2002–2017	46	Cross-project SDP
SV4	Kamei and Shihab [31]	1992–2015	65	Comprehensive
SV5	Malhotra [46]	1995–2013	64	Within-project & cross-project SDP
SV6	Radjenović et al. [60]	1991–2011	106	Software metrics for SDP
SV7	Hall et al. [21]	2000–2010	208	Within-project & cross-project SDP
SV8	Catal et al. [8][1]	1990–2009	68	Datasets, metrics, and models
SV9	Fenton and Neil [16][2]	1971–1999	55	Defect, failure, quality, complexity, metrics, and models

[1]Catal et al. [8] investigate 90 software defect/fault prediction papers in their survey, but only cite 68. We use this as the number of papers studied in their survey.
[2] Fenton and Neil [16] do not list explicitly the paper surveyed, and we only count the papers relevant to software metrics, defects, faults, quality, and failures.

The specific objectives and contributions of this survey are the following.

(1) understanding the scope of JIT-SDP research and the problems it tries to solve, in the broader context of SDP.
(2) survey of the input data and the commonly used features for JIT-SDP, which are not only helpful to build JIT-SDP models, but also aid understanding on the empirical relationship between factors in software development and defect occurrences.
(3) explanation of model building techniques including machine learning (ML) techniques and searching-based approaches for JIT-SDP. For instance, what machine learning techniques have been leveraged thus far to improve JIT-SDP performance.
(4) evaluation strategies and criteria used in the existing JIT-SDP models, including their strengths and limitations.
(5) synthesis and meta-analysis of the prior JIT-SDP studies including a discussion of open questions and future directions.

We organize this survey as follows. Section 2 describes the methodology we used to conduct this survey. In Section 3, we present JIT-SDP in detail and examine the problems in each step of the entire workflow.

We synthesize the prior research and conduct a meta-analysis of the research in Section 4, followed by a discussion of the future research directions in Section 5. Finally, we provide a brief summary of this study in Section 6.

2 REVIEW METHODOLOGY

Following Kitchenham et al. [38, 39], a systematic literature review process consists of planning the review (including identifying the need for the review, specifying the research questions, and developing a review protocol), conducting the review, and reporting the review. In the previous section, we outline the need to conduct this review and list the review objectives. For our review protocol, in this section we define the digital libraries that serve as sources for the literature search, and describe the search queries and the review process used to determine whether to include or exclude an article.

SDP has been a long standing research subject for nearly half a century, since the 1970s. There are a number of existing systematic surveys of the literature. Considering this, we divide the review methodology into two phases: 1) a meta survey on SDP, i.e., a tertiary review or a systematic review of systematic reviews [37]; and 2) a focused survey of JIT-SDP. The meta survey serves three purposes: 1) to justify the need for a focused survey on JIT-SDP; 2) to provide background information for JIT-SDP, such as, clear definitions of defect and SDP; and 3) to determine the distinct aspects of JIT-SDP to focus our survey on. For the meta survey, we identify and analyze 9 SDP surveys (Table 1). As a result of the meta survey, we focus our discussion on characteristics unique to JIT-SDP, while providing references to the existing SDP surveys for the common attributes (such as, some of the evaluation criteria in Section 3.7). Below, we discuss the article selection process of the focused survey on JIT-SDP. We provide a detailed description of the complete survey protocol, including the methodology used for the meta survey, in the online supplement that accompanies this article.

2.1 Identification of Research Articles for JIT-SDP Survey

We begin the literature search with a digital library keyword search. To reduce the chance that we miss any significant prior studies, we complement the keyword search with a snowball process.

Exclusion and Inclusion Criteria. Highly-cited JIT-SDP articles, such as, Kamei et al. [32] and Kim et al. [35] trace the first change-level defect prediction research to Mockus and Weiss, published in 2000 [49]. We conclude the keyword search in November 2021. As such, we consider peer-reviewed research articles written in English published in a journal or conference proceedings between 2000 and November 2021. We include only articles that study predictive modeling for JIT-SDP whose prediction is on the level or sub-level of software changes; however, exclude non-peer reviewed articles, posters, and abstract-only articles.

Digital Libraries. To locate existing SDP surveys and JIT-SDP studies, we search the following popular digital libraries: 1) ACM – https://dl.acm.org; 2) IEEE Xplore – https://ieeexplore.ieee.org/; 3) ScienceDirect – https://www.sciencedirect.com/search/; 4) SpringerLink – https://link.springer.com/; 5) Wiley – https://onlinelibrary.wiley.com/. These digital libraries archive and index leading journals and conference proceedings in Software Engineering and related fields.

Digital Library Keyword Search. Researchers refer to JIT-SDP by different terms. Kamei et al. [32] coined the term "Just-in-time" Software Quality Assurance for change-level defect prediction. Other researchers, such as, Jiang et al. [29] refer to it as "change-level" defect prediction. Aside from these, researchers and practitioners have used a range of terms to refer to the scenarios when software exhibits undesired behavior or outputs. These terms include "*defect*", "*fault*", "*bug*", "*error*", "*failure*", and "*exception*". These occur either in a "*software*" or in a "*program*". Considering these, we formulate our digital library search queries semantically as "`((((fault OR defect OR bug OR exception OR failure OR error) AND (prediction OR model))) OR ((fault OR defect OR bug OR exception OR failure OR error) AND risk AND (assessment OR prediction OR model))) AND (just-in-time OR change) AND (year >= 2000)`". With the queries, we search the digital libraries for keyword matches appearing in the title, abstract, and metadata of the article.

Literature Selection via 2-Pass Review. We combine all of the search results from the digital libraries and remove duplicates and divide the set of articles among the authors of this survey to evaluate whether to include or discard an article. The division ensures that we assign each article to two of the three authors and each article goes through two reviews by the two assigned authors (thus, the 2-pass review). Each author follows the following process. First, we remove any article whose title clearly indicates that it is not relevant. Second, for the remaining articles, we evaluate whether or not to include them by reading the abstract. Finally, we convene a meeting and resolve the difference via a discussion.

Literature Snowballing. The digital library keyword search may not identify all of the relevant studies. To alleviate this problem, we use the snowball method to discover new studies starting with the identified articles from the previous step. We consider a variant of the snowball method called the backward (or reverse) snowball where we examine the references of an identified article. Empirical evidence suggests that the snowball method should be effective to locate "high quality sources in obscure locations [20]." We apply the snowball method to the 55 JIT-SDP studies resulted from the digital library keyword search and 2-pass review, examining all of the referenced articles, removing duplicates, and selecting additional articles by reading the titles and abstracts.

Literature Identification Results. Table 2 provides a summary of the result of the above literature identification process. The final result is 67 JIT-SDP articles. The full list of the articles and a brief one-sentence summary of each is available in the online supplement.

3 JUST-IN-TIME SOFTWARE DEFECT PREDICTION

Fenton and Neil observe that "[d]efects, like quality, can be defined in many different ways but are more commonly defined as deviations from specifications or expectations which might lead to failures in operation [16]." A software failure is observable software misbehavior, however, a defect

Table 2. JIT-SDP Literature Search Results

Digital Library & Sources	Additional Constraint	# of Articles	
		Library Search or Snowballing	JIT-SDP after 2-Pass Review
ACM	–	196	
IEEE Xplore	–	55	
ScienceDirect	Research articles	269	55
SpringerLink	–	334	
Wiley	Computer Science	27	
Snowballing	On 55 JIT-SDP papers	2563	12

may not always lead to a failure. Software Defect Prediction (SDP) aims to predict the existence of defects, colloquially called bugs in software artifacts, typically in the software source code.

This section summarizes the progress in Just-in-Time Software Defect Prediction (JIT-SDP) described in the 67 studies we identified. To begin with, we discuss the motivation for JIT-SDP, in contrast to ordinary SDP. Next, we provide an overview of the typical JIT-SDP workflow, focusing on specific sub-problems that researchers have been concerned with. Throughout, we highlight the existing approaches for these problems in the JIT-SDP research literature.

3.1 Definition and Overview of JIT-SDP

The majority of prior research has assumed that SDP closely follows the project release schedule, i.e., leveraging snapshots and defects of the software from its historical releases to predict defect-related variables, e.g., defect density, defect severity on software modules of varying granularity (packages, files, classes, methods). We refer to this type of SDP as *Release SDP*. A recent alternative to Release SDP is to leverage software change histories to predict potential defect-inducing software changes as soon as they are committed into the repository ("just-in-time"), i.e., JIT-SDP. Developers typically record all software changes as commits to a Source Code Management (SCM) system or a Version Control System (VCS). Kamei at al. [32] argue that JIT-SDP has several benefits. Software changesets are typically smaller and each has a descriptive log message. Therefore, when it comes to carry out QA activities (e.g., code review) for changesets predicted as defect inducing, it also likely requires less effort from the developers. In addition, since each change has a single committer, it is usually easier to locate expertise to conduct the QA activities for the change. Finally, developers may quickly examine their own potentially defect-inducing software changes as they make commits to the SCM, when a JIT-SDP model prompts them to do so. In this scenario, it is likely that the developers have fresh memory about the recent changes, which eases their cognitive load. Due to these benefits, JIT-SDP is increasingly being emphasized by researchers in recent years.

Figure 1 provides an overview of the overall JIT-SDP workflow. The workflow begins with identifying and assembling the *Data Sources* that drive the model building process. The data sources include software code changes, issue reports, commit messages, and others. Section 3.2 provides a detailed discussion about the data sources. Next, the *Data Acquisition* step is to convert the data into "raw" feature vectors, e.g., to compute software change metrics from the changesets; more details about this are in Section 3.3. From the "raw" feature vectors, we *prepare* the feature vectors that are ready for building the model. We commonly refer to this step as *preprocessing*, discussed further in Section 3.5. The next two steps are *model building* and *model evaluation*, which are discussed in Sections 3.6 and 3.7. It is important to note that for supervised JIT-SDP models, we must have changeset data that is labeled as defect-inducing or clean for training. Unsupervised models do not need labeled data for training. However, for both supervised and unsupervised models, labeled changeset data are necessary for model evaluation. To assign a label of defect-inducing or clean to a software change, we begin with a known fix change, a change that fixes a defect. We can identify fix changes by keywords search in SCM commit messages, or by using an Issue Tracking System (ITS) [32, 75]. From the fix change, we search the history of the changes to identify the earliest change that introduces the defect, which is typically via invoking an algorithm called SZZ [36, 67]. There are two variants of SZZ, ITS SZZ and ad hoc SZZ (also called, approximate SZZ) depending on whether we identify fix changes using ITS or using keyword search [32, 75]. We refer to change labelings via these two variants of SZZ as *ITS change labeling* and *ad hoc change labeling*. Section 3.4 provides an overview of SZZ and its use for the change labeling process in JIT-SDP.

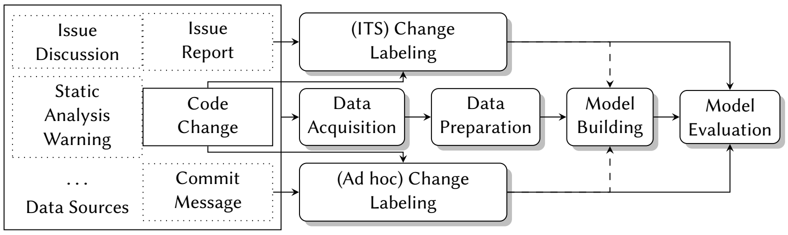


Fig. 1. Overview of JIT-SDP workflow

3.2 Data Sources

What types of SCM data are useful for JIT-SDP? Software change history data is the primary data source for JIT-SDP. However, several other types of data are also useful, including commit log messages, issue reports, issue discussions, change/modification requests, code reviews, and static program analysis warning messages.

3.2.1 Software Change History and Changeset. A software change (i.e., changeset) is the difference between two revisions of the software in a SCM (e.g., `git`). Upon completing a modification to a software project, a developer records the change to a SCM using the `commit` command, which results in a creation of a new revision number in the SCM. We compute a software change as the difference of two successive revisions in the SCM, e.g., the output of the `git diff` command. Figure 2 is an example of a software change computed by the `git diff` command containing two hunks. Each of the hunks corresponds to a part of a file in the commit, consisting of one or more nearby modifications, i.e., lines of code added or deleted.

Discussion. There are complexities extracting software change data sets that impact the quality of the data. First, there have been a variety of `diff` algorithms to compute differences between two files. For instance, `git` defines four `diff` algorithms, i.e., `myers`, `minimal`, `patience`, and `histogram`. Nugroho et al. [53] observe that there is a considerable difference among the results extracted using different `diff` algorithms. The difference is at the numbers of changed lines and the locations of change lines. Via a manual analysis of multiple Java projects, they observe that the `histogram` algorithm produces more accurate `diffs` and assert that the improvement of `diff` quality can lead to more accurate computation of code churn metrics and identification of defect inducing changes. However, whether and how the accuracy of `diffs` impact performance of JIT-SDP is unclear as the choice of the `diff` algorithm is rarely in the discussion of prior JIT-SDP studies.

Second, SCMs have varying capabilities to maintain revisions. In earlier SCMs like `CVS`, revision numbers are per file. In more recent SCMs, including both `git` and `subversion`, revision numbers

```
 1 diff --git a/parallel/help.cpp b/parallel/help.cpp
 2 index fba9e3b..59d1e9f 100644
 3 --- a/parallel/help.cpp
 4 +++ b/parallel/help.cpp
 5 @@ -4,7 +4,8 @@
 6  void print_help(const char *progname) {
 7     fprintf(stderr, "Usage: %s <NUMBER> <NUM_WORKERS>"
 8         "\nNUMBER: an integer greater than 1."
 9 -       "\nNUM_WORKERS: the number of worker threads (must be greater 0).\n",
10 +       "\nNUM_WORKERS: the number of worker threads (must be greater 0).\n"
11 +       "\nExit with 0 if perfect numbers, not otherwise.\n",
12         progname);
13  }
14 
15 diff --git a/parallel/perfectnum.cpp b/parallel/perfectnum.cpp
16 index a10d357..26fcbd8 100644
17 --- a/parallel/perfectnum.cpp
18 +++ b/parallel/perfectnum.cpp
19 @@ -1,4 +1,5 @@
20  #include <cstdio>
21 +#include <pthread.h>
22  ......
```

Fig. 2. A software change evaluated as a `diff` of two successive revisions of a toy C++ application with command, `git diff --diff-algorithm=histogram --patch db44335 b8dff778`. The command is to compute a `diff` for the software change $C_{db44335}$ as $C_{db44335} = \text{diff}(c_{db44335}, c_{b8dff78})$ where $c_{b8dff78}$ is the parent commit and $c_{db44335}$ the child. Here, `db44335` and `b8dff78` are "short hashes" that identify the two commits in the change history in the `git` repository. The `diff` algorithm is the `histogram` algorithm and the format of the `diff` is the `patch` format. The change contains two hunks, each of which corresponds to the modification to a file in the workset. Line 1 marks the beginning of the 1st hunk for the modification to file `help.cpp`. Each hunk defines a succession of lines modified via lines added or removed marked by the "+" and "-" signs. Line 15 marks the beginning of the 2nd hunk for `perfectnum.cpp`.

are per workset where a workset is a list of files that a developer changes together. Some software projects transition between different repositories (e.g., `CVS` to `git`). However, it is unclear whether these changesets are comparable and how they impact the interpretation of the results.

Third, `git` defines a staging area that stores information about what will go into the next revision, which allows a developer to control which modifications will be in the next revision; `subversion` has no such capability and includes all modifications in the next revision. This difference may have an impact on the likelihood of commits that reflect an intentional, coherent change to the software.

Fourth, there have been two versioning models, i.e., the lock-modify-unlock and copy-modify-merge models, and the later model allows concurrent modifications to the same file. SCMs like `git` and `subversion` adopt the copy-modify-merge model, and as a result, there are revisions that are only the merges of concurrent modifications. As such, a merge revision in the SCM has two parent revisions.

Fifth, distributed SCMs like `git` record all revisions locally until the developer pushes the local revisions to the central repository. As a result, anonymous development branches are common in `git` repositories. In addition, with local revisions, the recorded commit time is the local time of the committer. This can cause a problem if the committer has a misconfigured clock; this may be rare but it does occur. A more common problem is that the commits from two or more branches can interleave in commit time. As a result, we may deduce an incorrect order of successive changes and an incorrect parent commit if we simply use the chronological order to determine the parent and child commits to compute a software change. Trautsch et al. [75] observe this phenomenon and build a Directed Acyclic Graph of commits to "gain improvements with regard to changes on different branches." However, they do not provide a comparison to the case of identifying parent commits via commit time. Tan et al. [69], Cabral et al. [7], and Tabassum et al. [68] study online JIT-SDP and only leverage the chronological order of software changes. How we should best determine the order of software changes and how the order impacts on defect prediction remains an open research question.

3.2.2 Commit Message. Most SCMs require that developers enter a commit log message to describe concisely the nature of the committed change. Commit messages are useful to JIT-SDP in two ways. First, they are useful to label a change as defect-inducing or clean, which we discuss in Section 3.4; most researchers use commit messages in this context. Second, there are several studies that use features extracted from the commit messages as independent variables (or features) for defect prediction [3, 24, 25, 35].

Discussion. It is uncommon for JIT-SDP research to use commit messages as sources of independent variables (i.e., features), however, this may be worth further exploration. Barnett et al. [3] propose a metric that measures the level of detail in commit messages. They demonstrate that adding the metric can significantly improve AUC-ROC (see Section 3.7 for the definition of AUC-ROC) and explanatory power of a JIT-SDP model with change-based metrics. Hoang et al. [24] also observe that their deep neural network JIT-SDP model has improved AUC-ROC when adding commit messages to the features. In addition, commit messages may also be useful to reduce noise in the software change data. For instance, Herzig et al. [23] investigate tangled software changes where developers commit unrelated or loosely related code changes in a single commit transaction. They use commit messages to identify potential tangled changes and show that untangling tangled changes can result in more accurate regression-based defect prediction models.

3.2.3 ITS Data. Similar to commit messages, data in a project's ITS can help establish links between changesets and issues that report defects; this link is crucial to label a change as defect-inducing or clean, as discussed in Section 3.4. ITS data can also be used to construct features for JIT-SDP, albeit studies that do this are infrequent.

Discussion. Using ITS, developers describe and discuss change requests, comment on the code for code review, and propose future improvements. Although uncommon, ITS data, such as, the issue reports, discussions, change requests, can be useful to JIT-SDP, in particular for predicting on future changes addressing the reported issues. Tourani and Adams [74] and Tessema and Abebe [72] appear to be the only studies that investigate the use of some of these ITS data as features for JIT-SDP. Both show that their JIT-SDP models built using the features extracted from this type of data in addition to software change metrics outperform those using the software change metrics alone. There are two challenges utilizing the types of data as features. First, many changesets may not have a corresponding issue report in the ITS. Second, it requires to establish links between ITS data and changesets in the software repository. Noticing the difficulty of this requirement, Paixao et al. [54] curate a data set of code reviews and link them to revisions of source code. Data sets like this may help a broader adoption and further exploration of the ITS data in defect prediction.

3.2.4 Static Program Analysis Warning Messages. Warning messages produced by static program analysis tools (e.g., `lint`), although rarely, have also been used as a data source of information for building JIT-SDP models.

Discussion. Trautsch et al. [75] derive from the warning messages a warning density feature, i.e., the ratio of the number of static analysis warnings to the product size. Their results show that features extracted from the static program analysis warning messages are among the top 10 most important features for defect prediction. Static program analysis warning messages can have a value for defect prediction. However, a caveat is that a full static program analysis takes significant computational time to complete on a large code base [61] and for each changeset. To use this type of data broadly for JIT-SDP, we need to investigate how we can reduce the computational costs, perhaps by performing only partial, lightweight static analysis.

Table 3. Manually Designed Features for JIT-SDP

No.	Metrics or Features	Description	Example Study
1	Diffusion	how a changes is spread out, e.g., the number of files touched	Kamei et al. [32]
2	Purpose	the purposes of a change, e.g., is it a fix?	Kamei et al. [32]
3	Size	the magnitude of the change like the size of the change	Kamei et al. [32]
4	History	the history of the change, e.g., previous changes to the files touched	Kamei et al. [32]
5	Experience	developers' experience, e.g., the number of changes the developer made	Kamei et al. [32]
6	Code Churn	the size of a code change	Liu et al. [45]
7	Change Context	the lines of code surround the changed lines in a software change	Kondo et al. [40]
8	Indentation	the number of indentations of a change	Kondo et al. [40]
9	File-level Process Metrics	file-level change characteristics	Pascarella et al. [55]
10	Commit Message	the features characterize commit messages, e.g., the bag-of-words features, the message length, and the message content features	Tan et al. [69], Barnett et al. [3]
11	ITS Features	human discussion like code reviews, issue reports, change requests, and discussions	Tourani and Adams [74]
12	Static Analysis	static program analysis warning messages, e.g., message density	Trautsch et al. [75]

3.3 Feature Acquisition and Processing

Upon identifying the necessary and useful data sources, the next step is to transform the data into a format suitable for use by a JIT-SDP model. The primary questions here are as follows. *What information do we need to extract from the data?* In other words, *what are the independent variables that we use to build a JIT-SDP model?* In addition, *how do we extract them from the several types of input data discussed in the previous section?*

There are two approaches for these, feature engineering and feature learning. The former is to design features manually. The later is to algorithmically learn feature representations from the data, which is an emerging approach due to the recent advancements in deep learning, e.g., Deep Neural Networks (DNN). Table 3 is a summary of the categories of metrics proposed for JIT-SDP. The detailed list of metrics in each of the categories is in the online supplement.

3.3.1 Software Change Metrics. Software change metrics are those directly computed from the software code of the changes, e.g., the `diffs`. Rows 1 to 9 in Table 3 are the categories of software change metrics designed and evaluated in JIT-SDP studies. Table 4 lists the metrics used by Kamei et al. [32] plus the code churn metrics by Liu et al. [45] that JIT-SDP studies frequently cite.

3.3.2 Commit Message Features. Since commit messages are written as natural language text, features to encode them are usually borrowed from the natural language processing literature. These features include bag-of-words, the message length, and the message content features [3, 69].

3.3.3 ITS Data Features. ITS contains two types of data, natural language text, such as, issue reports, issue discussions, and code reviews and meta-data, such as, the type of issues and the issue creation time. Prior JIT-SDP studies design and investigate features mostly capturing the characteristics of the ITS data other than actual textual contents [72, 74].

3.3.4 Static Program Analysis Metrics. Trautsch et al. [75] collect static program analysis warning messages using two popular tools, `PMD` and `OpenStaticAnalyzer`. From these warning messages, aimed at JIT-SDP, they derive several warning density metrics.

Table 4. Example Software Change Metrics [32, 45]

Category	Metric	Description	Category	Metric	Description
Diffusion	`NS`	Number (#) of modified subsystems	Purpose	`FIX`	Whether the change is a defect fix
	`ND`	# of modified directories	History	`NDEV`	# of developers who modified the files
	`NF`	# of modified files		`NUC`	# of unique changes to modified files
	`Entropy`	Distribution of modified code across each file		`AGE`	Average time interval between the last and current change
Size	`LA`	Lines of code (LOC) added	Experience	`EXP`	Developer experience
	`LD`	LOC deleted		`REXP`	Recent developer experience
	`LT`	LOC in a file before the change		`SEXP`	Developer experience on a sub-system
	`Churn`	Size of the change, i.e., `LA` + `LD`			

3.3.5 Feature Representations. We refer to the type of features learned automatically via an algorithm as feature representations. Early JIT-SDP studies treat commit message and software code change as text and use the output of spam filters as features [3, 51]. More recently, deep learning, i.e., deep neural networks, has become the technique of choice to learn feature representations for JIT-SDP. For this, there are two general approaches. One is to directly compute from software changes (e.g., tokenized `diffs`) the "end-to-end" feature representation [24]. Examples of this approach are in Hoang et al. [24, 25]. The other approach is to take human designed features as inputs to learn feature representations, which Yang et al. [83] refer to as the "expressive features". More recent examples of this approach are in Zhao et al. [92] and Xu et al. [78].

3.4 Defect-Inducing Label Assignment

JIT-SDP's aim is to detect whether a change is defect inducing or not. To accomplish this, most approaches use supervised or unsupervised machine learning. A precondition to building and evaluating a supervised JIT-SDP model is the availability of a set of labeled software changes, the label of each of which is either defect-inducing or clean. While unsupervised JIT-SDP models do not require a labeled set of software changes to build the model, which is a strong advantage over supervised techniques [17, 28, 45, 80, 86], they also require access to a labeled set of software changes in order to be adequately evaluated. There is an intricacy of labeling changes as defect-inducing or clean. For instance, it is straight forward to label a software module as defective because the module is defective when the module fails a test. However, the test does not reveal to us which software change introduces the defect in the module that leads to the test failure. The primary question becomes, *how do we assign the defect-inducing (i.e., defective) or clean label to a software change?*

To label a software change as defect-inducing or clean, we generally follow a two-step approach to examine the change history of the software project [35], (1) to identify a defect-fixing change; and (2) to locate the prior change that introduced the defect in Step 1, i.e., the defect-inducing change.

For Step 1, to identify a defect-fixing change, we can examine either commit log messages , or issue reports in ITS, or both. There are two commonly used heuristics. One is to search for a natural language pattern in the text, e.g., "fixes", "fixed", "resolves", "resolved", "defect", and "bug" [32]. The other is to look for references to issue reports, e.g., "#123abc" [47].

For Step 2, to locate the corresponding defect-inducing change, we search backward in the change history to find the changeset that introduced the defect. Conceptually, this is possible because the current change fixes a defect, therefore the added lines are the fix and the removed lines contain the defect. To locate the defect-inducing changeset, we need to locate in which change the removed lines first appear in the change history, i.e., as part of the added lines in a prior change. This change is the one that induces the defect that the defect-fixing change resolves.

Following the above conceptual framework, Śliwerski et al. [67] propose an algorithm to automatically identify defect-inducting changes, based on defect-fixing changes. The algorithm is now known as SZZ, named after the first initials of the three inventors Śliwerski, Zimmermann, and Zeller [67]. The SZZ algorithm scales up the labeling of software changes and the use of SZZ in JIT-SDP research is ubiquitous.

Discussion. Investigations of the accuracy of this automated software change labeling have shown that it may not always be easy to identify a defect-fixing change (i.e., Step 1) [30, 35, 75] and it can also be difficult to locate the corresponding defect-inducing change (i.e., Step 2) [36, 52]. There have been several implementations (or variants) of SZZ with the aim of addressing some of these issues in Step 2. The questions are: 1) how should we identify defect-fixing changes; 2) which implementation of SZZ should we use; and 3) how do these choices impact JIT-SDP results? Trautsch et al. [75] differentiate "ITS SZZ" from "Ad Hoc SZZ", where for the former they identify defect-fixing changes using explicit links in ITS and SCM, while for the later they use heuristics. They show that ITS SZZ offers an advantage in predictive performance. However, explicit links between ITS and SCM are not always available [32]. Quach et al. [59] discuss their choice of the SZZ implementations and give their preference to MA-SZZ [12]. Fan et al. [15] investigate the impact of the mislabeled changes by SZZ and assert that RA-SZZ [52], an extension to MA-SZZ is most accurate [15]. However, Rosa et al.'s recent investigation on SZZ implementations does not yield a clear support for these exact preferences [64]. Some researchers still suggest manual vetting of SZZ results to ensure highest quality data is used to construct the JIT-SDP model [15, 59]. However, this is time-consuming and difficult to do for developers that are unfamiliar with a particular software project.

3.5 Data Transformation

The input data for building JIT-SDP models is typically a vector, consisting of the values of the features for a specific changeset. The vector is the result of the steps described in Section 3.3. The collection of vectors for a number of changesets forms a matrix. In most cases, the matrix data, e.g., software metrics of changesets, requires additional transformations prior to building a JIT-SDP model. Based on such a matrix, we divide the transformation techniques into two categories, column-based and row-based [26]. A column is in effect the values of a feature for all of the software changes while a row represents the "raw" feature values of a software change. The primary questions are two: *How do we transform each column of the "raw" feature matrix and why do we need this column-based processing? How do we select or transform the changesets represented by the feature "matrix" (i.e., the rows), and why and when do we need to perform the row-based processing?*

3.5.1 Column-based Processing.

Building a JIT-SDP model is in part to automatically discover regularities in the software change data that are predictive of whether a future change is defect inducing. JIT-SDP models encode the regularities in a set of model parameters. An important concern is whether we can uniquely determine the model parameters from the input data, either theoretically or numerically. This is called the parameter *identifiability* problem [62], which can be alleviated with column-based processing.

Dealing with Skew. Most features of the software change data are highly skewed, e.g., some variables vary between 0 and 1 while the others are several orders of magnitude higher; many features are not normally distributed. Most statistical and machine learning algorithms expect that the values of independent variables are standardized, e.g., the manual of the popular machine learning library `scikit-learn` [56] states, " [s]tandardization of data sets is a common requirement for many machine learning estimators implemented in `scikit-learn`; they might behave badly if the individual features do not more or less look like standard normally distributed data: Gaussian with zero mean and unit variance." There are three methods dealing with the data skews in the JIT-SDP studies, and we discuss them as follows.

The most popular method in the JIT-SDP studies to deal with this standardization problem is to apply a logarithmic transformation to the software metrics whose values are numeric [11, 14, 15, 28, 30, 42, 49, 74, 80, 91]. For instance, several studies apply the logarithmic function $\ln(x+1)$ to transform numeric variable x [14, 15, 30, 80]. An implied assumption here is that the software metrics take non-negative values, but some are less than 1. The logarithmic function $\ln(x+1)$ ensures the transformed features are positive for all of the software metrics. Another method is to apply a z-score function to the variables [40, 82, 93, 94]. The z-score function is $z(x) = (x - \mu)/\sigma$, where μ is the sample mean of the variable and σ is the sample standard deviation of the variable. Similar to the z-score method but involving simpler computation is the max-min method that computes the normalized feature value as $z(x_{ij}) = (x_{ij} - \min(x_j))/(\max(x_j) - \min(x_j))$, where x_{ij} is the value of feature j of software change i, x_j represents column j of the feature matrix, and $x_j = (x_{1,j}, x_{2,j}, \ldots, x_{N,j})$, where N is the training data set size. For instance, Yang et al. [83] use this method in their study.

Dealing with Collinearity and Multicollinearity. Collinearity means that two features correlate with each other while multicollinearity is when a feature correlates with a combination of multiple features. Correlated features, such as correlated software metrics, can introduce two problems in JIT-SDP. First, numerical non-uniqueness can occur to a Logistic Regression model due to the collinearity of the features [16, 62], where numerical non-uniqueness is the inability to determine a unique set of model parameters. Second, correlated features can also lead to misinterpretation of effects of individual features on predicting defect-inducing software changes [15, 70]. Because of these, it is necessary to eliminate correlated or multi-correlated variables. A typical procedure is to use the combinations of correlation analysis, independence analysis, and redundancy analysis as exhibited by prior researches [14, 15, 32].

Discussion. De-skewing and dealing with collinearity and multicollinearity are two necessary steps. There are no strong evidences which de-skewing method discussed in the above is superior. However, as cautioned by several studies, we should de-skew the variables before we address the collinearity and multicollinearity problems [14, 15, 30]. There is a practical consideration when we address the collinearity and multicollinearity problems, such as, if two variables are correlated, which of these two should we remove [15, 18, 30, 32]? One factor to consider is that we should keep the variable that helps ease model interpretation following prior studies [15, 18, 30, 32]. For instance, Kamei et al. [30] and Fukushima et al. [18] find in their studies that variables `NF` and `ND` are highly correlated and remove `ND` since prior studies have examined the relationship of `NF` and software defects.

3.5.2 *Row-based Processing and Class Imbalance.*

Row-based processing aims to filter software changes in order to improve the generality or predictive performance of JIT-SDP models. One important problem typically addressed via row-based processing is the class imbalance problem.

Dealing with Class Imbalance. Software change data are often class imbalanced. In particular, the difference between the number of defect-inducing changesets and that of clean changesets can be significant for some software projects. For instance, the ratios of the clean changes to the defect-inducing changes of the 6 open-source projects that Kamei et al. [32] select for evaluating their JIT-SDP models range from 1.7 : 1 to 18.1 : 1 as shown in Table 5. Without transformation, most machine learning algorithms learn the traits in the majority class (i.e., clean) at the expense of learning the traits in the minority class (i.e., defective). Solutions to address this problem are to resample the original data set, typically in one of the three categories, 1) undersampling the majority class, 2) oversampling the minority class, or 3) a combination of the two. The objective is to have balanced classes, each of which has an identical number of software changes. Tan et al. [69] further divide the oversampling and undersampling class-balancing strategies into four methods, namely, Simple Duplicate, Synthetic Minority Over-sampling Technique (SMOTE), Spread Subsample, and Bootstrapping with/without Replacements. It is important to note that for properly assessing the generality of a predictive model, we ought not to apply these technique to balance the validation or test data set.

Table 5. Characteristics of Changeset Data of Selected Projects [32]

Project	# of Changes	Average LOC of Modified Files	Average LOC of Changes	# of Developers Per File	Clean (N_c)	Defect Inducing (N_d)	$N_c : N_d$
Bugzilla	4620	389.8	37.5	8.4	2924	1696	1.7 : 1
Columba	4455	125.0	149.4	1.6	3094	1361	2.3 : 1
JDT	35386	260.1	71.4	4.0	30297	5089	6.0 : 1
Mozilla	98275	360.2	106.5	6.4	93126	5149	18.1 : 1
Platform	64250	231.6	72.2	2.8	54798	9452	5.8 : 1
Postgres	20431	563.0	101.3	4.0	15312	5119	3.0 : 1

Discussion. The prior JIT-SDP studies adopt more frequently the undersampling and the SMOTE approaches than the others. For instance, Kamei et al. [32] use an undersampling approach in their study, in effect Spread Subsample with target class ratio as 1 : 1. Several studies follow Kamei et al. and use the identical approach [15, 27, 42, 74, 82, 86, 91]. Catolino et al. [9], Trautsch et al. [75], and Zhu et al. [94] choose SMOTE in their studies.

Huang et al. [28]'s experimental results show that Kamei et al. [32]'s approach of undersampling of the majority class always outperforms SMOTE, however, without sharing any experimental data on what performance metrics are in use. Zhu et al.[94] argue that the simple duplicate method is prone to over-fitting and can cause numerical stability problem; however, they do not provide any quantitative evidence. Duan et al. [14] also investigate both oversampling and undersampling approaches, comparing Kamei et al. [32]'s undersampling and SMOTE approaches over several performance metrics, and the results are inconclusive whether undersampling or oversampling is superior. For instance, their evaluation results suggest for most cases in their study undersampling contributes to higher AUC and F1 scores of some evaluated software projects while oversampling yields overall higher average AUC [14]. Although Tan et al. use the 4 bootstrap methods, i.e., Simple Duplicate, SMOTE, Spread Subsample, and Bootstrapping with/without Replacements, they use these data balancing strategies for model selection without comparing the 4 balancing strategies [69]. Therefore, researchers remain inconclusive on the optimal choice of data balancing strategies for JIT-SDP models.

Finally, there is a choice of not to balance the data. Instead, we address the problem via the design of a predictive model. When a JIT-SDP model is to minimize a loss function or to maximize an objective function, we can assign a larger class weight to the minority class than do to the majority class [92]. An ensemble model, such as, an ensemble of classifiers can as a whole use all of the data while each of the classifier uses a balanced subset of the whole data set [82, 95]. In addition, we should also be cautious about balancing the data when the interpretation of the model is our primary concern since the balanced data may change the interpretation of the model, such as, shift the ranking of the most important metrics [44]. For instance, because of this, Lin et al. [44] opt not to use any balancing techniques.

3.6 Modeling

There are several key decisions we ought to make in JIT-SDP Modeling. First, *what are the dependent variables?* The choices of dependent variables give rise to two related defect prediction models, defect prediction and effort-aware prediction. Second, *what type of algorithms are we using?* The options are usually machine learning or searching-based models. Searching-based JIT-SDP models are based on heuristics, some of them are unsupervised non-learning algorithms and do not require labeled data to learn. Most JIT-SDP models are supervised machine learning models. For the supervised models we must have changesets with defect-inducing or clean labels to draw training instances. We must also consider, *what are the learning settings, i.e., under what setting do we learn to predict defects using a machine learning algorithm?* Most prior JIT-SDP models are in batch learning setting. In batch learning, there are no requirements on the order of the training instances when we train the model by feeding the instances to it. Some argue that batch learning is not realistic and opt for an online learning setting [7, 68]. In online learning, there is a presumed order of the training and testing instances, typically, the order is the arriving order of the instances in the SCM. Next, *what is the prediction setting?* There are typically two prediction settings, within- and cross-project predictions. JIT-SDP are for aiding software quality assurance, and as such, we ought to consider the software development process and the characteristics of the data originated from the process.

In Section 3.6.1, we discuss JIT-SDP models for defect prediction, in Section 3.6.2 effort-aware models, and in Section 3.6.3 cross-project models. We then discuss how we address some characteristics of software development process and the software data in Section 3.6.4. Finally, we discuss our observation on the tug of war of the simple and complex JIT-SDP models in Section 3.6.5.

3.6.1 Defect Prediction. JIT-SDP is typically formulated as predicting defect proneness or defect inducing at the granularity of software changes.

Modeling Technique. The most common model choices are supervised machine learning techniques. There have been several investigations of unsupervised models [45, 86], however, the predictive performance of these models are not competitive. Pursuing the best performing models, JIT-SDP studies examine a variety of techniques from standalone to ensemble learners. The standalone learners include Logistic Regression [32, 49], Naive Bayes [29], Support Vector Machine [2, 35], Decision Tree [71], and Neural Networks [83] while the ensemble learners span from the ensemble of a single type of base learner, such as, Random Forest [18] to the multi-type and multi-layer ensembles [87]. Our analysis (detailed in the online supplement) shows that Logistic Regression, Tree-based models (including Random Forest, C4.5 Decision Tree and ADTree) and ensemble models (including Random Forest, XGBoost, and others) are more popular modeling techniques and the use of neural network-based models (including deep neural networks) is on the rise. Section 4.2 provides an analysis of model performance.

Prediction Granularity. JIT-SDP models are to detect defects primarily on the level of software changeset. For instance, in the context of *modern* SCMs like `git`, we predict which commit is detect-inducing. Some of the prior works also use changesets from legacy SCMs like `CVS` and the prediction target are often on logical change transactions, i.e., clusters of commits that share some commonality within a time window [32]. There are several studies that predict defects on finer granularity, such as, scoring files [55, 75], classes in the files [1], or lines added in a changeset [57, 79]. These works show that defect prediction on finer granularity can potentially further reduce quality assurance effort than that on changeset-level granularity.

3.6.2 *Effort-Aware Prediction.*
The effort required for developers to review the predicted defect-inducing changes, i.e., the QA effort, is an important design consideration.

Table 6. List of Effort-Aware Prediction Models

Model	Research	Change Score	QA Sorting Order
`DEJIT`	Yang et al. [85]	$Y(c)$/`Churn(c)`	descending (↓)
`EATT`	Li et al. [42], Zhang et al. [91]	$\sum_{i=1}^{3} p_i(c)$/`Churn(c)`	descending (↓)
`MULTI`	Chen et al. [11]	$P(c)$ and `Churn`(c)	optimization & selection
`CCUM`	Liu et al. [45]	1/`Churn`(c)	descending (↓)
`OneWay`	Fu and Menzies [17]	`LT`(c), `AGE`(c), . . .	ascending (↑)
`CBS+`	Huang et al. [28]	$P(c)$/`LT`(c)	descending (↓)
`CBS`	Huang et al. [27]	`LT`(c)	descending (↓)
`TLEL`	Yang et al. [82]	$\sum_{i=1}^{N} b_i(c)$/`LT`(c)	descending (↓)
`LT`, `AGE`	Yang et al. [86]	`LT(c)` or `AGE`(c)	ascending (↑)
`Deeper`	Yang et al. [83]	$P(c)$	descending (↓)
`EALR`	Kamei et al. [32]	$Y(c)$/`Churn(c)`	descending (↓)

$p_i(c)$. The predicted defect proneness of change c by classifier i
$P(c)$. The predicted defect proneness of change c
$b_i(c)$. The predicted bipolar change label, 1 for defect inducing and -1 for clean by classifier (e.g., decision tree) i
$Y(c)$. The predicted binary change label, 1 for defect inducing and 0 for clean.
$Y(c)$/`Churn(c)`. The change defect density, also denoted as $D(c)$.

All the studies in Table 6 define the QA effort for a change as the total lines changed, i.e., `Churn` = `LA` + `LD`. The models therein generally define an effort-awareness score to rank or to order the software changes for QA. The score is either the change defect proneness, or the change defect density, or the QA effort, or some combination of the above. Table 6 lists the change scores and the order we would conduct QA for the changes. To achieve this, we simply sort the software changes and begin QA from the beginning of the sorted changes.

Unsupervised versus Supervised Models. Among the models in Table 6, the models by Yang et al. [86] and the `CCUM` model by Liu et al. [45] are unsupervised. These models are based on simple heuristics, such as, the observation that smaller modules are proportionally more defect-prone and should be inspected first [28, 86]. For instance, Yang et al. show that models `LT`, `AGE`, `NUC`, and `Entropy` rank software changes based on these metrics and can sometimes outperform Kamei et al. [32]'s supervised model `EALR`. Yang et al's findings influence several other supervised effort-aware models. These supervised models take advantage of both of the heuristics and the power of a learning algorithm to improve effort-aware predictive performance. `CBS` and `CBS+` classify the changes as defect-inducing and clean first, then search the changes that are classified as defect inducing according to a heuristic [27, 28]. `OneWay` is based on the observation that the best heuristics vary among software projects [17]. For instance, the choice of searching-based model, such as, `LT`, `AGE`, `NUC`, and `Entropy`, should be based on the specific software project in order to maximize effort-aware predictive performance. Therefore, `OneWay` selects the best heuristics based on labeled changeset data first.

Exploration of Modeling Techniques. Recent developments in effort-aware models have been those that take advantage of advancements in predictive modeling, such as, multi-objective optimization, evolutionary computation, and ensemble models. Prior studies mostly structure supervised effort-aware models in two ways. One is to form a pipeline where they predict defect proneness of a software change via a classification model, and use the prediction to compute the effort-awareness score, such as, Huang et al. [28]. The other is to build a dedicated model to predict the effort-awareness score via a regression model, such as, Kamei et al. [32]. Chen et al. [11] take a different approach and propose a model called `MULTI` where they model JIT-SDP problem as a dual-objective optimization problem to maximize defect proneness probability while simultaneously minimizing QA efforts. Observing that prior studies often consider effort-aware JIT-SDP as a classification or regression problem, Yang et al. [85] approach the effort-aware prediction problem as an optimization problem. They design an optimization objective function called density-percentile-average (DPA) representing the average percentile of the defect density of each software change. Their model (`DEJIT`) learns model parameters by using the differential evolution algorithm to maximize the objective function. Li et al. [42] and Zhang et al. [91] propose a semi-supervised model called `EATT` – Effort-Aware Tri-Training, an ensemble model of decision trees. Section 4.3 examines the performance of some of the effort-aware models.

3.6.3 Cross-Project Prediction. We can divide supervised JIT-SDP models into two categories, Within-Project JIT-SDP (WP-JIT-SDP) and Cross-Project JIT-SDP (CP-JIT-SDP), based on the choice of training and target software projects. WP-JIT-SDP is to train a prediction model from historical software change and defect data of one project, and to use the model to carry out defect prediction or effort-aware prediction for the same project, i.e., the target project is identical to the source projects. Different from this, CP-JIT-SDP is to build a prediction model from the change and defect data of one or more projects, and to use the model to predict defect prone changes for a project. In this case, the target project may or may not be in the set of the source project. Why do we desire CP-JIT-SDP? Sometimes, the historical change and defect data of a single software project may not be available or may not be sufficient to train a JIT-SDP model, e.g., in the initial development phase of the project. Therefore, there is a practical need to develop models that can be trained using the historical data from other projects for defect prediction, which gives rise to the very idea of CP-JIT-SDP. However, there is a problem about CP-JIT-SDP. Defect prediction works because the training data and the testing data share some common "traits", e.g., their software change metrics share similar statistical distributions. However, two software projects may have significantly different characteristics. As a result, it begs the question, can applying a JIT-SDP model trained from one or more projects directly to the other yield as good predictive performance?

Fukushima et al.[18] show that JIT-SDP models trained on similar projects can perform as well in the cross-project context. Kamei et al. [30]'s further exploration indicates that we can have improved predictive performance when using three approaches, namely, data selection, data merging, and model ensembling. Data selection is to "select models trained using other projects that are similar to the testing project [30]", data merging is to "combine the data of several other projects to produce a larger pool of training data [30]", and model ensembling is to "combine the models of several other projects to produce an ensemble model [30]."

However, when the interpretation of the model is of our chief concern we may have to be cautious about using JIT-SDP models for cross-project prediction according to a recent study. Lin et al. [44] report that the most important metric of the CP-JIT-SDP model using data merging (global model) is only consistent with 55% of the studied projects' local models (a model trained from an individual project), which suggests that the interpretation of global models cannot capture the variation of the interpretation for all local models. Because of this, Lin et al. [44] advocate mixed-effect modeling that considers individual projects and contexts.

3.6.4 Design for Software Development Process and Data. JIT-SDP aims to support quality assurance activities for software development and we cannot divorce it from software development process and practice and the characteristics of the data originated from the process and practice.

Concept Drift. A necessary condition that underlies JIT-SDP models is that there are some regularities in historical software changes and defects that can be translated to future software change data. Therefore, a JIT-SDP model can discover the regularities from the *historical* data via a training process and predict defect proneness for *future* changes. Concept drift is the phenomenon that the regularities in the data gradually change or shift. In a probabilistic model, we can define concept drift more formally as "changes in the underlying joint probability distribution of the problem [13]." McIntosh and Kamei [47] are the first to investigate the concept drift problem in JIT-SDP. Via a longitudinal study of the QT and OpenStack systems. They show that the characteristics of software change data fluctuate during the life cycle of software so significantly that the performance and interpretation of JIT-SDP models trained on old software change data can degrade greatly. Bennin et al. [4] experiment on a different set of software projects, and in essence confirm McIntosh and Kamei [47]'s finding.

McIntosh and Kamei [47] address the problem by providing a guideline on how to select recent software change data to train a JIT-SDP model. A different approach to tackle the problem is via an incremental learning or an online learning where we update the model via learning from newly arrived data as proposed by Cabral et al. [7] and Tabassum et al. [68]. Cabral et al. [7] investigate a specific type of concept drift, the evolution of the unbalanced status of software change data. Additionally, for CP-JIT-SDP, Tabassum et al. [68] examine how the software change and defect data from the source projects can help prevent sudden drops in the predictive performance of the target projects, which may otherwise be the result of concept drift. These studies conclude that simply re-building classifiers from scratch over time is not enough to obtain good predictive performance over time; and without addressing the class imbalance evolution, a model can get a high rate of false alarms or miss a substantial amount of defect-inducing software changes. They also suggest ways of combining project data to improve or to stabilize the performance over time in the cross-project prediction setting.

Verification Latency. A defect-inducing change always appears before its defect-fixing change. To identify a defect-inducing change in the data set we must first observe the commit containing the fix (Section 3.4). In reality, defects are not found and fixed immediately and, therefore, there exists a verification latency, i.e., a lag time between when a defect-inducing change is committed to the SCM and identified as such. Cabral et al. [7] show that the verification latency ranges from 1 to 410 days among the 10 open source projects that they examine; and they assert that verification latency is more likely to affect newer changes than older ones.

The JIT-SDP model is impacted by verification latency by a reduction in the generality of the model as the training data and the testing data may contain falsely labeled instances. As a result, typical cross-validation, e.g., a k-fold validation may indicate a better predictive performance when all data is available. For instance, Tan et al. [69] find that among the 6 open-source and 1 proprietary projects, the precision of their cross-validation without considering the verification is 55.5%–72.0% while it is only 18.5%—59.9% with a time-sensitive validation considering the verification latency for the same data. Cabral et al. [7] factor in both class imbalance evolution, a specific type of concept drift and the verification latency in their model. Their results further confirm that it is important to take verification latency into account when we evaluate and select a JIT-SDP model.

Types of Defect. How do the types of defect impact JIT-SDP? One aspect is about the quality of data, such as, some types of defect may be more noisy than the other. Intrinsic defects stem from the code found in the project's SCM; while extrinsic ones the results of external factors, such as, errors in an external API [63]. The extrinsic defect do not have defect-inducing changes in the project's history although often reported in the project's ITS as defects, leading to misidentified defect-inducing changes by SZZ. Rodriguez-Perez et al. [63]'s investigation show that extrinsic defects negatively impact JIT-SDP models and a close examination is necessary. Quach et al. [59] study performance and non-performance defects motivated by the intuition that SZZ is more error-prone to identify defect-inducing changes for performance defects than non-performance ones. Their results are nuanced in that correcting SZZ errors on performance defects can indeed improve model performance; however, combining all SZZ identified defect-inducing labels, correct or wrong, yields the best predictive performance, in the absence of a large set of correctly labeled performance defect-inducing changes. The other aspect is, can we predict certain type of defect of our interest? Security audit of software code is a labor intensive endeavor. With a JIT-SDP model and the metrics designed to predict software vulnerability defect, Yang et al. [81] demonstrate that they can inspect more defects with less effort and their model can assist as an early step of continuous security inspections as it provides “just-in-time” feedback to the developers.

Other Data Characteristics. Jiang et al. [29] argue that different developers have different coding styles, commit frequencies, and experience levels and exhibit different defect patterns in the software change data. They propose developer-specific JIT-SDP models (or personalized models) and show that the personalized models can predict more defects than non-personalized one.

Can we predict defectiveness of a software change better by selecting a model trained using a subset of the data similar to the software change considering that there are variations in the data? We can address this problem by using Menzies et al. [48]’s concept of local model and global model. Yang et al. [84]’s empirical results show that local models underperform the global model for defect prediction and outperform them for effort-aware prediction.

Gesi et al. [19] argue that we should pay attention to other type of imbalance of the data other than class imbalance. For instance, software change data can be biased along other dimensions, such as, File Count, Edit Count, Multiline Comments, Inward Dependency Sum. They propose a deep neural network model called SifterJIT combining Siamese network and DeepJIT [24] and the model outperforms DeepJIT [24] and CC2Vec [24] due to the proposed model’s few-shot learning capability.

Software projects use branches to support ongoing software development and release. For instance, some software projects maintain a “development” or a main branch for continuous development and release branches to address issues occurred post-release. Duan et al. [14] observe that developers may apply duplicate fixes to multiple branches. They suggest that more accurate evaluation of JIT-SDP can be conducted by removing duplicate changes.

3.6.5 Debate on Complex and Simple Models. Should we use a simple or a complex model? As Breiman puts, “Occam’s Razor, long admired, is usually interpreted to mean that simpler is better. Unfortunately, in prediction, accuracy and simplicity (interpretability) are in conflict [6].” Past JIT-SDP studies appear to exemplify this argument or debate.

There have been a growing number of studies using complex machine learning techniques, e.g., deep neural networks, to build complex JIT-SDP models [19, 24, 25, 78, 92, 93]. These works show improved predictive performance over prior models. Recently Pornprasit et al. [57] design a Random Forest model (called `JITLine`) and Zeng et al.[89] a Logistic Regression model (`LAPredict`). Their models not only yield better predictive performance than but also multiple orders of magnitude faster than the state-of-the-art deep neural network models. The debate goes on.

3.7 Evaluation

Depending on the learning setting, there have been several evaluation strategies and sets of evaluation criteria in the literature. First, we discuss the performance metrics (i.e., evaluation criteria) for JIT-SDP, specifically for defect proneness prediction, effort-aware prediction, and online learning. Second, we summarize the evaluation strategies used for JIT-SDP.

3.7.1 Evaluation Criteria for Defect Prediction. The dependent variable in JIT-SDP is usually defect proneness, i.e., the probability that a change is defect-inducing, or defect inducing, i.e., defective or clean. To evaluate that for a JIT-SDP model, researchers commonly adopt the evaluation criteria for binary classification. The common evaluation criteria include *accuracy*, *precision*, *recall*, *F-measure* (e.g., *F1 or F2 score*), and *AUC-ROC*. Hall et al. [21] provide a thorough investigation about this in the context of SDP. The evaluation here is similar to that of SDP. Readers should refer to Hall et al. for the definitions of these evaluation criteria.

3.7.2 Evaluation Criteria for Effort-Aware Prediction. There are several evaluation criteria to assess effectiveness of JIT-SDP models in terms of QA effort. These criteria include P_{opt} and $Recall$@20%.

P_{opt} is the area under the normalized cost-effectiveness curve [11, 17, 28, 32, 42, 45, 73, 86, 91]. For model m it is, as [45], $P_{opt}(m) = 1-(Area(m_{optimal}) - Area(m))/(Area(m_{optimal}) - Area(m_{worst}))$

The optimal model $m_{optimal}$ corresponds to the one that is to sort all changes in descending order by the actual defect density. The worst model m_{worst} is to sort all changes in ascending order by the actual defect density. In the predicted model m, we sort all the changes in descending order by the predicted defect density or in an order specified in a model by a proposed defect score or rank [32, 42]. P_{opt} is a threshold-free evaluation criteria.

In the literature, there are several evaluation criteria based on a threshold of the amount of QA effort, such as, 20% of QA effort. A common definition of the total QA effort is the sum of all code churns for all changes in the test data set, i.e., $E_{total} = \sum_{i=1}^{N_{test}} \texttt{Churn}(c_i)$ where c_i, $i = 1, 2, \ldots, N_{test}$ are the changes in the test data set. Then, a 20% effort is $20\% E_{total}$.

$Recall$@20% is to measure the ratio of defect-inducing changes when we spend 20% of total QA effort [14, 15, 73]. Some studies refer to it as *ACC* [14, 15, 73] and others $PofB$@20 [29, 45, 82, 83, 94]. "PofB20" is from the phrase "the Percent of Buggy changes at 20% of effort".

To estimate $Recall$@20%, we spend 20% of QA effort reviewing the predicted defect-inducing software changes and count the actual ones. We call this count $NofB20$ (the Number of Buggy changes at 20% of effort) and use this to compute the Recall (i.e., $NofB20/N_P$ where N_P is the number of known defect-inducing changes. $NofB20$ is equivalent to True Positives within 20% QA effort. Similarly, we can also use this effort threshold to form a confusion matrix to compute precision, F1 score, and so on and we would have precision@20%, F1-score@20%, and so on [80].

Huang et al. [27, 28] propose two evaluation criteria, *PCI*@20 and *IFA*. The motivation for *PCI*@20 is from the following observation. The distribution of change size, i.e., code churn is highly skewed and it is typical that a significant portion of changes are small. A JIT-SDP model, such as, Yang et al. [86]'s unsupervised models like `LT` and `AGE` can yield high effort-aware recall, e.g., $Recall$@20; however, they may require developers to review significantly more but smaller changes than alternate models, such as, Kamei et al. [32]'s `EALR`. Due to the frequent context switches when reviewing the numerous but small changes, developers may have low productivity due to fatigue.

Huang et al. [27, 28] argue that frequently switching between reviewing many small changes may lower a developer's productivity compared to infrequently switching between reviewing fewer, larger changes. They define *PCI*@20 to account for the effects of context switching on developers who inspect the changes. *PCI*@20 is the Proportion of Changes Inspected when inspecting 20% lines of code of the changeset.

Another problem is related to false positives, in particular, when reviewing numerous small changes. Due to the false alarms, some developers may choose to give up, which is the so-called "tool abandonment problem." To account for this, they define *IFA* as the number of the Initial False Alarms encountered before we find the first defect. A high *IFA* may lead to abandonment of the QA tool built using a JIT-SDP model.

3.7.3 Evaluation Criteria for Online and Time-Sensitive Learning. In an online learning JIT-SDP model, an evaluation criterion can be a function of time. Cabral et al. [7] and Tabassum et al. [68] propose a time-varying G-mean of $Recall0$ and $Recall1$, where $Recall0$ is Recall of the clean class and $Recall1$ is Recall of the defect-inducing class. The time-varying G-mean, denoted as $c_G(t)$ is, $c_G(t) = (c_{RC0}(t)c_{RC1}(t))^{\frac{1}{2}}$ where $c_{RC0}(t)$ is $Recall0$ and $c_{RC1}(t)$ $Recall1$ at time t. Cabral et al. [7] and Tabassum et al. [68] compute $Recall0$ and $Recall1$ prequentially and use a fading (or decay) factor to enable tracking changes in predictive performance over time, i.e., $c_{RCi}(t) = \theta c_{RCi}(t-1) + (1-\theta)\mathbb{1}_{\hat{y}=i}, i = 0, 1$ where $\hat{y} \in \{0, 1\}$ is the predicted class label, $\mathbb{1}_{\hat{y}=i}$ is the indicator function that evaluates to 1 if $\hat{y} = i$ is true or 0 otherwise, and θ, $0 \leq \theta \leq 1$ is the fading or decay factor. Tabassum et al. [68] choose $\theta = 0.99$ in their study while Cabral et al. [7] vary it among $\{0.9, 0.99, 0.99\}$ to examine its effect.

3.7.4 Validation Strategies and Evaluation Settings. Validation is important to assess the ability of generalization of a model, i.e., the ability that the model can yield a predictive performance on test data similar to that on training data.

We divide the validation strategies into two broad categories, time-insensitive evaluation and time-sensitive evaluation. In time-insensitive evaluation, there is no assumed order in changeset instances while in time-sensitive evaluation there is.

A frequently adopted time-insensitive evaluation strategy is *cross-validation*, suitable for the batch learning JIT-SDP models. The k-fold cross validation [30] is the most common one. Most studies choose $k = 10$, the rest $k = 5$ [24, 25], $k = 2, 3, \ldots, 10$ [87], or unstated [93]. Some studies also adopt the leave-one-out evaluation, essentially, a k-fold cross validation with $k = N_{data}$ where N_{data} is the number of changeset instances in the whole data set (training and testing data sets) [9]. Another time-insensitive evaluation strategy favored by several studies is *out-of-sample bootstrap* [14, 15]. Using the out-of-sample bootstrap, we form a set of bootstrap samples from drawing from the changeset data, use a subset of the bootstrap samples to train the model, and test the model using the unused instances from the bootstrap samples.

An implicit assumption in the batch learning evaluation strategies like the k-fold cross validation, the leave-one-out validation, and the bootstrap validation is that data instances are independent, i.e., software changes are independent to each other when we use these validation strategies for JIT-SDP. However, software changes are inter-dependent. For instance, there is a verification latency between a defect-inducing change and its defect-fixing change, i.e., we can only know and label a change as defect-inducing after we observe a defect-fixing change and trace back to the defect-inducing change [7, 69]. As discussed in Section 3.6.4, several studies examine this type of inter-dependent relationship of software changes, and propose time-sensitive change classifications or evaluate their models in a time-sensitive setting [7, 69, 86].

An additional dimension emerges when we examine CP-JIT-SDP. An evaluation setting is to conduct cross-project validation, such as, the 10-fold cross-project prediction evaluation [86].

Besides, several studies suggest that software projects exhibit temporal variances in their defect characteristics, and recommend evaluations that differentiate a short-period or a long-period prediction in the context of JIT-SDP [24, 47, 73][1].

[1]Kamei et al. [32] use the term short-term prediction for JIT-SDP while long-term prediction Release SDP [32]. So do Pascarella et al. [55]. Nevertheless, we should not confuse the meaning of the short-term and long-term prediction in those works with the short-period and long-period evaluations here.

4 ANALYSIS AND OPEN QUESTIONS

In this section, we analyze existing JIT-SDP studies in order to gain insights into several questions[2], such as, *what is the predictive performance of JIT-SDP models*, and *for what types of software projects are JIT-SDP models more likely to be effective?*

To answer these questions, we adopt the approach by Hall et al. [21] and Hosseini et al. [26] in synthesizing the prior research in JIT-SDP. Section 4.1 discusses the method to select articles for the synthesis. From the selected article, we extract the performance data reported in each of the selected papers and visualize the data. A method of visualization is violin plots, which illustrate the distribution of the predictive performance. A violin plot is like a box plot that reports summary statistics such as minima, maxima, and median, but is also more informative than a box plot since the violin plot also shows the distribution of the data. The form of the violin plots we render in this section is as follows. The left and right contours of a plot exhibit the distribution of the data. The upper and lower limits of the plot show the maximum, and the minimum and the middle bar the average. The upper and lower limits of the filled rectangle inside the plot are the 3rd and 1st quantile while the hollow circle is the median (i.e., the 2nd quantile).

4.1 Study Selection Criteria

First, we establish that each selected study must present predictive performance and include sufficient descriptions of data processing, model building, and evaluation method. Second, we must have sufficient studies to synthesize. As the result, we select from the 67 studies only those with tabular performance results, which yields 48. To ensure that the performance results are comparable, we only select the batch learning studies with cross-validation or alike evaluations. This is because there are only one, two, or three studies in other predictive and evaluation settings and these studies are significantly different and too few to examine in isolation. Since our analysis is to show the distribution of the performance via the violin plots (i.e., Sections 4.2 and 4.3) or to show a trend via a context factor (i.e., Section 4.4), we exclude the studies that do not have sufficient performance results, i.e., report performance results on three or fewer software projects. The final results are 22 articles. We summarize the results of the article selecting process in Table 7. To discern each of the synthesized studies easily, we assign each a short name, as shown in Table 8. To compare the defect proneness and effort-aware predictive performance in Sections 4.2 and 4.3 and to analyze the relationship between the performance results and the context factors in Section 4.4, we rely on the articles that use an identical data set with similar evaluation settings. The most frequently referenced data set is the one availed by Kamei et al. [32] (the Kamei-2012 data set). To indicate those studies using the Kamei-2012 data set, we shade the violin plots for the studies.

Table 7. Selecting Studies for Meta-Analysis

JIT-SDP Studies	Tabular Results	Batch Learning	Cross-Validation Evaluation	Sufficient Performance Results
67	48	42	31	22

4.2 Performance of Defect Prediction

In this section, we examine the within-project predictive performance trends in the JIT-SDP models presented by researchers. First, we identify the evaluation criteria to contract the models. Since defect prediction (including both defect proneness prediction and defect inducing classification) is commonly conceptualized as binary classification, as discussed in Section 3.7.1, F1-score is the

[2]The data and the scripts used in this section is at our Github repository at https://github.com/huichen-cs/jitsdpsurvey.

Table 8. Short Name Assignment of Synthesized Studies and Additional Notes

Short Name	Study	Short Name	Study	Short Name	Study
Chen18(MULTI)	Chen et al. [11]	Duan21	Duan et al. [14]	Fan19	Fan et al. [15]
Huang17(CBS)	Huang et al. [27]	Huang19(CBS+)	Huang et al. [28]	Kamei12(DPLR)	Kamei et al. [32]
Kamei12(EALR)	Kamei et al. [32]	Kang20(Maritime)	Kang et al. [33]	Li20(EATT)	Li et al. [42]
Liu17(CCUM)	Liu et al. [45]	Jiang13(PCC+)	Jiang et al. [29]	Pascarella119	Pascarella et al. [55]
Qiao19(FCNN)	Qiao et al. [58]	Tourani16	Tourani and Adams [74]	Yang15(Deeper)	Yang et al. [83]
Yang17(TLEL)	Yang et al. [82]	Young18(DSL)	Young et al. [87]	Yang16(LT)	Yang et al. [86]
Yan20(Alibaba)	Yan et al. [80]	Yang20(DEJIT)	Yang et al. [85]	Zhu18	Zhu et al. [95]
Zhu20	Zhu et al. [94]				

"Kamei12(DPLR):OS" refers to the result obtained from the Logistic Regression model using the six open-source projects in Kamei et al. [32]. This model servers as a baseline model in several studies. We add "Z-BL"to a short name, e.g., as in "Kamei12(DPLR):OS:Z-BL" to indicate that model Kamei12(DPLR):OS is serving as the baseline model in Zhu et al. [94], and "Y-BL" Yang et al. [83].
Six studies are the data sources for Figure 5. These studies are Kamei et al. [32], Yang et al. [83], Kamei et al. [30], Yang et al. [82], Young et al. [87], and Li et al. [42].

most commonly reported evaluation criterion – 14 studies report F1-score, 12 studies precision and recall (in addition to F1-score), and 6 studies AUC-ROC. There is an additional advantage in using F1-score, i.e., F1-score is the harmonic mean of both precision and recall and therefore provides a more complete picture than either of the two constituent metrics alone. Second, among the 14 studies, there are 6 studies that use the identical data set and features, i.e., the 14 change metrics of 6 open-source projects shared by Kamei et al. [32]. Each of these 6 studies also computes F1-score via k-fold cross-validation, most with $k = 10$. Directly comparing the F1-scores of the 6 studies and those of Kamei et al. [32] is thus the most meaningful; for convenience, in the following, we refer to these changesets and metrics as the Kamei-2012 data set.

Figure 3 shows the violin plots of F1 score in the selected JIT-SDP studies; the filled violin plots represent the studies that use the Kamei-2012 data set. For clarity, we sort the filled violin plots by the median F1-scores reported in those studies. Although 14 studies report F1-scores, we exclude the study by Trautsch et al. [75] from the figure since it only gives a single mean and standard deviation of the 39 projects evaluated, which is not sufficient to produce a violin plot.

Our observations based on Figure 3 are as follows. The study by Zhu et al. [94] appears to be an outlier in that it offers by far the best performance on the Kamei-2012 data set. However, in their study, Zhu et al. [94] show widely improved performance for the baseline as well, indicating that the reported performance may not be comparable. More specifically, as a baseline, Zhu et al. [94] reproduced Kamei et al. [32]'s model using the same data set, parameters and features. Kamei et al. [32]'s model as reproduced by Zhu et al. [94] has a mean F1 score of 0.646 ± 0.044. In contrast, in the original work by Kamei et al. [32], the mean F1 score is 0.452 ± 0.139. The reason for this discrepancy is unclear to us. Several other studies have also reproduced the identical baseline model of Kamei et al. [32], reporting similar F1 scores as those seen in the original work, e.g., Yang et al. [83] report a F-score of the model of Kamei et al. [32]'s as 0.439 ± 0.144.

There is an improvement in the defect prediction as the trajectory of the F1 scores of the models of Kamei et al. [32], Yang et al. [83], Yang et al. [82], and Young et al. [87] demonstrate in Figure 3. However, the improvement appears to have plateaued with smaller improvements with each new study. At present, it appears that the ensemble models by Yang et al. [82] and Young et al. [87] offer the best F1-score performance, both of these models leverage on model bagging and stacking (Section 3.6).

Another clear observation from Figure 3 is that the F1-score varies greatly across different software projects, as shown in the unfilled violin plots, which makes the direct comparison of the models' predicative performance across different projects difficult.

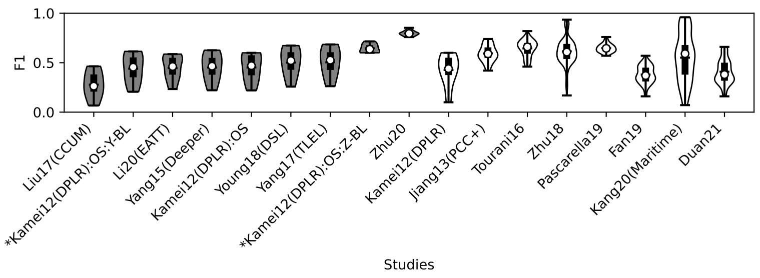


Fig. 3. Comparison of F1 scores of defect prediction where filled violin plots are those studies that use the Kamei-2012 data set while the unfilled differ by evaluation data sets. The models starting with an asterisk are reproductions of the original model conducted by other researchers.

Findings. Our synthesis of the reported performance in the prior JIT-SDP studies indicates that defect prediction models DSL and TLEL appear to outperform the others. Both DSL and TLEL are ensemble models. Ensemble model can learn different and complementary patterns from data, offering an advantage over standalone learning algorithms. The Random Forest classifier is also an ensemble model based on decision trees. Kamei et al. [30] and Fukushima et al. [18] favor the Random Forest classifier in their studies, and Duan et al. [14] and Fan et al. [15] demonstrate that it yields a superior predictive performance over Logistic Regression and Naive Bayes.

Based on the trends in the current set of studies of JIT-SDP, ensemble models perform the best at predicting defectiveness of software changes. However, improvements in predictive performance seem to have plateaued in recent approaches.

Open Questions. How can we continue to improve defect prediction performance? There have been several recent directions that may markedly improve JIT-DSP. First, models based on deep learning, e.g., convolutional neural networks, show promising results [24, 25]. Since there is a significant variation in JIT-SDP predictive performance among different software projects, we have yet to observe how deep learning models would perform on a broader variety of projects. Second, researchers are investigating different types of software change metrics and additional sources of data of software changes to help improve predictive performance [40, 74, 75].

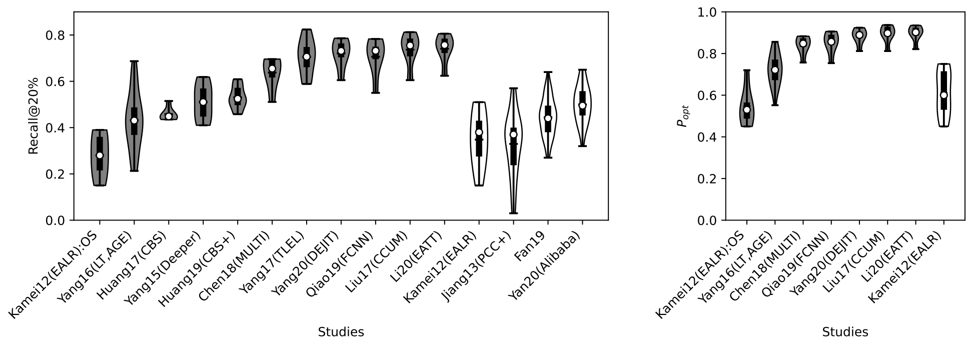
4.3 Performance of Effort-Aware Prediction

Next, we investigate the performance of within-project effort-aware prediction. The most frequently reported evaluation criteria for effort aware JIT-SDP are P_{opt} and $Recall@20\%$ – 6 studies report P_{opt} while 11 $Recall@20\%$. The former is an effort-aware threshold free evaluation criterion while the later is computed at the threshold of 20% of the review effort.

Figure 4 shows violin plots of $Recall@20\%$ and P_{opt} for the selected studies. Again, several studies use the Kamei-2012 data set for evaluation, which, as before, we highlight with filled violin plots. Our observations are as follows.

Similar to defect proneness prediction, there is a significant variance of the effort-aware prediction performance as illustrated by the violin plots. However, different from it, the filled violin plots show that there is a clearer improvement trend for effort-aware prediction performance. For instance, the EATT model by Li et al. [42] has the highest *Recall*@20% and P_{opt} and CCUM by Liu et al. [45] is a close second.

The relationship between the performance of a model for defect prediction and effort-aware prediction can be inversely proportional, i.e., when one gets better, the other gets worse [86]. In other words, when examining both Figure 3 and 4 together, we observe that CCUM and EATT have poorer F1-scores for defect prediction than many of the other defect prediction models, however, their *Recall*@20% and P_{opot} are the best. This is likely due to the size distribution (or the review effort distribution) among the software changes. In particular, the change defect density tends to be higher among the smaller changes [86].



(a) *Recall*@20% (b) P_{opt}

Fig. 4. Comparison of *Recall*@20% and P_{opt} of effort-aware prediction. As before, the filled violin plots are those studies that use the Kamei-2012 data set while the unfilled differ by evaluation data sets.

Findings. The three models, TLEL, CCUM, and EATT, appear to be the best performing effort-aware models. CCUM is a simple searching-based unsupervised model that does not rely on any learning algorithms. Yan et al. [80] show that the CCUM is the model that has the best *Recall*@20% among 14 proprietary projects from the Alibaba company.

The models TLEL and EATT are ensemble models that can take advantage of either the diversity of patterns in data or the diversity of learning capability. TLEL is an ensemble model that explores the diversity in data while EATT the diversity of learning capability. Since, as observed in Section 4.2, ensemble models also have superior defect prediction, it seems likely that supervised ensemble models like TLEL are likely to be the most versatile ones if there is a need to solve both the effort-aware and non-effort-aware JIT-SDP problems simultaneously.

Ensemble models are also the best effort-aware models for JIT-SDP. Searching-based unsupervised models yield effort-aware predictive performance similar to the ensemble models, however, they perform much worse on defect predictions than ensembles.

Open Questions. One of the important limitation in prior research in JIT-SDP is the lack of a shared benchmark. While several studies use the Kamei-2012 data set, considering the difference in the remaining (unfilled) violin plots, it is unclear that the conclusions drawn from the models using the Kamei-2012 evaluation data set are applicable to the other studies with different evaluation data sets. This indicates that there may be a need for a standard data set with greater variety of projects.

4.4 Context Factors of Software Project and Changeset Data

Hall et al. [21] indicate the importance of software project context factors in understanding SDP model performance. Context factors presented in prior studies include: the number of commits, the ratio of the defective changes, average change size, average number of developers per modified file, and several others [32]. Table 5 shows several of the context factors for the 6 open source software projects from Kamei et al. [32].

To understand how software project context factors impact the predictive performance of JIT-SDP, we combine the results from JIT-SDP studies that evaluate their models with the Kamei-2012 data set. In Figure 5 we contrast the combined F1 scores of these JIT-SDP studies to 4 commonly mentioned context factors.

In Figures 5(a) and 5(b) we observe that the F1-scores correlate with the number of changes and the ratio of defective changes. Figure 5(a) shows a trend of F1-scores getting smaller when the number of commits increases while in Figure 5(b), F1-scores getting bigger when the ratio of defect-inducing changes becomes greater. The number of changes can be considered as a measure for project maturity for JIT-SDP, and the ratio of defect-inducing changes as a measure of project quality. In the remaining plots, Figure 5(c) and Figure 5(d) , do not show a relationship between the F1-scores with the two context factors, the average change size and the average number of developers per modified file in commits. These observations are consistent with the Spearman correlation coefficients between F1 and the context factors as shown in Table 9.

Next, we try to confirm the above observations from the other individual studies that use different software projects, i.e., not only the Kamei-2012 data set. Needless to say that these studies differ greatly from each other, and therefore, we do not aggregate them but plot each individual study as a line showing the average F1-scores versus the defect ratios of software projects used in the study.

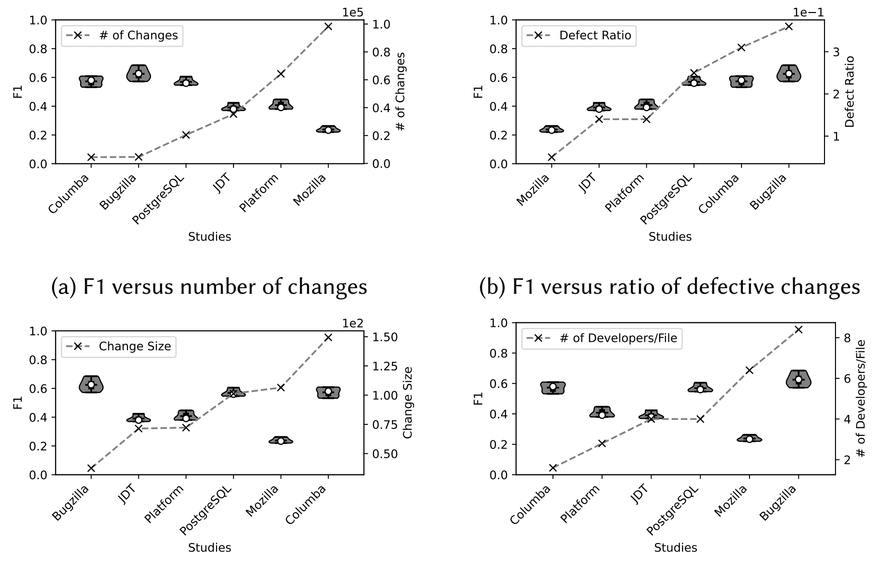
We present these results in Figure 6 and Table 10. The results indicate that half of these studies appear to confirm the observation in the above, namely, the F1 scores in Tourani and Adams [74], in Fan et al. [15], and in Duan et al. [14] increase as the defective ratio rises and the F1 scores are also strongly correlate with the defective ratio. The study by Zhu et al. [95] only weakly confirm the observation due to a smaller correlation and a large p-value. The studies by Jiang et al. [29] and Pascarella et al. [55] are unclear or do not support this conclusion at all.

Table 9. Spearman Correlations between F1 and Context Factors

F1 vs. # of Changes		F1 vs. Ratio of Defective Changes		F1 vs. Change Size		F1 vs. # of Developers Per File	
Coefficient	P-value	Coefficient	P-value	Coefficient	P-value	Coefficient	P-value
-0.886	0.019	0.986	0.000	-0.200	0.704	-0.029	0.957

Findings. The most important finding in this analysis is the observation that the predictive performance appears to correlate with the ratio of defect-inducing changes in the software project.

In most studies, the predictive performance of JIT-SDP positively correlates with the defect ratio of changes in the software project, i.e., projects with more defects are easier for JIT-SDP to predict defect-inducing changes.



(a) F1 versus number of changes

(b) F1 versus ratio of defective changes

(c) F1 versus change size

(d) F1 versus number of developers per file

Fig. 5. F1 scores versus context factors. Sources of F1-scores are in Table 8.

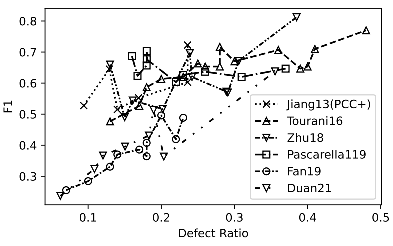


Fig. 6. F1 versus ratio of defective changes of selected studies

Table 10. Spearman Correlations of F1 and Ratio of Defective Changes

Study	Coefficient	p-value
Jian13(PCC+)	0.49	0.321
Tourani16	0.80	0.000
Zhu18	0.53	0.117
Pascarella119	-0.42	0.229
Fan19	0.91	0.000
Duan21	0.76	0.028

Open Questions. There are several context factors that have not been reported or explored in JIT-SDP studies. For instance, for which application domain do we obtain better predicative performance, do programming languages matter for JIT-SDP, and does project maturity impact performance? When assembling the prior studies to conduct this meta analysis, we notice that although several studies evaluate their models with multiple software projects, many potential context factors about the software projects are not reported.

5 IDEAS AND CONSIDERATIONS FOR FUTURE RESEARCH

Based on the survey and analysis of JIT-SDP techniques presented up to this point, in this section, we identify a set of novel research directions and a few key considerations for researchers that we believe to be important in the context of JIT-SDP.

5.1 JIT-SDP Application Domain

Several studies have emphasized the importance of application domains of defect prediction models (e.g., is the model used for prioritizing maintenance effort, cost prediction, resource allocation, and reducing technical debt). For instance, Zeller et al. [88] argue that we should ground SDP in practice, consider what developers think about the models and the results, and how the models and the results help the developers in their daily work. Wan et al. [76] carry out a study to investigate what practitioners think, behave, and expect when it comes to defect prediction. As a result, they detail a few recommendations. More investigations along this direction would help researches in SDP, including both Release SDP and JIT-SDP, to make SDP easier to adopt, i.e., to *improve the quality of developer experience in using SDP*.

JIT-SDP is not a debugging technique, as it does not reveal the type of defects, the symptom of the defects, and the condition under which the defects occur. Rather, JIT-SDP is a risk assessment technique, as it reports a likelihood that a defect may occur. In order for this risk assessment to provide developers with high quality of user experience, we need to have a reasonably high predictive performance, however, what exactly the level of performance is required to make a specific practical application of JIT-SDP work? It requires a careful analysis for each application domain.

To further understand the importance of the application domain of a JIT-SDP model, we examine one application scenario: QA resource allocation. Let us consider that we build a JIT-SDP model to allocate QA resources for a software project that has N new software changes. The model predicts $N_{TP+FP} = N_{TP} + N_{FP}$ changes as defect prone, where N_{TP} is the number of true positives (a predicted defective change indeed has a defect) and N_{FP} the number of false positives (no defects found in a predicted defective change). From prior model validation, we know the precision c_{PR}, which we assume holds for the new changes. Thus, we can anticipate that $N_{TP} = c_{PR}N_{TP+FP}$ changes are actually defective since $c_{PR} = N_{TP}/N_{TP+FP}$. An estimate of the required QA effort is $E_{QA} = e_{review}N_{TP+FP} + e_{fixing}N_{TP} = e_{review}N_{TP+FP} + e_{fixing}c_{PR}N_{TP+FP}$, where e_{review} is the effort to review a change and e_{fixing} is the effort to fix the actual defect induced by the change. In this scenario, both of the accuracy of c_{PR} and the magnitude of N_{TP+FP} matter.

Let $E_{fixing} = e_{fixing}c_{PR}N_{TP+FP}$. Due to the nature of randomness, the actual defects vary from the anticipated $c_{PR}N_{TP+FP}$. Denote the difference as a ratio δ that the user can determine. If the actual outcome is $E_{fixing}|_{-\delta} = e_{fixing}(1-\delta)c_{PR}N_{TP+FP}$, we may over-allocate QA resources; if the actual outcome is $E_{fixing}|_{\delta} = e_{fixing}(1+\delta)c_{PR}N_{TP+FP}$, we may under-allocate QA resources. What is the probability that we allocate the appropriate amount of resources?

To answer this question, we assume that it is an independent event that each predicted defect prone change is actually defective. We could model an event that a defect prone change C is an actual defective one following the Bernoulli distribution with probability $p = c_{PR}$, i.e., $P(C = \text{defective}) = P(C = 1) = p$ and $P(C = \text{clean}) = P(C = 0) = 1 - p$. For convenience, let $M = N_{TP+FP}$, $S = C_1 + C_2 + \ldots + C_M = \sum_{i=1}^{M} C_i$ where we denote $C_i \in \{0, 1\}$ as the random variable corresponding to the label of a change, $S_L = (1-\delta)pM$, and $S_H = (1+\delta)pM$. Since C follows the Bernoulli distribution, the variance of random variable C is $\text{Var}[C] = \sigma^2 = p(1-p)$ and the mean is $\text{E}[C] = p$. It follows that $\text{E}[S] = pM$ and $\text{Var}[S] = \sigma_S^2 = p(1-p)M$. Then, according to the Central Limit Theorem, we have:

$$P(S_L \leq S \leq S_H) = P\left(-\delta\left(\frac{pM}{1-p}\right)^{\frac{1}{2}} \leq z \leq \delta\left(\frac{pM}{1-p}\right)^{\frac{1}{2}}\right) \tag{1}$$

where $z = (S - \mathrm{E}[S])/\sigma_S$ follows the Normal distribution $N(0, 1)$. Since $P(S_L \leq S \leq S_H)$ is the probability at which the project has the satisfactory resource allocation for QA, both $p = c_{PR}$ and $M = N_{TP+FP}$ matter to the project where p is ultimately determined by the capability of the defect prediction algorithm and M the size of the project. Although in the above analysis we fix S_L and S_H, we can influence these two in order to have a larger $P(S_L \leq S \leq S_H)$, signifying a greater likelihood we allocate appropriate amount of QA resources. From equation (1), the greater M (i.e., a larger project) and p (i.e., a better JIT-SDP algorithm) are, the larger the probability at which we will successfully allocate QA resources; however, we can compensate a small p with a large M and vice versa, which implies that we can apply JIT-SDP for QA resources allocation equally well for a larger project with a weaker JIT-SDP algorithm and vice versa.

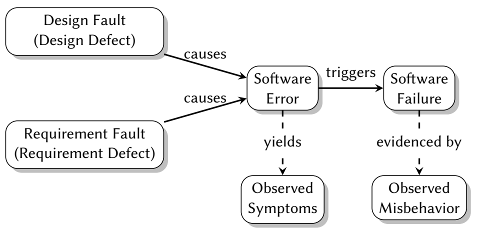
The above analysis shows that with a JIT-SDP model with a moderate precision, we can still successfully allocate QA resources for a large collections of software changes. More broadly it demonstrates that each individual application domain (e.g., prioritizing maintenance tasks, planning activities to reduce technical debt) has its own specific demands from a JIT-SDP model. Finally, what predictive performance we need from a JIT-SDP model is dependent on its planned application.

5.2 Relationship of Defects to Software Use

Software defects are often the root case of a software failure. Failures negatively impact the reliability and thus the quality of the software. Since JIT-SDP techniques are primarily targeted towards improving software quality, we must pay attention to the relationship of defects (via failures) on software quality, i.e., the impact of the specific defects on the actual reliability of the software. To clarify these concepts, let us examine the relationship among defect, error, and failure.

We illustrate the relationship of defects, failures, and errors in Figure 7(a). Defects cause errors where an error is the deviation of the system state from the desired one [60, 65]. However, an error may or may not lead to a failure. Whether a defect manifests itself as a software failure depends on how the user executes the code that has the defect, i.e., depends on the current system state and the specific input. To understand this relationship, following the spirit of the example given by Salfner et al. [65], we consider the code snippet in Listing 7(b), which is a C code snippet from the SEI CERT C Coding Standard [66]. This example is not compliant with the C Coding Standard. The function call to `malloc()` at line 5 allocates an object referenced by pointer `text_buffer`. It fails to free the object before `text_buffer` reaches the end of its lifetime. Therefore, the object will remain in the heap of the process' address space until the process terminates. This is a defect caused by missing the `free(text_buffer)` statement that should have been at line 9. This defect causes an error, i.e., an unreachable address block in the process's address space after the control of the process returns to the caller invoking the function `f()`. If the process invokes the function `f()` repeatedly, a failure may occur, i.e., the system may become slower (an observable symptom) due to excessive paging or the process may crash due to a failed memory allocation as the process exhausts its free address space in the heap. However, whether this failure occurs depends on many factors, e.g., the number of invocations to the function `f()`, the availability of free heap address space, and the available physical memory. For instance, if the process is short-lived with a limited invocation of the function `f()`, there wouldn't be any noticeable misbehavior in the system, in particular, after the operating system frees the memory allocated to the process as the process terminates.

Since defects are root cause of failures, reducing defects, in turn, reduces failures. It is not always simple to apply this principle to QA. Defects and failures have a non-trivial relationship as illustrated in the above example. Fenton and Neil observe that failures caused by defects have a significant variance in the mean time to failure [16]. Clearly, *it is of little practical value to examine*



(a) Relations of software defects (faults), errors, and failures.

```c
#include <stdlib.h>
enum { BUFFER_SIZE = 32 };

int f(void) {
    char *text_buffer = (char *)malloc(BUFFER_SIZE);
    if (text_buffer == NULL) {
        return -1;
    }

    return 0;
}
```

(b) A code snippet showing a memory allocation defect [66]

Fig. 7. Relationship of defect, error, and failure and an example of defective code

defects that almost never manifest themselves as software failures, and it is of great urgency to remove a defect that causes software failure daily, even at a significant QA effort.

Therefore, we should consider JIT-SDP defects, not in isolation, but, combine with common patterns of software usage. For instance, Tian et al. [73] have recently begun to investigate JIT-SDP as it relates to software reliability. More studies like this may be an important direction in JIT-SDP. For instance, in the future, researchers can study how to link software changes with different types of defects, or how to associate different defects with software usage patterns.

5.3 Quantifying Effort in JIT-SDP

Fenton and Neil [16] divide software defects into two broad categories, requirement defects, i.e., the defects in software requirements, and design defects, i.e., the defects in design specification, implementation, and maintenance. Requirement defects lead to validation failures while design defects verification failures. As, generally, requirement defects are more costly to fix than design defects, we should reexamine how to quantify effort for effort-aware JIT-SDP.

In effort-aware JIT-SDP, the quality assurance effort is measured as the lines of code reviewed [32]. *Future JIT-SDP may consider alternative quality assurance effort measures*, such as, by differentiating the efforts to address validation failures and those to address verification failures. For instance, researchers can attempt to use traceability analysis approaches that link "high-level" artifacts, e.g., requirements, to "low-level", e.g., software code [50]. As another alternative, several studies examine software faults and failure types and their occurrences in different phases of software development life cycle processes [22].

5.4 Uncertainty Quantification in JIT-SDP

JIT-SDP is affected by the uncertain quality of the input data used to build the model [31]. In the context of Release SDP, Kamei and Shihab [31] argue that "the distribution of metrics can vary among releases" and the training and testing data sets may not have similar distributions, i.e., the data used to train the SDP model no longer reflects the current project data. This type of uncertainty that Kamei and Shihab [31] describe is also found in JIT-SDP. For instance, Bennin et al. [4] formulate the significant change in statistical distribution over time found in software change history data as a "concept drift" and investigate its impact on JIT-SDP. Several studies also address the concept drift problem in the context of online learning in JIT-SDP [7, 68].

Another type of uncertainty in JIT-SDP has recently come into attention in several studies. As discussed in Section 3.4, JIT-SDP requires the SZZ algorithm to determine whether a software change is defect-inducing given a defect-fixing change. Recent studies of SZZ, e.g., Fan et al. [15] and

Rosa et al. [64], indicate that the algorithm is not always reliable, which can introduce downstream errors in JIT-SDP.

A direction of research we propose is to quantify these different types of uncertainties in JIT-SDP and use them to convey the magnitude of uncertainty to JIT-SDP users. In this way, the users can obtain an accurate "error bar" for each prediction made by a JIT-SDP model that can serve to enhance how the model is used and which recommendation is taken.

5.5 Explainable and Actionable Prediction

Tantithamthavorn and Hassan [70] point out that the value of defect prediction models is not only in predicting which software artifacts are more defect prone than the others, but also, perhaps even more importantly, in deriving empirical theories and in providing a guidance on operational decisions about software quality. Zeller et al. [88] express a similar opinion and argue that "an empirical finding is the more valuable the more actionable it is" and the examples of actions include how a developing team changes their future practice and what the risk of the changes is. Importantly, these actionable insights once confirmed by future practice and the practice can help unearth the systemic factors that drive software quality. As a full circle, these actionable insights in turn can help us design and build superior defect prediction models.

How do we make our defect prediction results actionable? It begins with making the prediction results explainable [34]. One approach is to link the prediction results with context factors of the data sets. Hall et al. [21] note that without the context factors of the data sets it becomes difficult to understand the implications of SDP models and results. They list several such factors, e.g., project age, maturity, programming language, and application domain. Recent JIT-SDP studies have included some context factors, however, software projects have diverse context factors and it remains unclear what context factors best characterize a software project [90]. Perhaps, relevant project context factors can vary from project to project and from application domain to application domain. Another approach is to link the prediction results with software metrics. Recent works in JIT-SDP also indicate that a variety of software metrics from a variety of data sources in addition to software changes can help boost defect predictive performance [40, 74, 75]. Therefore, *there is a need to continue exploring project context factors and software metrics, and their relationship with defection prediction results.*

An important and related question is, *how can we make a complex JIT-SDP model explainable?* Recently, deep learning models have found their success in both of Release SDP and JIT-SDP [10, 24, 25]. These models offer superior predictive performance. In particular, these models often take an "end-to-end" approach, e.g., in the context of JIT-SDP, taking directly software changes and commit messages as the input and predict defect prone changes. Xie et al. [77] provide an overview about building explainable deep learning models. Leveraging this research and alike, future research directions may include the design of explainable deep neural network models for JIT-SDP.

Finally, we would like to emphasize what Zeller et al. [88] have cautioned us, i.e., "being non-actionable may still be better than suggesting the wrong actions" and argue that we should be careful when using complexity metrics or overly simple features in defect prediction since we may more easily arrive at suggesting wrong actions. Similar in spirit, we should also caution about complex models. Is the superior predictive power of a complex model the result of overfitting or the result of a superior model underpinned by the systemic factors driving software quality?

5.6 Beyond Predicting Defects

SDP models estimate *defect* risk or the probability that a software artifact or a software change is defect inducing. Effort-aware models are a direct extension of defect proneness prediction. Tantithamthavorn and Hassan indicate that there might be some other prediction dependent variables worthy of investigating [70]. For instance, they argue that change risk is worthy of modeling where they define the change risk as any changes that might lead to a schedule slippage [70]. Also, in Release SDP, some studies investigate the problem of predicting defect severity and model it as a multi-class classification problem [21]; however, contrasted to Release SDP, there not yet been a JIT-SDP study whose dependent variable is defect severity.

A broader question for JIT-SDP researchers is as follows. *Are there any other defect related dependent variables that we should investigate?* Perhaps, this requires us to think beyond statistical modeling and machine learning and to interrogate more closely the key factors influencing the software development process.

6 CONCLUSION

This article presents a systematic survey of Just In Time Software Defect Prediction (JIT-SDP), a sub-area of Software Defect Prediction (SDP). SDP is a long standing research topic within Software Engineering with a large corpus of research. The two branches of SDP, Release SDP and JIT-SDP, are complementary to each other. Wan et al. [76] define Release SDP as predicting defects occurring in software artifacts, i.e., files, packages, and subsystems, that can help developers gain insights into overall software quality, while JIT-SDP predicts the likelihood of defect occurrences in software changes, a finer granularity and can aid practitioners in quickly locating defects. Both Release SDP and JIT-SDP are primarily risk assessment techniques and not debugging aids.

In our literature survey, we discuss the major achievement of JIT-SDP research in context of the entire workflow of JIT-SDP. Different from Release SDP where the input is software snapshots, the input to JIT-SDP is the software's change history. As such, JIT-SDP studies frequently leverage software change metrics as input features and, as output, predict defect proneness, change defect density, or scores for effort-aware defect prediction. From the perspective of modeling, JIT-SDP has at least one distinct characteristic different from Release SDP, i.e., software change data is intertwined with a dual arrival process, the arrivals of software changes and the arrivals of software change labels (i.e., defect-inducing or clean). The former arrival process is the result of implementing the software; while the later the result of validating or verifying the software. Emerging studies have noted the impact of this distinct characteristic on JIT-SDP models and evaluations, and have begun to investigate online learning, verification latency, and time-sensitive evaluation. JIT-SDP studies have experimented with a broad range of preprocessing techniques and machine learning algorithms. As Box suggests, "all models are wrong, some are useful [5]", some JIT-SDP studies focus on usefulness of models for effort-aware defect prediction in software changes, and propose simple unsupervised models. Some of these unsupervised models are "searching-based" without using any machine learning techniques and produce effort-aware prediction, sometimes comparable to sophisticated supervised machine learning algorithms.

With the aim at understanding the state-of-the-art of the JIT-SDP models' performance, we conduct a meta-analysis of existing studies. In conducting the meta-analysis, we encounter the same difficulties as Hall et al. [21] and Hosseini et al. [26], such as, insufficient methodological information, incomplete evaluation criteria, and lack of consistent benchmark data sets. Despite these difficulties, our meta-analysis indicates that (1) there has been an improvement of predictive performance over early JIT-SDP studies although the improvement appears to have plateaued; (2) ensemble models appear to be the best performing models, and (3) the performance of JIT-SDP models appears to correlative positively with the ratio of defective changes in a software project.

Last, but not least, considering the findings and the insights from prior research, we point out several research directions and considerations for future research. These directions and considerations include, among others, consideration for the application domain where JIT-SDP is to be applied, differentiating defects based on software use, explainable and actionable predictions, and quantifying uncertainty in JIT-SDP.

REFERENCES

[1] Pasquale Ardimento, Lerina Aversano, Mario Luca Bernardi, Marta Cimitile, and Martina Iammarino. 2021. Just-in-time software defect prediction using deep temporal convolutional networks. *Neural Computing and Applications* (2021), 1–21.

[2] Lerina Aversano, Luigi Cerulo, and Concettina Del Grosso. 2007. Learning from bug-introducing changes to prevent fault prone code. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*. 19–26.

[3] Jacob G Barnett, Charles K Gathuru, Luke S Soldano, and Shane McIntosh. 2016. The relationship between commit message detail and defect proneness in java projects on github. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 496–499.

[4] Kwabena E Bennin, Nauman bin Ali, Jürgen Börstler, and Xiao Yu. 2020. Revisiting the impact of concept drift on just-in-time quality assurance. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 53–59.

[5] George EP Box. 1979. Robustness in the strategy of scientific model building. In *Robustness in statistics*. Elsevier, 201–236.

[6] Leo Breiman. 2001. Statistical modeling: The two cultures (with comments and a rejoinder by the author). *Statistical science* 16, 3 (2001), 199–231.

[7] George G Cabral, Leandro L Minku, Emad Shihab, and Suhaib Mujahid. 2019. Class imbalance evolution and verification latency in just-in-time software defect prediction. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 666–676.

[8] Cagatay Catal. 2011. Software fault prediction: A literature review and current trends. *Expert systems with applications* 38, 4 (2011), 4626–4636.

[9] Gemma Catolino, Dario Di Nucci, and Filomena Ferrucci. 2019. Cross-project just-in-time bug prediction for mobile apps: an empirical assessment. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 99–110.

[10] Jinyin Chen, Keke Hu, Yue Yu, Zhuangzhi Chen, Qi Xuan, Yi Liu, and Vladimir Filkov. 2020. Software visualization and deep transfer learning for effective software defect prediction. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 578–589.

[11] Xiang Chen, Yingquan Zhao, Qiuping Wang, and Zhidan Yuan. 2018. MULTI: Multi-objective effort-aware just-in-time software defect prediction. *Information and Software Technology* 93 (2018), 1–13.

[12] Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E Hassan. 2016. A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering* 43, 7 (2016), 641–657.

[13] Honghui Du, Leandro L Minku, and Huiyu Zhou. 2019. Multi-source transfer learning for non-stationary environments. In *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.

[14] Ruifeng Duan, Haitao Xu, Yuanrui Fan, and Meng Yan. 2021. The impact of duplicate changes on just-in-time defect prediction. *IEEE Transactions on Reliability* (2021).

[15] Yuanrui Fan, Xin Xia, Daniel Alencar da Costa, David Lo, Ahmed E Hassan, and Shanping Li. 2019. The impact of changes mislabeled by SZZ on just-in-time defect prediction. *IEEE transactions on software engineering* (2019).

[16] Norman E Fenton and Martin Neil. 1999. A critique of software defect prediction models. *IEEE Transactions on software engineering* 25, 5 (1999), 675–689.

[17] Wei Fu and Tim Menzies. 2017. Revisiting unsupervised learning for defect prediction. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 72–83.

[18] Takafumi Fukushima, Yasutaka Kamei, Shane McIntosh, Kazuhiro Yamashita, and Naoyasu Ubayashi. 2014. An empirical study of just-in-time defect prediction using cross-project models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. 172–181.

[19] Jiri Gesi, Jiawei Li, and Iftekhar Ahmed. 2021. An Empirical Examination of the Impact of Bias on Just-in-time Defect Prediction. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–12.

[20] Trisha Greenhalgh and Richard Peacock. 2005. Effectiveness and efficiency of search methods in systematic reviews of complex evidence: audit of primary sources. *Bmj* 331, 7524 (2005), 1064–1065.

[21] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. 2011. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering* 38, 6 (2011), 1276–1304.

[22] Maggie Hamill and Katerina Goseva-Popstojanova. 2009. Common trends in software fault and failure data. *IEEE Transactions on Software Engineering* 35, 4 (2009), 484–496.

[23] Kim Herzig, Sascha Just, and Andreas Zeller. 2016. The impact of tangled code changes on defect prediction models. *Empirical Software Engineering* 21, 2 (2016), 303–336.

[24] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. 2019. DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 34–45.

[25] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. CC2Vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 518–529.

[26] Seyedrebvar Hosseini, Burak Turhan, and Dimuthu Gunarathna. 2017. A systematic literature review and meta-analysis on cross project defect prediction. *IEEE Transactions on Software Engineering* 45, 2 (2017), 111–147.

[27] Qiao Huang, Xin Xia, and David Lo. 2017. Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 159–170.

[28] Qiao Huang, Xin Xia, and David Lo. 2019. Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction. *Empirical Software Engineering* 24, 5 (2019), 2823–2862.

[29] Tian Jiang, Lin Tan, and Sunghun Kim. 2013. Personalized defect prediction. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Ieee, 279–289.

[30] Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E Hassan. 2016. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering* 21, 5 (2016), 2072–2106.

[31] Yasutaka Kamei and Emad Shihab. 2016. Defect prediction: Accomplishments and future challenges. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, Vol. 5. IEEE, 33–45.

[32] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2012. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (2012), 757–773.

[33] Jonggu Kang, Duksan Ryu, and Jongmoon Baik. 2020. Predicting just-in-time software defects to reduce post-release quality costs in the maritime industry. *Software: Practice and Experience* (2020).

[34] Chaiyakarn Khanan, Worawit Luewichana, Krissakorn Pruktharathikoon, Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, Morakot Choetkiertikul, Chaiyong Ragkhitwetsagul, and Thanwadee Sunetnanta. 2020. JITBot: An explainable just-in-time defect prediction bot. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1336–1339.

[35] Sunghun Kim, E James Whitehead, and Yi Zhang. 2008. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering* 34, 2 (2008), 181–196.

[36] Sunghun Kim, Thomas Zimmermann, Kai Pan, E James Jr, et al. 2006. Automatic identification of bug-introducing changes. In *21st IEEE/ACM international conference on automated software engineering (ASE'06)*. IEEE, 81–90.

[37] Barbara Kitchenham and Stuart Charters. 2007. *Guidelines for performing systematic literature reviews in software engineering*. Technical Report EBSE-2007-01. School of Computer Science and Mathematics, Keele University.

[38] Barbara Kitchenham, O. Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. 2009. Systematic literature reviews in software engineering – A systematic literature review. *Information and Software Technology* 51, 1 (2009), 7 – 15. https://doi.org/10.1016/j.infsof.2008.09.009 Special Section - Most Cited Articles in 2002 and Regular Research Papers.

[39] Barbara Ann Kitchenham, David Budgen, and Pearl Brereton. 2015. *Evidence-based software engineering and systematic reviews*. Vol. 4. CRC press.

[40] Masanari Kondo, Daniel M German, Osamu Mizuno, and Eun-Hye Choi. 2020. The impact of context metrics on just-in-time defect prediction. *Empirical Software Engineering* 25, 1 (2020), 890–939.

[41] Ning Li, Martin Shepperd, and Yuchen Guo. 2020. A systematic review of unsupervised learning techniques for software defect prediction. *Information and Software Technology* (2020), 106287.

[42] Weiwei Li, Wenzhou Zhang, Xiuyi Jia, and Zhiqiu Huang. 2020. Effort-aware semi-supervised just-in-time defect prediction. *Information and Software Technology* 126 (2020), 106364.

[43] Zhiqiang Li, Xiao-Yuan Jing, and Xiaoke Zhu. 2018. Progress on approaches to software defect prediction. *IET Software* 12, 3 (2018), 161–175.

[44] Dayi Lin, Chakkrit Tantithamthavorn, and Ahmed E Hassan. 2021. The Impact of Data Merging on the Interpretation of Cross-Project Just-In-Time Defect Models. *IEEE Transactions on Software Engineering* (2021).

[45] Jinping Liu, Yuming Zhou, Yibiao Yang, Hongmin Lu, and Baowen Xu. 2017. Code churn: A neglected metric in effort-aware just-in-time defect prediction. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 11–19.

[46] Ruchika Malhotra. 2015. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing* 27 (2015), 504–518.

[47] Shane McIntosh and Yasutaka Kamei. 2017. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. *IEEE Transactions on Software Engineering* 44, 5 (2017), 412–428.

[48] Tim Menzies, Andrew Butcher, David Cok, Andrian Marcus, Lucas Layman, Forrest Shull, Burak Turhan, and Thomas Zimmermann. 2012. Local versus global lessons for defect prediction and effort estimation. *IEEE Transactions on software engineering* 39, 6 (2012), 822–834.

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

[50] Kevin Moran, David N Palacio, Carlos Bernal-Cárdenas, Daniel McCrystal, Denys Poshyvanyk, Chris Shenefiel, and Jeff Johnson. 2020. Improving the effectiveness of traceability link recovery using hierarchical bayesian networks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 873–885.

[51] Keita Mori and Osamu Mizuno. 2015. An Implementation of Just-in-Time Fault-Prone Prediction Technique Using Text Classifier. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, Vol. 3. IEEE, 609–612.

[52] Edmilson Campos Neto, Daniel Alencar da Costa, and Uirá Kulesza. 2018. The impact of refactoring changes on the SZZ algorithm: An empirical study. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 380–390.

[53] Yusuf Sulistyo Nugroho, Hideaki Hata, and Kenichi Matsumoto. 2020. How different are different diff algorithms in Git? *Empirical Software Engineering* 25, 1 (2020), 790–823.

[54] Matheus Paixao, Jens Krinke, Donggyun Han, and Mark Harman. 2018. CROP: Linking code reviews to source code changes. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 46–49.

[55] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. 2019. Fine-grained just-in-time defect prediction. *Journal of Systems and Software* 150 (2019), 22–36.

[56] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[57] Chanathip Pornprasit and Chakkrit Tantithamthavorn. 2021. JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction. *arXiv preprint arXiv:2103.07068* (2021).

[58] Lei Qiao and Yan Wang. 2019. Effort-aware and just-in-time defect prediction with neural network. *PloS one* 14, 2 (2019), e0211359.

[59] Sophia Quach, Maxime Lamothe, Bram Adams, Yasutaka Kamei, and Weiyi Shang. 2021. Evaluating the impact of falsely detected performance bug-inducing changes in JIT models. *Empirical Software Engineering* 26, 5 (2021), 1–32.

[60] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. 2013. Software fault prediction metrics: A systematic literature review. *Information and software technology* 55, 8 (2013), 1397–1418.

[61] Foyzur Rahman, Sameer Khatri, Earl T Barr, and Premkumar Devanbu. 2014. Comparing static bug finders and statistical prediction. In *Proceedings of the 36th International Conference on Software Engineering*. 424–434.

[62] Zhi-Yong Ran and Bao-Gang Hu. 2017. Parameter identifiability in statistical machine learning: a review. *Neural Computation* 29, 5 (2017), 1151–1203.

[63] Gema Rodriguez-Perez, Meiyappan Nagappan, and Gregorio Robles. 2020. Watch out for extrinsic bugs! A case study of their impact in just-in-time bug prediction models on the OpenStack project. *IEEE Transactions on Software Engineering* (2020).

[64] Giovanni Rosa, Luca Pascarella, Simone Scalabrino, Rosalia Tufano, Gabriele Bavota, Michele Lanza, and Rocco Oliveto. 2021. Evaluating SZZ Implementations Through a Developer-informed Oracle. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE.

[65] Felix Salfner, Maren Lenk, and Miroslaw Malek. 2010. A survey of online failure prediction methods. *ACM Computing Surveys (CSUR)* 42, 3 (2010), 1–42.

[66] SEI Authors. 2016. SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems.

[67] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? *ACM sigsoft software engineering notes* 30, 4 (2005), 1–5.

[68] Sadia Tabassum, Leandro L Minku, Danyi Feng, George G Cabral, and Liyan Song. 2020. An investigation of cross-project learning in online just-in-time software defect prediction. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 554–565.

[69] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. 2015. Online defect prediction for imbalanced data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 99–108.

[70] Chakkrit Tantithamthavorn and Ahmed E Hassan. 2018. An experience report on defect modelling in practice: Pitfalls and challenges. In *Proceedings of the 40th International conference on software engineering: Software engineering in practice*. 286–295.

[71] Alexander Tarvo, Nachiappan Nagappan, Thomas Zimmermann, Thirumalesh Bhat, and Jacek Czerwonka. 2013. Predicting risk of pre-release code changes with checkinmentor. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 128–137.

[72] Hailemelekot Demtse Tessema and Surafel Lemma Abebe. 2021. Enhancing Just-in-Time Defect Prediction Using Change Request-based Metrics. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 511–515.

[73] Yuli Tian, Ning Li, Jeff Tian, and Wei Zheng. 2020. How Well Just-In-Time Defect Prediction Techniques Enhance Software Reliability?. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 212–221.

[74] Parastou Tourani and Bram Adams. 2016. The impact of human discussions on just-in-time quality assurance: An empirical study on openstack and eclipse. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 189–200.

[75] Alexander Trautsch, Steffen Herbold, and Jens Grabowski. 2020. Static source code metrics and static analysis warnings for fine-grained just-in-time defect prediction. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 127–138.

[76] Zhiyuan Wan, Xin Xia, Ahmed E Hassan, David Lo, Jianwei Yin, and Xiaohu Yang. 2018. Perceptions, expectations, and challenges in defect prediction. *IEEE Transactions on Software Engineering* 46, 11 (2018), 1241–1266.

[77] Ning Xie, Gabrielle Ras, Marcel van Gerven, and Derek Doran. 2020. Explainable deep learning: A field guide for the uninitiated. *arXiv preprint arXiv:2004.14545* (2020).

[78] Zhou Xu, Kunsong Zhao, Tao Zhang, Chunlei Fu, Meng Yan, Zhiwen Xie, Xiaohong Zhang, and Gemma Catolino. 2021. Effort-Aware Just-in-Time Bug Prediction for Mobile Apps Via Cross-Triplet Deep Feature Embedding. *IEEE Transactions on Reliability* (2021).

[79] Meng Yan, Xin Xia, Yuanrui Fan, Ahmed E Hassan, David Lo, and Shanping Li. 2020. Just-in-time defect identification and localization: A two-phase framework. *IEEE Transactions on Software Engineering* (2020).

[80] Meng Yan, Xin Xia, Yuanrui Fan, David Lo, Ahmed E Hassan, and Xindong Zhang. 2020. Effort-aware just-in-time defect identification in practice: a case study at Alibaba. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1308–1319.

[81] Limin Yang, Xiangxue Li, and Yu Yu. 2017. Vuldigger: a just-in-time and cost-aware tool for digging vulnerability-contributing changes. In *GLOBECOM 2017-2017 IEEE Global Communications Conference*. IEEE, 1–7.

[82] Xinli Yang, David Lo, Xin Xia, and Jianling Sun. 2017. TLEL: A two-layer ensemble learning approach for just-in-time defect prediction. *Information and Software Technology* 87 (2017), 206–220.

[83] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. 2015. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 17–26.

[84] Xingguang Yang, Huiqun Yu, Guisheng Fan, Kai Shi, and Liqiong Chen. 2019. Local versus global models for just-in-time software defect prediction. *Scientific Programming* 2019 (2019).

[85] Xingguang Yang, Huiqun Yu, Guisheng Fan, and Kang Yang. 2020. A differential evolution-based approach for effort-aware just-in-time software defect prediction. In *Proceedings of the 1st ACM SIGSOFT International Workshop on Representation Learning for Software Engineering and Program Languages*. 13–16.

[86] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. 2016. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 157–168.

[87] Steven Young, Tamer Abdou, and Ayse Bener. 2018. A replication study: just-in-time defect prediction with ensemble learning. In *Proceedings of the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*. 42–47.

[88] Andreas Zeller, Thomas Zimmermann, and Christian Bird. 2011. Failure is a four-letter word: A parody in empirical research. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*. 1–7.

[89] Zhengran Zeng, Yuqun Zhang, Haotian Zhang, and Lingming Zhang. 2021. Deep just-in-time defect prediction: how far are we?. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 427–438.

[90] Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. 2016. Towards building a universal defect prediction model with rank transformed predictors. *Empirical Software Engineering* 21, 5 (2016), 2107–2145.

[91] Wenzhou Zhang, Weiwei Li, and Xiuyi Jia. 2019. Effort-Aware Tri-Training for Semi-supervised Just-in-Time Defect Prediction. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 293–304.

[92] Kunsong Zhao, Zhou Xu, Meng Yan, Lei Xue, Wei Li, and Gemma Catolino. 2021. A compositional model for effort-aware Just-In-Time defect prediction on android apps. *IET Software* (2021).

[93] K. Zhao, Z. Xu, T. Zhang, Y. Tang, and M. Yan. 2021. Simplified Deep Forest Model Based Just-in-Time Defect Prediction for Android Mobile Apps. *IEEE Transactions on Reliability* (2021), 1–12. https://doi.org/10.1109/TR.2021.3060937

[94] Kun Zhu, Nana Zhang, Shi Ying, and Dandan Zhu. 2020. Within-project and cross-project just-in-time defect prediction based on denoising autoencoder and convolutional neural network. *IET Software* 14, 3 (2020), 185–195.
[95] Xiaoyan Zhu, Binbin Niu, E James Whitehead Jr, and Zhongbin Sun. 2018. An empirical study of software change classification with imbalance data-handling methods. *Software: Practice and Experience* 48, 11 (2018), 1968–1999.

A Systematic Survey of Just-In-Time Software Defect Prediction: Online Supplement

YUNHUA ZHAO, CUNY Graduate Center, USA
KOSTADIN DAMEVSKI, Virginia Commonwealth University, USA
HUI CHEN[*][†], CUNY Brooklyn College, USA

CCS Concepts: • **Software and its engineering** → **Software defect analysis**; **Risk management**; • **Computing methodologies** → *Cross-validation*; *Learning settings*; *Learning paradigms*.

ACM Reference Format:
Yunhua Zhao, Kostadin Damevski, and Hui Chen. 2022. A Systematic Survey of Just-In-Time Software Defect Prediction: Online Supplement. *ACM Comput. Surv.* Vol, Num, Article 1 (January 2022), 23 pages. https://doi.org/xx.xxxx/xxyyzza.aabbcde

This is an supplement that accompanies survey article with the same title. For convenience, we use "this survey" or "the survey" to refer to the survey article, an "this supplement" or "the supplement" this online supplement.

In the supplement, Section 1 details the methodology of this systematic survey, Section 2 illustrates the number of JIT-SDP studies over time, Section 3 lists the software projects used for evaluating JIT-SDP models in the surveyed JIT-SDP studies, Section 4 gives additional and detailed information about software metrics (features or independent variables), Section 5 lists machine learning models that the JIT-SDP studies are based on, Section 6 examines the availability of replication packages in the JIT-SDP studies, finally, Section 7 lists the surveyed JIT-SDP studies and provides a one-sentence summary describing the primary topic of each study.

1 REVIEW METHODOLOGY

Kitchenham et al. [39, 40] advocate a systematic literature review method in software engineering aiming at providing scientific value to the research community. According to Kitchenham et al. [38, 40], a systematic literature review process consists of the stages of planning the review (including identifying the need for the review, specifying the research questions, and developing a review protocol), conducting the review, and reporting the review.

[*]The corresponding author
[†]Also with CUNY Graduate Center, Department of Computer Science.

Authors' addresses: Yunhua Zhao, Department of Computer Science, CUNY Graduate Center, 365 5th Avenue, New York, NY, USA, 10016, yzhao5@gradcenter.cuny.edu; Kostadin Damevski, Department of Computer Science, Virginia Commonwealth University, 401 West Main Street, Richmond, VA, USA, 23284, damevski@acm.org; Hui Chen, Department of Computer & Information Science, CUNY Brooklyn College, 2900 Bedford Avenue, Brooklyn, NY, USA, 11210, huichen@acm.org.

0360-0300/2022/1-ART1 $15.00
https://doi.org/xx.xxxx/xxyyzza.aabbcde

1.1 Literature Search

Through the systematic literature review process, we use two methods to identify relevant studies, digital library keyword search and literature snowballing.

1.1.1 Digital Library Keyword Search. To locate existing surveys and papers, we use the digital libraries listed in Table 1. These digital libraries archive and index leading journals and conference proceedings in Software Engineering and the related. For instance, they index and archive the conference proceedings and journals in Table 2. Not surprisingly, existing software engineering research surveys also reference to these digital libraries. For instance, Li et al. [42] cite digital libraries 1–4 as the digital libraries to carry out their survey while Zakari el al. [79] 1–5.

Table 1. Digital Libraries

Digital Libraries	URL to Query User Interface
1. ACM	https://dl.acm.org/
2. IEEE Xplore	https://ieeexplore.ieee.org/
3. ScienceDirect	https://www.sciencedirect.com/search/
4. SpringerLink	https://link.springer.com/
5. Wiley	https://onlinelibrary.wiley.com/

1.1.2 Literature Snowballing. The digital library keyword search may not identify all of the relevant studies. To alleviate this problem, we use the snowball method to discover new studies starting with the selected articles from the previous step. We consider a variant of the snowball method called the backward (or reverse) snowball where we examine the references of an identified article. Empirical evidence suggests that the snowball method should be effective to locate "high quality sources in obscure locations [20]."

1.2 Planning

Kitchenham et al.[37, 38] published a guideline in 2007 and refined it in 2013. Applying the method by Kitchenham et al., we begin this research with an exploratory phase, an informal search and examination of literature about defect prediction. This belongs to the planning stage of Kitchenham et al.'s systematic method.

Software defect prediction (SDP) has been a long standing research subject for nearly half a century since 1970s. Not only have researches in this area evolved and taken different directions, but also there are relevant systematic surveys in the literature developed over time. Following the exploration phase, we proceed to the second phase of the planning stage, i.e., we carry out a meta-survey whose process we describe in Section 1.2.1 of this supplement. Kitchenham et al. term this type of survey as a tertiary survey, a systematic survey of systematic surveys [38] and argue that it is potentially less resource intensive to conduct a tertiary survey than conduct a new systematic review of primary studies in order to answer wider research questions [38]. In this meta-survey phase, we investigate existing surveys on SDP. As the result of this phase, we articulate the need to conduct this literature review on Just-In-Time Defect Prediction (JIT-SDP) in Section 1 of this survey and define the research questions in Section 1.3.1 of this supplement.

Following the planning stage, we turn our focus to a systematic literature review on JIT-SDP and describe the process in Section 1.3 of this supplement. With this focused survey, we answer the research questions in Sections 3, 4, and 5 of the survey.

Query 1. Semantics of digital library keyword search query for meta survey

```
(
    (
        (fault OR defect OR bug OR exception OR failure OR error)
            AND
        (prediction OR model))
    )
    OR
    (
        (fault OR defect OR bug OR exception OR failure OR error)
            AND
        risk
            AND
        (assessment OR prediction OR model)
    )
)
AND
(
    review OR survey OR mapping OR progress OR accomplishment OR critique
)
```

1.2.1 *Meta-Survey.*

The goal of the meta survey is to define the scope of SDP, to learn its relationship with the related areas, to understand the topics surveyed in prior literature *surveys* or *reviews* on SDP.

Researchers and practitioners have used a range of terms to refer the scenarios that software exhibit undesired behavior or outputs. These terms include "*defect*", "*fault*", "*bug*", "*error*", "*failure*", and "*exception*". These occur in either a piece of "software" or in a "program". Based on these, we construct Query 1. The digital libraries in Table 1 vary by their user interfaces and the query is to convey the semantics of our queries using the digital libraries.

Survey Article Publication Venues. Kitchenham et al. points out that the quality of tertiary surveys depends on the quantity and quality of systematic reviews [38]. In order to control the quality of the meta-survey, we choose only survey papers from the most significant software engineering journals and conferences by consulting Google Scholar[1] , the Computing Research and Education Association of Australasia (CORE)[2], Microsoft Academic where we select the "Topic" "Computer science" and its subtopic "Software engineering"[3], and journals and conferences cited by prior surveys. These journals and conferences are in Table 2.

1.2.2 *Results of Literature Search for Meta-Survey.*

We list in Table 3 the prior surveys on SDP that we identify. These SDP surveys have focused on a variety of aspects of the SDP problem, including the specific definition of the problem (e.g., predicting a probability or binary value), selected features, data granularity, training and test datasets, model design and evaluation metrics. While some of the surveys mention JIT-SDP, they focus only on the difference in data type (i.e., JIT-SDP uses software changes), but fail to provide coverage of the more nuanced aspects of the problem. For instance, JIT-SDP introduces a label identification latency stemming from the fact

[1]See https://scholar.google.com/citations?view_op=top_venues&vq=eng_softwaresystems
[2]See http://portal.core.edu.au/jnl-ranks/?search=software&by=all&source=CORE2020&sort=arank&page=1
[3]See https://academic.microsoft.com/

Table 2. Selected Software Engineering Conferences and Journals

Publication Type	Publication
Conferences	ACM/IEEE International Conference on Software Engineering (ICSE)
	ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)
	ACM/IEEE International Conference on Automated Software Engineering (ASE)
	ACM/IEEE International Conference on Mining Software Repositories (MSR)
	ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)
	IEEE International Conference on Software Maintenance and Evolution (ICSME)
	IEEE International Conference on Software Quality, Reliability and Security (QRS)
Journals	ACM Transactions on Software Engineering and Methodology (TSEM)
	IEEE Transactions on Software Engineering (TSE)
	IEEE Transactions on Reliability (TR)
	(Elsevier) Journal of Systems and Software (JSS)
	(Elsevier) Information and Software Technology (IST)
	(Elsevier) Applied Soft Computing
	(Springer) Empirical Software Engineering (ESE)
	(Wiley) Software Testing, Verification & Reliability (STVR)
	(Wiley) Software: Practice and Experience

This table is for filtering the digital library search results to identify SDP surveys. It is not for identifying JIT-SDP studies

that it takes time for developers to identify defects, which, in turn, changes certain past software changesets from clean to defect-inducing.

Table 3. Summary of Software-Defect Prediction Surveys

No.	Authors	Duration	Survey Coverage # of Articles Surveyed	Survey Topic
SV1	Li et al. [42]	2000–2018	49	Unsupervised SDP
SV2	Li et al. [44]	2014–2017	70	Comprehensive
SV3	Hosseini et al. [26]	2002–2017	46	Cross-project SDP
SV4	Kamei and Shihab [32]	1992–2015	65	Comprehensive
SV5	Malhotra [47]	1995–2013	64	Within-project & cross-project SDP
SV6	Radjenović et al. [56]	1991–2011	106	Software metrics for SDP
SV7	Hall et al. [22]	2000–2010	208	Within-project & cross-project SDP
SV8	Catal et al. [9][1]	1990–2009	68	Datasets, metrics, and models
SV9	Fenton and Neil [16][2]	1971–1999	55	Defect, failure, quality, complexity, metrics, and models

[1]Catal et al. [9] investigate 90 software defect/fault prediction papers in their survey, but only cite 68. We use this as the number of papers studied in their survey.

[2] Fenton and Neil [16] do not list explicitly the paper surveyed, and we only count the papers relevant to software metrics, defects, faults, quality, and failures.

As the result of the meta survey, we are able 1) to justify the need for a focused survey on JIT-SDP; 2) to provide background information for JIT-SDP, such as, clear definitions of defect and SDP; and 3) to determine the distinct aspects of JIT-SDP to focus our survey on.

1.3 Focused Survey on JIT-SDP

Upon the completion of the meta-survey, we commerce the focused survey on JIT-SDP.

1.3.1 Research Questions. Our end goal is to provide a comprehensive understanding of the state of the art of JIT-SDP. For this, we define and answer the following research questions.

RQ.1 What is the scope of the SDP research? Our literature search results in several related terms or areas about predicative modeling for quality assurance in software engineering. These terms include software reliability prediction, software failure prediction, software fault prediction, and software defect prediction. It begs the question how we define SDP, what the scope of SDP is, and how we differentiate it from the related areas.

RQ.2 What is the scope of the JIT-SDP research? JIT-SDP is an area in SDP. To comprehend the studies in JIT-SDP and present our understanding of it in a way complementary to other than repeating prior surveys in SDP, we need to identify the scope of JIT-SDP and to differentiate JIT-SDP from SDP that has been investigated in prior surveys, and present our understanding in the context of SDP, a larger area than JIT-SDP.

RQ.3 What are input data and the features or the independent variables in JIT-SDP? A necessary type of data to JIT-SDP is software changesets. Are there any other types of data that can help improve JIT-SDP? What are the features that we can extract from the input data? How do these features impact JIT-SDP performance? These are not only helpful to build JIT-SDP models, but also potentially aid our understanding on the relationship between factors in the software development life cycle process and defect occurrences, which help produce explainable and actionable models and insights.

RQ.4 On what target do we make predictions and what are the dependent variables in JIT-SDP? JIT-SDP is to predict defects on software changes. Are software changes the only target on which we predict defects? What are we really predicting? In another word, is defect proneness the only dependent variable? Understanding these is important to understand the limitation and the potential of existing model techniques.

RQ.5 What are the modeling techniques in JIT-SDP? Statistical analysis and machine learning are important model building techniques for JIT-SDP. What are the machine learning techniques used in JIT-SDP and how do they compare, such as, in terms of predicative performance? This helps address several issues. First, what machine learning techniques should we explore to continue to improve JIT-SDP? Second, which machine learning technique should we choose as a baseline to compare with if we are to build a new model? Third, if a user wishes to use JIT-SDP to help QA, which machine learning model should the user choose. Last, but not the least, is machine learning the only way to build JIT-SDP models? If not, how do the alternative approaches perform when compared with machine learning?

RQ.6 What are the evaluation strategies and criteria used in the existing JIT-SDP models? First, to understand the strength and the limitation of a JIT-SDP model, we need to know how we evaluate it. Second, to be able to assemble and compare existing models, we need to understand the evaluation criteria and strategies.

RQ.7 How is performance of JIT-SDP models with respect to the evaluation criteria? Which JIT-SDP model performs the best? This helps researchers to develop new models and compare it with the existing ones and helps users to select existing ones to build applications of JIT-SDP.

RQ.8 How are the JIT-SDP researches address the reproducibility (or replication) problem? Reproducibility is an important problem that has garnered increased scrutiny from the research community and the public in empirical research. JIT-SDP is an empirical research. Reproducibility is an important concern. How is the practice of the prior JIT-SDP studies with regard to facilitating replication to examine whether a JIT-SDP research is reproducible.

We focus our survey on JIT-SDP. The answer to RQ. 1 is thus out of scope of this survey. The answer to RQ.2 is in Section 3.1 of the survey where we define Release SDP and JIT-SDP. Sections 3.2

Query 2. Semantics of digital library keyword search query for JIT-SDP

```
(
    (
        (fault OR defect OR bug OR exception OR failure OR error)
            AND
        (prediction OR model))
    )
    OR
    (
        (fault OR defect OR bug OR exception OR failure OR error)
            AND
        risk
            AND
        (assessment OR prediction OR model)
    )
)
    AND
(just-in-time OR change)
    AND
(year >= 2000)
```

and 3.3 of the survey answer RQ.3. In Sections 3.6.1 and 3.6.2 of the survey, we divide JIT-SDP models into two categories, defect prediction and effort-aware prediction based on dependent variables, which is an answer to RQ.4. Section 3.6 of the survey documents modeling techniques, thus answers RQ. 5. For RQ.6, we report JIT-SDP evaluation strategies in Section 3.7 of the survey. Through a synthesis of the prior JIT-SDP studies, we provide an answer to RQ.7. In Section 6 of this online supplement, we collect and discuss replication package and data, which is an answer to RQ.8.

1.3.2 Digital Library Keyword Search Query for JIT-SDP. Kamei et al. coined the term "Just-in-time" Quality Assurance in their 2012 article [33]. JIT-SDP is the change-level SDP, i.e., is to predict existence of defects in software changes. Mockus and Weiss appear to be the first to examine change-level defect prediction [49] in year 2000. Following Query 2, we search the digital libraries in Table 1.

1.3.3 Literature Selection via 2-Pass Review. We combine all of the search results from the digital libraries and remove duplicates and divide the set of articles among the authors of this survey to evaluate whether to include or discard an article. The division ensures that we assign each article to two of the three authors and each article goes through two reviews by the two assigned authors (thus, the 2-pass review). Each author follows the following process. First, we remove any article whose title clearly indicates that it is not relevant. Second, for the remaining articles, we evaluate whether or not to include them by reading the abstract. Finally, we convene a meeting and resolve the difference via a discussion.

1.3.4 Exclusion and Inclusion Criteria. We include only articles written in English that study predictive modeling for JIT-SDP whose prediction is on the level or sub-level of software changes. For instance, we exclude Amasaki et al. [2] because they make predictions on the level of software component albeit claiming that they study JIT-SDP. We also exclude non-peer reviewed articles, posters, and abstract-only articles.

1.3.5 Results of Literature Search for JIT-SDP. We search JIT-SDP studies published from 2000 and completed our literature search in November 2021. Table 4 summarizes the literature search process and the results. The digital library keyword search yield 881 entries. After we remove duplicates, complete a two-pass review, we identify 55 JIT-SDP articles. We then begin the snowballing process on these 55 JIT-SDP articles. As shown in Table 4, these 55 JIT-SDP papers lists in total 2563 entries in their reference sections. After removing duplicates and a two-pass review, we find 12 additional JIT-SDP articles. Table 15 in Section 7 lists these 67 articles and provides a one-sentence summary describing the primary topic of each study.

Table 4. JIT-SDP Literature Search Results

Digital Library & Sources	Additional Constraint	# of Articles	
		Library Search or Snowballing	JIT-SDP after 2-Pass Review
ACM	–	196	
IEEE Xplore	–	55	
ScienceDirect	Research articles	269	55
SpringerLink	–	334	
Wiley	Computer Science	27	
Snowballing	On 55 JIT-SDP papers	2563	12

2 PUBLICATIONS TREND

Figure 1 plots the number of selected JIT-SDP papers versus publication year[4]. It shows that there has been an elevated interest in JIT-SDP in recent years.

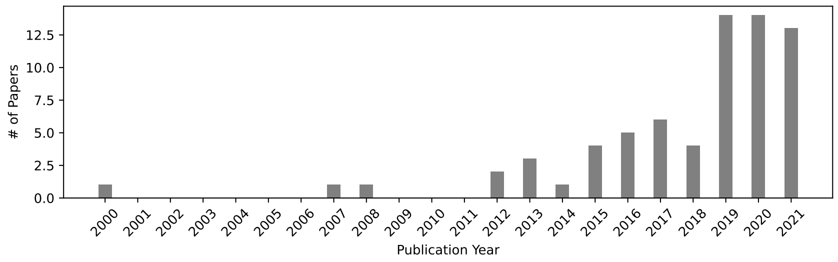


Fig. 1. The number of selected JIT-SDP papers over publication year

3 EVALUATION DATA

Table 5 is a summary of software projects used for evaluating JIT-SDP models. Most studies use open source projects. As shown in Table 5, 11 studies include proprietary/commercial projects among the 67 papers surveyed (listed in Table 15).

[4]The publication year is from the online publication date if available. The online publication date may be different from the bibliographic or the final publication date.

Table 5. Software Projects Used for Evaluation in JIT-SDP Studies

Begin of Table

Count Total(Proprietary)	Projects	Study
11(5)	Bugzilla, Eclipse JDT, Eclipse Platform, Mozilla, Columba, PostgreSQL; 5 commercial projects	Kamei et al. [33]
6(0)	Bugzilla, Columba, Eclipse JDT, Eclipse Platform, Mozilla, PostgreSQL	Yang et al. [73]
		Yang et al. [77]
		Fu and Menzies [17]
		Huang et al. [27]
		Liu et al. [46]
		Yang et al. [72]
		Young et al. [78]
		Albahli [1]
		Qiao and Wang [54]
		Yang et al. [76]
		Yang et al. [74]
		Zhang et al. [81]
		Huang et al. [28]
		Chen et al. [11]
		Li et al. [43]
		Yang et al. [75]
		Zhu et al. [85]
5(0)	Bugzilla, Columba, Eclipse Platform, Mozilla, PostgreSQL	Bennin et al. [6]
4(0)	Eclipse Platform, Eclipse JDT, Mozilla, PostgreSQL	Jahanshahi et al. [29]
4(0)	Bugzilla, Eclipse Platform, Eclipse JDT, Mozilla	Tessema et al. [64]
11(0)	Bugzilla, Columba, Eclipse JDT, Eclipse Platform, Mozilla, PostgreSQL; Gimp, Maven-2, Perl, Ruby on Rails, Rhino Fukushima et al. [18]	Kamei et al. [31]
2(0)	QT, OpenStack	McIntosh and Kamei [48]
		Hoang et al. [24]
		Rodriguezperez et al. [58]
		Hoang et al. [25]
		Gesi et al. [19]
		Pornprasit et al. [53]
		Zeng et al. [80]
15(0)	Android Firewall, Alfresco, Android Sync, Android Wallpaper, AnySoft Keyboard, Apg, Applozic Android SDK, Chat Secure Android, Delta Chat, Android Universal Image Loader, Kiwix, Observable Scroll View, Own Cloud Android, Page Turner, Notify Reddit	Catolino et al. [10]
		Zhao et al. [83]
		Zhao et al. [82]
		Zhao et al. [84]
19(0)	Android Firewall, Alfresco, Android Sync, Android Wallpaper, AnySoft Keyboard, Apg, Applozic Android SDK, Chat Secure Android, Delta Chat, Android Universal Image Loader, Kiwix, Observable Scroll View, Own Cloud Android, Page Turner, Notify Reddit, Facebook Android SDK, Lottie, Atmosphere, Telegram	Xu et al. [68]
10(0)	Apache ActiveMQ, Camel, Derby, Geronimo, Hadoop Common, HBase, Mahout, OpenJPA, Pig, Tuscany	Fan et al. [15]
8(0)	Apache ActiveMQ, Camel, Derby, Geronimo, Hadoop Common, HBase, OpenJPA, Pig	Duan et al. [12]
18(0)	Apache ActiveMQ, Ant, Camel, Derby, Geronimo, Hadoop, HBase, IVY, JCR, JMeter, LOG4J2, LUCENE, Mahout, OpenJPA, Pig, POI, VELOCITY, Xerces-C++	Tian et al. [65]

To be continued

Continuation of Table 5

Count	Projects	Study
13(3)	Apache Fabric8, Camel, Tomcat; JGroups; Brackets; OpenStack Neutron, Nova; Spring-integration; Broadleaf Commerce; NPM; and 3 proprietary projects	Tabassum et al. [61]
	Cabral et al. [8]	
2(0)	Apache Cassandra, Hadoop	Quach et al. [55]
39(0)	Apache Ant-Ivy, Archiva, Calcite, Cayenne, Commons BCEL, Commons BeanUtils, Commons Codec, Commons Collections, Commons Compress, Commons Configuration, Commons DBCP, Commons Digester, Commons IO, Commons Jcs, Commons JEXL, Commons Lang, Commons Math, Commons Net, Commons SCXML, Commons Validator, Commons VFS, DeltaSpike, Eagle, Giraph, Gora, JSPWiki, Knox, Kylin, Lens, Mahout, ManifoldCF, Nutch, OpenNLP, Parquet-MR, Santuario-java, SystemML, Tika, Wss4j	Trautsch et al. [67]
10(0)	Apache Lucene, Tomcat, jEdit, Ant, Synapse, Flink, Hadoop; Voldemort; iTextpdf; Facebook Buck	Zhu et al. [86]
20(0)	Apache Accumulo, Camel, Cinder, Kylin, Log4j, Tomcat; Eclipse Jetty; OpenStack Nova; Angular-js, Brackets, Bugzilla, Django, Fastjson, Gephi, Hibernate-ORM, Hibernate-Search, ImgLib2, osquery, PostgresSQL, Wordpress	Lin et al. [45]
10(0)	Apache Accumulo, Hadoop, OpenJPA; Angular-js; Bugzilla; Eclipse Jetty; Gerrit; Gimp; JDeodorant; JRuby	Pascarella et al. [52]
6(0)	ZooKeeper, Xerces-Java, JFreeChart, Jackson Data Format, Jackson Core, Commons Imaging	Ardimento et al. [3]
14(0)	Deeplearning4j, JMeter, H2O, LibGDX, Jetty, Robolectric, Storm, Jitsi, Jenkins, Graylog2-server, Flink, Druid, Closure-compiler, Activemq	Yan et al. [69]
1(0)	Jenkins	Borg et al. [7]
5(0)	Apache Hadoop, Camel; Gerrit; OsmAnd; Bitcoin; Gimp	Kondo et al. [41]
1(0)	Mozilla Firefox	Yang et al. [71]
15(0)	OpenStack Cinder, Devstack, Glance, Heat, Keystone, Neutron, Nova, OpenStack-Manuals, Swift, Tempest; Eclipse CDT, EGit, JGit, Linux-Tools, Scout.rt	Tourani and Adams [66]
3(0)	Apache OpenJPA, James; Eclipse Birt	Mori et al. [50]
6(0)	Linux Kernel, PostgreSQL, Xorg Xserver, Eclipse JDT, Lucene, Jackrabbit	Jiang et al. [30]
7(1)	Linux Kernel, PostgreSQL, Xorg Xserver, Eclipse JDT, Lucene, Jackrabbit; 1 Cisco project (proprietary)	Tan et al. [62]
12(0)	Apache HTTP 1.3 Sever, Bugzilla, Columba, Gaim, GForge, jEdit, Mozilla, Eclipse JDT, Plone, PostgreSQL, Scarab, Subversion	Kim et al. [36]
11(1)	Apache HTTP 1.3 Server, Columba, Gaim, GForge, jEdit, Mozilla, Eclipse JDT, Plone, PostgreSQL, Subversion; and a commercial project (proprietary, in Java)	Shivaji et al. [60]
2(0)	JHotDraw and DNS-Java	Aversano et al. [4]
324(0)	324 unspecified repositories	Barnett et al. [5]
21(0)	21 unspecified OSS projects	Khanan et al. [35]
2(2)	2 maritime projects (proprietary)	Kang et al. [34]
1(1)	1 unspecified project (proprietary)	Eken et al. [14]
14(14)	14 unspecified Alibaba projects (proprietary, mainly in Java)	Yan et al. [70]
0(1)	1 telecommunication project (proprietary)	Eken et al. [13]
12(12)	12 Ubisoft projects (proprietary)	Nayrolles et al. [51]
1(1)	Windows Phone (proprietary)	Tarvo et al. [63]
1(1)	5ESS®switching system software (proprietary)	Mockus and Weiss [49]

End of Table

4 SOFTWARE METRICS AND FEATURES FOR JIT-SDP

In Section 3.3 of this survey, we provide a table of categories of software metrics that prior studies find useful for JIT-SDP. We provide a more detailed discussion for these categories of metrics.

4.1 Software Change Metrics

Kamei et al. summarize past studies on the relationship between the characteristics of software changes and defects, and list 14 software change metrics that have been useful for JIT-SDP [33]. Liu et al. build an unsupervised effort-aware JIT-SDP model called CCUM and argue that the code churn metric, i.e., the size of a code change is particularly useful for unsupervised models [46]. Kondo et al. argue that the context lines of a software change, i.e., the lines of code surround the changed lines in a software change has an impact on defect proneness of the software change [41]. They propose and evaluate a suite of metrics called the context metrics, i.e, the metrics computed from the context lines [41]. Additionally, they also adapt the indentation metric from Hindle et al. [23] and propose two change complexity metrics [41].

Pascarella et al. investigate defect prediction on a finer granularity than software changes, i.e., to predict whether a specific file in the software change is defect prone [52]. They adapt the process metrics in Rahman and Devanbu [57] and evaluate a suite of file-level process metrics for changesets [52]. Table 7 summarizes file-level software change metrics; each of the metrics is computed on a file in the changeset.

Table 6 lists the metrics by Kamei et al., Liu et al., and Kondo et al. [33, 41, 46] organized according to different categories.

4.2 Commit Message Features

Since commit messages are in natural language text, features to encode them are usually borrowed from the natural language processing literature. An example is term frequency (TF) that counts the occurrences of a specific word in the commit message, e.g., used by Tan et al. [62]. A typical workflow to compute the TF feature is to assemble the corpus of the commit messages, i.e., the collection of all commit messages, to use a stemmer to obtain the root of each word, to remove stop words or rare words, to obtain a word dictionary, to assign an index to each word in the dictionary, and finally to form a vector recording occurrences of the dictionary words in a commit message. The TF vector of a commit message is likely a sparse vector with most elements as 0. Barnett et al. [5] hypothesize that the level of detail in commit messages is useful for JIT-SDP and confirm it via their investigation of more than 300 repositories. For this, they propose two commit message metrics, commit volume and commit content. The former is the number of words in a commit message after the stop words are removed. The later is a score computed via a SPAM filter, and this is in effect a feature representation of the commit message and a surrogate representing the content of the commit message. Table 8 lists these metrics.

4.3 ITS Data Features

ITS data, such as, issue reports, issue discussions, change requests, and code reviews can be useful to predict defects in future changes as the result of these data. Tourani and Adams [66] propose a suite of issue discussion and code review discussion metrics that attempt to capture the characteristics of these data sources other than actual textual content; Table 9 lists these metrics.

Tessema and Abebe [64] propose 6 metrics for change requests in ITS. They augment these metrics with Kamei et al [33]'s software change metrics and show that the JIT-SDP models with augmented metrics outperform those with the change metrics alone. Table 10 summarize the 6 change request metrics. These metrics are from the meta-data of the change requests.

Table 6. Software Change Metrics [31, 33, 46]

Category	Metric	Description
Diffusion	NS	Number of modified subsystems
	ND	Number of modified directories
	NF	Number of modified files
	Entropy	Distribution of modified code across each file
Size	LA	Lines of code added
	LD	Lines of code deleted
	LT	Lines of code in a file before the change
Purpose	FIX	Whether or not the change is a defect fix
History	NDEV	The number of developers that changed the modified files
	AGE	The average time interval between the last and current change
	NUC	The number of unique changes to the modified files
Experience	EXP	Developer experience
	REXP	Recent developer experience
	SEXP	Developer experience on a sub-system
Size	Churn	Size of the change, i.e., LA + LD
	RChurn	Relative Churn, i.e., (LA + LD)/LT
	RLA	Relative LA, i.e., LA / LT
	RLD	Relative LD, i.e., LD / LT
	RLT	Relative LT, i.e., LT / NF
Change Complexity (Indentation)	AS	Number of white spaces on all the "+" (added) lines in a commit
	AB	Sum of the difference of left-braces and right-braces on all the "+" lines in each function in a commit
Change Context	NCW	Number of words in context
	NCKW	Number of programming language keywords
	NCCW	number of words in the context and the changed lines
	NCCKW	Number of programming language keywords in the context and the changed lines

4.4 Static Program Analysis Metrics

Trautsch et al. [67] collect static program analysis warning messages from two popular tools, PMD and OpenStaticAnalyzer. From these warning messages, aimed at JIT-SDP, they derive the warning density metrics listed in Table 11.

5 JIT-SDP MODELS

The prior JIT-SDP studies have examined a broad range of machine learning algorithms. As such, one may argue that JIT-SDP is a microcosm of recent development in machine learning. It is important to note that there have been several investigations of non-machine-learning algorithms for JIT-SDP. We refer to this type of JIT-SDP models as searching-based models. Searching-based models like those in Yang et al. [77] and Liu et al. [46] are unsupervised. Several studies extend these unsupervised searching-based models by adding a supervised component to improve their predictive performance. Table 12 lists the modeling techniques in the prior JIT-SDP studies. It shows that Logistic Regression, Tree-based models (including Random Forest, C4.5 Decision Tree and ADTree) and ensemble models (including Random Forest, XGBoost, and others) are more popular modeling techniques and the use of neural network-based models (including deep neural networks) is on the rise.

Table 7. Software File Change Metrics [41, 52]

Category	Metric	Description
Change Process	COMM	Number of changes to the file up to the considered commit
	ADEV	Number developers who modified the file up to the considered commit
	DDEV	Cumulative number of distinct developers contributed to the file up to the considered commit
	ADD	Number of lines added to the file in the considered commit
	DEL	Number of lines removed from the file in the considered commit
	OWN	Whether the commit is done by the owner of the file
	MINOR	Number of contributors who contributed less than 5% of the file up to the considered commits
	SCTR	Number of packages modified by the committer in the commit
	NADEV	Number of developers who changed the files in the commits where the file has been modified
	NDDEV	Cumulative number of distinct developers who changed the files in commits where the file has been modified
	NCOMM	Number of commits made to files in commits where the file has been modified
	NSCTR	Number of different packages touched by the developer in commits where the file has been modified
	OEXP	Percentage of lines authored in the project
	AEXP	Mean of the experiences of all the developers who touched the file

Table 8. Commit Message Metrics [5, 62]

Feature	Description	Study
CM-TF	Term frequency of commit message	Tan et al. [62]
CM-VOLUME	Number of words in commit message excluding stop words	Barnett et al. [5]
CM-SPAM	Commit message content represented by SPAM score computed via a SPAM filter	Barnett et al. [5]

Table 9. Issue Report, Issue Discussion, and Code Review Metrics [66]

Category	Metric	Description
Thread Focus	COMMEXP	Commenter experience
	RPTEXP	Reporter experience
	RVWEXP	Reviewer experience
	PATCHNUM	Number of patch revisions
	NINLCMMT	Number of inline comments
Thread Length	NUMCMMT	Number of comments
	LENCMMT	Length of comments
Thread Time	RVWTIME	Review time
	FIXTIME	Fix time
	DISCLAG	Average discussion lag
Sentiment	CMMTSENT	Comment sentiment

6 REPLICATION PACKAGES AND DATA

Reproducibility is an important issue in empirical studies [17]. Table 13 lists the studies that indicate the availability of the replication packages among the 67 studies in Table 15. Among the studies, 2

Table 10. Change Request Metrics [64]

Metrics	Description
CR-TTM	Time span between submission of the change request to its resolution (a new change)
CR-NDA	Number of developers involved in the change request
CR-PRIORITY	Priorities assigned to the change request
CR-SEVERITY	Severity assigned to the change request
CR-NC	Number of comments about the change request
CR-DD	Depth of discussion computed as the number of words used during the discussion of the change request

Table 11. Static Program Analysis Metrics [67]

Category	Metric	Description
Program Analysis	`SysWD`	Warning density of the project
	`FSysWD`	Cumulative difference between warning density of the file and the project
	`AuDWD`	Cumulative sum of the changes in warning density by the author

replication packages appear no longer accessible and 2 are identical, which results in 26 accessible replication packages.

Some of the replication packages have an impact not only on the reproducibility of the studies that provide the packages, but also on generating new models or new insights or the both. For instance, Kamei et al. [33] make available a replication package including both the source code and the data sets. More than 10 studies take advantage of either the source code, the data set, or the both. Yang et al. [77] include the the source code in their replication package and the code allows Fu and Menzies [17] to quickly replicate the Yang et al.'s results and helps Fu and Menzies reach new discovery. Table 14 provide several examples of those impactful studies and their replication packages. The table also lists the studies that use the code of, or the data of, or the both code and the data in the replication packages.

The research community has curated additional tools that facilitate the JIT-SDP research. Two prime examples are Commit Guru and several SZZ implementations. Commit Guru [59] is a public available tool to compute software change metrics for software projects whose source code is in a Git repository. The studies that use the data extracted from Commit Guru include Tabassum et al. [61], Cabral et al. [8], Khanan et al. [35], and Kondo et al. [41]. Public available SZZ tools have made labeling of large collections of software changes feasible.

7 SELECTED JIT-SDP STUDIES

Table 15. JIT-SDP Studies and Primary Topics.

Begin of Table			
ID	Study	Year[a]	Primary Topics
P1	Ardimento et al. [3]	2021	applying temporal convolutional neural networks with hierarchical attention layers to a set of 40+ production and process software metrics data to predict defect proneness of SCM commits
P2	Duan et al. [12]	2021	modeling impact of duplicate changes, i.e., identical changes applied to multiple SCM branches on prediction performance

[a]The publication year is from the online publication date if available. The online publication date may be different from the bibliographic or the final publication date.

Continuation of Table 15

ID	Study	Year[a]	Primary Topics
P3	Eken et al. [14]	2021	deploying a JIT-SDP model to an industrial project (presumably closed source and proprietary), comparing online and offline prediction settings, presenting lessons learned
P4	Gesi et al. [19]	2021	addressing data imbalance beyond class label imbalance, i.e., data bias along dimensions, such as, File Count, Edit Count, and Multiline Comments on JIT-SDP predictive performance and proposing a few-short learning JIT-SDP model (`SifterJIT`) combing Siamese networks and DeepJIT [24]
P5	Lin et al. [45]	2021	investigating the impact of data merging on interpretation of cross-project JIT-SDP models (e.g., most important independent variables), and advocating mixed-effect models for sound interpretation.
P6	Pornprasit et al. [53]	2021	replicating CC2Vec [25] to contrast feature representation learning when including and excluding test data set, which leads to a Random Forest based JIT-SDP model (`JITLine`) to rank lines added in software changes based on defect-inducing risk via a Local Interpretable Model-Agnostic Explanations model (LIME)
P7	Quach et al. [55]	2021	observing SZZ's weaker ability to identify performance defects than do non-performance ones and studying impacts of non-performance defects on predictive performance of JIT-SDP models
P8	Tessema and Abebe [64]	2021	augmenting publicly available change metrics dataset with six change request-based metrics collected from issue tracking systems and examining their impact on JIT-SDP predictive performance.
P9	Xu et al. [68]	2021	designing a deep neural network triplet loss function (called `CDFE`) for cross-project JIT-SDP and learning high-level feature representations from software metrics data for improving defect prediction performance for mobile apps
P10	Zeng et al. [80]	2021	replicating CC2Vec [25] and DeepJIT [24] to discover the role of high-level feature about lines added to a software change, which leads to a performant logistic regress-based JIT-SDP model (`LAPredict`)
P11	Zhao et al. [83]	2021	proposing a deep neural network JIT-SDP model (`KPIDL`) that employs kernel-based PCA to learn high-level features from software metrics data and addressing class imbalance problem with a custom cross-entropy loss function and evaluating the model on Android apps
P12	Zhao et al. [82]	2021	proposing to address class imbalance problem using class weights, realizing it with a cross-entropy loss function in a deep neural network JIT-SDP model (`IDL`), and evaluating the model using software metrics data from Android apps
P13	Zhao et al. [84]	2021	applying a custom deep forest model for high-level feature representation learning from software metrics data and evaluating the model on Android apps
P14	Bennin et al. [6]	2020	investigating impact of concept drift in software change data on the performance of JIT-SDP
P15	Hoang et al. [25]	2020	considering the hierarchical structure of `diffs` of software changes and designing a feature representation learning framework (`CC2Vec`) using a convolutional network with hierarchical attention layers, and evaluating the framework using `DeepJIT` [24]
P16	Kang et al. [34]	2020	studying within and cross-project JIT-SDP for post-release changes of maritime software and integrating a cost-benefit analysis in the JIT-SDP models
P17	Khanan et al. [35]	2020	designing explainable JIT-SDP bot that uses a model-agnostic technique (LIME) to "explain" a defect proneness change prediction with the "contribution" of software metrics

[a]The publication year is from the online publication date if available. The online publication date may be different from the bibliographic or the final publication date.

Continuation of Table 15

ID	Study	Year[a]	Primary Topics
P18	Li et al. [43]	2020	investigating semi-supervised effort-aware JIT-SDP using a tri-training method (also see Zhang et al. [81])
P19	Rodriguez-Perez et al. [58]	2020	studying the impact of extrinsic bugs in JIT-SDP and concluding that extrinsic bugs negatively impact predictive performance of JIT-SDP models
P20	Tabassum et al. [61]	2020	designing online cross-project JIT-SDP models and concluding that combing incoming cross-project and within-project data can improve G-mean and reduce performance drops due to concept drift
P21	Tian et al. [65]	2020	evaluating long-term JIT-SDP for reliability improvement and short-term JIT-SDP for early defect prediction while considering the relationship of software usage and defect
P22	Trautsch et al. [67]	2020	designing static analysis warning message metrics, comparing two software change labeling strategies (ad hoc SZZ and ITS SZZ), and investigating predictive performance of sub-change-level (i.e., file in a changeset) JIT-SDP
P23	Yan et al. [69]	2020	proposing a two-phase model, which in the first phase predicts defectiveness of a software change and in the second phase rank defect inducing risks of lines added in predicted defect prone software changes via a probabilistic model
P24	Yan et al. [70]	2020	investigating the effectiveness of supervised (`CBS+`, `OneWay`, and `EALR`) and unsupervised (`LT` and `Code Churn`) *effort-aware* JIT-SDP models in an *industry* setting (on Alibaba projects)
P25	Yang et al. [75]	2020	proposing an effort-aware JIT-SDP model (`DEJIT`) that uses a differential evolution algorithm to optimize density-percentile-average (DPA) objective function, purposely designed for effort-aware prediction
P26	Zhu et al. [85]	2020	proposing a deep neural network model (`DAECNN-JDP`) based on denoising autoencoder and convolutional neural network and investigating the predictive performance of the model using software metrics data
P27	Albahli [1]	2019	devising an ensemble JIT-SDP model whose base classifiers are Random forest, XGBoost, and Multi-layer perceptron
P28	Borg et al. [7]	2019	presenting an open-source implementation of SZZ (`SZZ Unleashed`) and illustrating the use of it by applying a Random Forest classifier to predict defective commits in the Jenkins project
P29	Cabral et al. [8]	2019	investigating the problems of class imbalance evolution and verification latency in software change data and proposing an online JIT-SDP model based on Oversampling Online Bagging (ORB) to tackle these problems
P30	Catolino et al. [10]	2019	investigating cross-project JIT-SDP models for mobile apps and comparing model performance of four classifiers and four ensemble techniques
P31	Eken et al. [13]	2019	applying JIT-SDP to an industrial project of a telecommunication company in Turkey, for which, extracting features from multiple sources (software changes and commit messages)
P32	Fan et al. [15]	2019	investigating labeling errors of SZZ variants and their impacts on predictive performance of JIT-SDP
P33	Hoang et al. [24]	2019	proposing an “end-to-end” JIT-SDP model (`DeepJIT`) that learns feature representations from tokenized software changes (`diffs`) and commit messages and evaluating the predictive performance in the cross-validation, short-term, and long-term prediction settings
P34	Jahanshahi et al. [29]	2019	investigating concept drift by replicating the study by McIntosh and Kamei [48] using the data sets in Kamei et al. [33]

[a]The publication year is from the online publication date if available. The online publication date may be different from the bibliographic or the final publication date.

Continuation of Table 15

ID	Study	Year[a]	Primary Topics
P35	Kondo et al. [41]	2019	designing and investigating "context metrics", metrics that measure the complexity or the number of the surrounding lines of a change for JIT-SDP
P36	Pascarella et al. [52]	2019	designing JIT-SDP models to predict defectives of files in a commit and investigating the models using the product and the process metrics
P37	Qiao and Wang [54]	2019	applying a fully-connected neural network for effort-aware JIT-SDP
P38	Yang et al. [76]	2019	addressing limited training data problem by applying progressive sampling to identify a small but sufficient set of data for training JIT-SDP models
P39	Yang et al. [74]	2019	comparing local and global JIT-SDP models where local models are those trained using a subset of homogeneous data and global models trained using all of the training data
P40	Zhang et al. [81]	2019	investigation of semi-supervised effort-aware JIT-SDP model (`EATT`) using a tri-training method (also see Li et al. [43])
P41	Huang et al. [28]	2018	investigating a supervised effort-aware model (called `CBS`) combining Kamei et al.'s supervised `EALR` model [33] and Yang et al.'s unsupervised `LT` [77]
P42	Nayrolles et al. [51]	2018	designing a two-phase approach (called `CLEVER`) that in the first phase predicts defect risks of SCM commits and in the second phase suggests a possible fix by comparing with known fix-commits, also presenting lessons learned by deploying it to software company Ubisoft
P43	Young et al. [78]	2018	comparing the prediction of defect-prone changes using traditional machine learning techniques and ensemble learning algorithms by a replicating study
P44	Zhu et al. [86]	2018	experimenting class imbalance handing methods (resampling and ensemble learning methods) across learning algorithms for JIT-SDP, and examining effort-aware and defect proneness predictive performance and model interpretation (effects of contribution of different groups of change features on dependable variables)
P45	Yang et al. [71]	2017	proposing a model (`VulDigger`) that predicts vulnerability defect-inducing changes with a Random Forest classifier using software change metrics derived from both software defect prediction and vulnerability prediction
P46	Chen et al. [11]	2017	formulating JIT-SDP as a dual-objective optimization problem based on logistic regression and NSGA-II to balance the benefit, i.e., the number of predicted defective changes and the cost (the efforts of reviewing the software changes for quality assurance)
P47	Fu and Menzies [17]	2017	investigating Yang et al.[77]'s unsupervised models (e.g., `LT`) and proposing an effort-aware JIT-SDP model (`OneWay`) that uses the supervised models to prune unsupervised models before employing Yang et al.'s approach
P48	Huang et al. [27]	2017	investigating a supervised effort-aware model (called `CBS`) combining Kamei et al.'s supervised `EALR` model [33] and Yang et al.'s unsupervised models, such as, `LT` [77]
P49	Liu et al. [46]	2017	investigating the effectiveness of the code churn metric based unsupervised defect prediction model (`CCUM`) for effort-aware JIT-SDP
P50	McIntosh and Kamei [48]	2017	investigating evolving nature of software project leading to fluctuations of software metrics data (or concept drift) and presenting insights, such as, JIT models that should be retrained using recently recorded data
P51	Yang et al. [72],	2017	proposing and investigating a two-layer ensemble model (`TLEL`) for effort-aware JIT-SDP

[a]The publication year is from the online publication date if available. The online publication date may be different from the bibliographic or the final publication date.

Continuation of Table 15

ID	Study	Year[a]	Primary Topics
P52	Barnett et al. [5]	2016	investigating the usefulness of SCM commit message volume and commit message content for JIT-SDP and showing benefits by adding commit message features to software change defect prediction
P53	Kamei et al. [31]	2016	examining cross-project JIT-SDP and providing insights and guidelines to improve predictive performance (also see Fukushima et al. [18])
P54	Tourani and Adams [66]	2016	investigating usefulness of ITS data, such as, issue reports, issue discussions, and code reviews and designing ITS data metrics for JIT-SDP
P55	Yang et al. [77]	2016	investigating the predictive power of simple unsupervised models, such as, `LT` and `AGE` in effort-aware JIT defect prediction and comparing these simple models with supervised models
P56	Mori et al. [50]	2015	applying text classifiers (i.e., spam filter) to software changes to assess the probability of files in changesets to be defect-inducing
P57	Rosen et al. [59]	2015	describing a publicly available defect prediction tool called Commit Guru
P58	Tan et al. [62]	2015	investigating two problems, the class imbalance problem and the problem about cross-validation, and proposing online change classification for JIT-SDP using resampling and updatable classification techniques
P59	Yang et al. [73]	2015	proposing a model called `Deeper` consisting of a deep belief network and a logistic regression classifier to predict defect proneness of software changes
P60	Fukushima et al. [18]	2014	examining cross-project JIT-SDP and showing its feasibility by demonstrating that models trained using historical data from other projects can be as accurate as JIT-SDP models that are trained on a single project (also see Kamei et al. [31])
P61	Jiang et al. [30]	2013	building (file) change-level defect prediction model for each developer from file modification histories (i.e., a personalized defect prediction)
P62	Tarvo et al. [63]	2013	using a classification model to identify those pre-release code changes that can cause post-release failures using code metrics, change size, historical code churn, and organization metrics, and also investigating impacts of changes on trunk and branches
P63	Kamei et al. [33]	2012	predicting defect-proneness of software changes with logistic regression and quality assurance effort of software changes with linear regression (`EALR`) from software change metrics
P64	Shivaji et al. [60]	2012	investigating feature selection techniques for change-level defect prediction
P65	Kim et al. [36]	2008	proposing a JIT-SDP model based on Support Vector Machine (`SVM`) and using the *bag-of-words* features to classify whether software changes are defect-inducing or clean
P66	Aversano et al. [4]	2007	studying defect inducing change prediction by representing software snapshots as TF-IDF vectors and software changes as vector differences of two snapshots and by comparing multiple classification and clustering algorithms
P67	Mockus and Weiss [49]	2000	predicting from software change metrics with `logistic regression` the defect-proneness of the Initial Modification Requests (IMR) in 5ESS network switch project

End of Table

[a]The publication year is from the online publication date if available. The online publication date may be different from the bibliographic or the final publication date.

REFERENCES

[1] Saleh Albahli. 2019. A deep ensemble learning method for effort-aware just-in-time defect prediction. *Future Internet* 11, 12 (2019), 246.

Table 12. JIT-SDP Modeling Techniques

Algorithm	Studies
K-Nearest Neighbor	Bennin et al. [6], Kang et al. [34], Tian et al. [65], Aversano et al. [4]
Linear Regression	Kamei et al. [33], Tian et al. [65], Yan et al. [70]
Non-linear Regression	Rodriguezperez et al. [58], McIntosh and Kamei [48]
Logistic Regression	Duan et al. [12], Lin et al. [45], Zeng et al. [80] Yang et al. [75], Kang et al. [34], Li et al. [43], Trautsch et al. [67], Yan et al. [69], Yan et al. [70], Catolino et al. [10], Fan et al. [15], Huang et al. [28], Kondo et al. [41], Yang et al. [74], Chen et al. [11], Huang et al. [27], Tourani and Adams [66], Rosen et al. [59], Jiang et al. [30], Tarvo et al. [63], Kamei et al. [33], Aversano et al. [4], Mockus and Weiss [49], Tessema et al. [64]
Naive Bayes	Duan et al. [12], Eken et al. [14], Bennin et al. [6], Kang et al. [34], Tian et al. [65], Catolino et al. [10], Fan et al. [15], Zhu et al. [86], Barnett et al. [5], Jiang et al. [30], Shivaji et al. [60]
Decision Table	Catolino et al. [10],
C4.5 Decision Tree	Zhu et al. [86] Tarvo et al. [63], Aversano et al. [4]
Alternating Decision Tree (ADTree)	Tan et al. [62], Jiang et al. [30]
Random Forest	Fukushima et al. [18], Kamei et al. [31], Yang et al. [71], Nayrolles et al. [51], Zhu et al. [86], Borg et al. [7], Catolino et al. [10], Fan et al. [15], Jahanshahi et al. [29], Kondo et al. [41], Pascarella et al. [52], Yang et al. [76], Bennin et al. [6], Kang et al. [34], Khanan et al. [35], Li et al. [43], Trautsch et al. [67], Tian et al. [65], Duan et al. [12], Lin et al. [45], Pornprasit et al. [53], Quach et al. [55], Tessema et al. [64]
Support Vector Machine	Kang et al. [34], Li et al. [43], Catolino et al. [10], Zhu et al. [86], Shivaji et al. [60], Kim et al. [36], Aversano et al. [4]
Neural Network & Deep Neural Network	Yang et al. [73], Hoang et al. [24], Qiao and Wang [54], Bennin et al. [6], Hoang et al. [25], Kang et al. [34], Tian et al. [65], Zhu et al. [85], Tessema et al. [64] Ardimento et al. [3], Gesi et al. [19], Xu et al. [68], Zeng et al. [80], Zhao et al. [83], Zhao et al. [82],
Deep Forest	Zhao et al. [84]
Ensemble (XGBoost)	Bennin et al. [6], Eken et al. [13], Tessema et al. [64]
Ensemble (others)	Aversano et al. [4] Yang et al. [72], Young et al. [78], Albahli [1], Cabral et al. [8], Catolino et al. [10], Zhang et al. [81], Li et al. [43], Tabassum et al. [61], Tian et al. [65], Tessema et al. [64]
Spam Filter (Text Classifier)	Mori et al. [50]
Searching-based Algorithm	Liu et al. [46] Yang et al. [77]
Supervised Learning + Searching-based Algorithm	Yan et al. [70], Huang et al. [28], Huang et al. [27], Fu and Menzies [17]

[2] Sousuke Amasaki, Hirohisa Aman, and Tomoyuki Yokogawa. [n.d.]. A Preliminary Evaluation of CPDP Approaches on Just-in-Time Software Defect Prediction. In *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 279–286.

[3] Pasquale Ardimento, Lerina Aversano, Mario Luca Bernardi, Marta Cimitile, and Martina Iammarino. 2021. Just-in-time software defect prediction using deep temporal convolutional networks. *Neural Computing and Applications* (2021), 1–21.

[4] Lerina Aversano, Luigi Cerulo, and Concettina Del Grosso. 2007. Learning from bug-introducing changes to prevent fault prone code. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*. 19–26.

[5] Jacob G Barnett, Charles K Gathuru, Luke S Soldano, and Shane McIntosh. 2016. The relationship between commit message detail and defect proneness in java projects on github. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 496–499.

[6] Kwabena E Bennin, Nauman bin Ali, Jürgen Börstler, and Xiao Yu. 2020. Revisiting the impact of concept drift on just-in-time quality assurance. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 53–59.

Table 13. Replication Packages of JIT-SDP Studies

No.	Study	Year	Replication Package	Remark
1	Lin et al. [45]	2021	https://github.com/SAILResearch/suppmaterial-19-dayi-risk_data_merging_jit	
2	Pornprasit et al. [53]	2021	http://doi.org/10.5281/zenodo.4433498	
3	Quach et al. [55]	2021	https://github.com/senseconcordia/Perf-JIT-Models	
4	Xu et al. [68]	2021	https://figshare.com/search?q=10.6084/m9.figshare.13635347	
5	Zeng et al. [80]	2021	https://github.com/ZZR0/ISSTA21-JIT-DP	
6	Zhao et al. [83]	2021	https://github.com/sepine/IET-2021	
7	Duan et al. [12]	2021	https://github.com/deref007/Duplicate-change-TR	
8	Rodriguezperez et al. [58]	2020	https://gemarodri.github.io/2019-Study-of-Extrinsic-Bugs/	
9	Yan et al. [69]	2020	https://github.com/MengYan1989/JIT-DIL	
10	Hoang et al. [25]	2020	https://github.com/CC2Vec/CC2Vec	
11	Tian et al. [65]	2020	https://github.com/lining-nwpu/JiTReliability	
12	Trautsch et al. [67]	2020	https://doi.org/10.5281/zenodo.3974204	
13	Li et al. [43]	2020	https://github.com/NJUST-IDAM/EATT	
14	Borg et al. [7]	2019	https://github.com/wogscpar/SZZUnleashed	
15	Qiao and Wang [54]	2019	https://github.com/donaldjoe/Effort-Aware-and-Just-in-Time-Defect-Prediction-with-Neural-Network	
16	Yang et al. [74]	2019	https://github.com/yangxingguang/LocalJIT	
17	Fan et al. [15]	2019	https://github.com/YuanruiZJU/SZZ-TSE	
18	Hoang et al. [24]	2019	https://github.com/AnonymousAccountConf/	
19	Pascarella et al. [52]	2019		not found
20	Huang et al. [28]	2018	https://doi.org/10.5281/zenodo.1432582	
21	Cabral et al. [8]	2019	https://doi.org/10.5281/zenodo.2555695	
22	Zhang et al. [81]	2019	https://github.com/NJUST-IDAM/EATT	identical to Li et al. [43]
23	Guo et al. [21]	2018	https://github.com/yuchen1990/EAposter	
24	Chen et al. [11]	2017	https://github.com/Hecoz/Multi-Project-Learning	
25	Fu and Menzies [17]	2017	https://github.com/WeiFoo/RevisitUnsupervised	
26	McIntosh and Kamei [48]	2017	https://github.com/software-rebels/JITMovingTarget	
27	Huang et al. [27]	2017	https://doi.org/10.5281/zenodo.836352	
28	Yang et al. [77]	2016	http://ise.nju.edu.cn/yangyibiao/jit.html	inaccessible
29	Kamei et al. [33]	2012	http://research.cs.queensu.ca/~kamei/jittse/jit.zip	

Table 14. Use of Replication Packages

Replication Package Name	Original Study	Dependent Study
Kamei-2012	Kamei et al. [33]	Fukushima et al. [18], Yang et al. [73], Kamei et al. [31], Yang et al. [77], Huang et al. [27], Liu et al. [46], Guo et al. [21], Young et al. [78], Chen et al. [11], Jahanshahi et al. [29], Huang et al. [28], Albahli [1], Bennin et al. [6], Li et al. [43], Yang et al. [75], Tessema et al. [64]
Yang-2016	Yang et al. [77]	Fu and Menzies [17]
McIntosh-2017	McIntosh et al. [48]	Hoang et al. [24], Hoang et al. [25], Rodriguezperez et al. [58]
Catolino-2019	Catolino et al. [10]	Xu et al. [68], Zhao et al. [83], Zhao et al. [82], Zhao et al. [84]
Hoang-2019, Hoang-2020	Hoang et al. [25] and Hoang et al. [24]	Gesi et al. [19], Pornprasit et al. [53], Zeng et al. [80]

[7] Markus Borg, Oscar Svensson, Kristian Berg, and Daniel Hansson. 2019. SZZ Unleashed: An open implementation of the szz algorithm-featuring example usage in a study of just-in-time bug prediction for the jenkins project. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*. 7–12.

[8] George G Cabral, Leandro L Minku, Emad Shihab, and Suhaib Mujahid. 2019. Class imbalance evolution and verification latency in just-in-time software defect prediction. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 666–676.

[9] Cagatay Catal. 2011. Software fault prediction: A literature review and current trends. *Expert systems with applications* 38, 4 (2011), 4626–4636.

[10] Gemma Catolino, Dario Di Nucci, and Filomena Ferrucci. 2019. Cross-project just-in-time bug prediction for mobile apps: an empirical assessment. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 99–110.

[11] Xiang Chen, Yingquan Zhao, Qiuping Wang, and Zhidan Yuan. 2018. MULTI: Multi-objective effort-aware just-in-time software defect prediction. *Information and Software Technology* 93 (2018), 1–13.

[12] Ruifeng Duan, Haitao Xu, Yuanrui Fan, and Meng Yan. 2021. The impact of duplicate changes on just-in-time defect prediction. *IEEE Transactions on Reliability* (2021).

[13] Beyza Eken, RiFat Atar, Sahra Sertalp, and Ayşe Tosun. 2019. Predicting Defects with Latent and Semantic Features from Commit Logs in an Industrial Setting. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. IEEE, 98–105.

[14] Beyza Eken, Selda Tufan, Alper Tunaboylu, Tevfik Guler, Rifat Atar, and Ayse Tosun. 2021. Deployment of a change-level software defect prediction solution into an industrial setting. *Journal of Software: Evolution and Process* 33, 11 (2021), e2381.

[15] Yuanrui Fan, Xin Xia, Daniel Alencar da Costa, David Lo, Ahmed E Hassan, and Shanping Li. 2019. The impact of changes mislabeled by SZZ on just-in-time defect prediction. *IEEE transactions on software engineering* (2019).

[16] Norman E Fenton and Martin Neil. 1999. A critique of software defect prediction models. *IEEE Transactions on software engineering* 25, 5 (1999), 675–689.

[17] Wei Fu and Tim Menzies. 2017. Revisiting unsupervised learning for defect prediction. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 72–83.

[18] Takafumi Fukushima, Yasutaka Kamei, Shane McIntosh, Kazuhiro Yamashita, and Naoyasu Ubayashi. 2014. An empirical study of just-in-time defect prediction using cross-project models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. 172–181.

[19] Jiri Gesi, Jiawei Li, and Iftekhar Ahmed. 2021. An Empirical Examination of the Impact of Bias on Just-in-time Defect Prediction. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–12.

[20] Trisha Greenhalgh and Richard Peacock. 2005. Effectiveness and efficiency of search methods in systematic reviews of complex evidence: audit of primary sources. *Bmj* 331, 7524 (2005), 1064–1065.

[21] Yuchen Guo, Martin Shepperd, and Ning Li. 2018. Bridging effort-aware prediction and strong classification: a just-in-time software defect prediction study. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 325–326.

[22] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. 2011. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering* 38, 6 (2011), 1276–1304.

[23] Abram Hindle, Michael W Godfrey, and Richard C Holt. 2008. Reading beside the lines: Indentation as a proxy for complexity metric. In *2008 16th IEEE International Conference on Program Comprehension*. IEEE, 133–142.

[24] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. 2019. DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 34–45.

[25] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. CC2Vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 518–529.

[26] Seyedrebvar Hosseini, Burak Turhan, and Dimuthu Gunarathna. 2017. A systematic literature review and meta-analysis on cross project defect prediction. *IEEE Transactions on Software Engineering* 45, 2 (2017), 111–147.

[27] Qiao Huang, Xin Xia, and David Lo. 2017. Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 159–170.

[28] Qiao Huang, Xin Xia, and David Lo. 2019. Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction. *Empirical Software Engineering* 24, 5 (2019), 2823–2862.

[29] Hadi Jahanshahi, Dhanya Jothimani, Ayşe Başar, and Mucahit Cevik. 2019. Does chronology matter in JIT defect prediction? A Partial Replication Study. In *Proceedings of the Fifteenth International Conference on Predictive Models and*

Data Analytics in Software Engineering. 90–99.

[30] Tian Jiang, Lin Tan, and Sunghun Kim. 2013. Personalized defect prediction. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Ieee, 279–289.

[31] Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E Hassan. 2016. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering* 21, 5 (2016), 2072–2106.

[32] Yasutaka Kamei and Emad Shihab. 2016. Defect prediction: Accomplishments and future challenges. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, Vol. 5. IEEE, 33–45.

[33] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2012. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (2012), 757–773.

[34] Jonggu Kang, Duksan Ryu, and Jongmoon Baik. 2020. Predicting just-in-time software defects to reduce post-release quality costs in the maritime industry. *Software: Practice and Experience* (2020).

[35] Chaiyakarn Khanan, Worawit Luewichana, Krissakorn Pruktharathikoon, Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, Morakot Choetkiertikul, Chaiyong Ragkhitwetsagul, and Thanwadee Sunetnanta. 2020. JITBot: An explainable just-in-time defect prediction bot. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1336–1339.

[36] Sunghun Kim, E James Whitehead, and Yi Zhang. 2008. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering* 34, 2 (2008), 181–196.

[37] Barbara Kitchenham and Pearl Brereton. 2013. A systematic review of systematic review process research in software engineering. *Information and software technology* 55, 12 (2013), 2049–2075.

[38] Barbara Kitchenham and Stuart Charters. 2007. *Guidelines for performing systematic literature reviews in software engineering*. Technical Report EBSE-2007-01. School of Computer Science and Mathematics, Keele University.

[39] Barbara Kitchenham, O. Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. 2009. Systematic literature reviews in software engineering – A systematic literature review. *Information and Software Technology* 51, 1 (2009), 7 – 15. https://doi.org/10.1016/j.infsof.2008.09.009 Special Section - Most Cited Articles in 2002 and Regular Research Papers.

[40] Barbara Ann Kitchenham, David Budgen, and Pearl Brereton. 2015. *Evidence-based software engineering and systematic reviews*. Vol. 4. CRC press.

[41] Masanari Kondo, Daniel M German, Osamu Mizuno, and Eun-Hye Choi. 2020. The impact of context metrics on just-in-time defect prediction. *Empirical Software Engineering* 25, 1 (2020), 890–939.

[42] Ning Li, Martin Shepperd, and Yuchen Guo. 2020. A systematic review of unsupervised learning techniques for software defect prediction. *Information and Software Technology* (2020), 106287.

[43] Weiwei Li, Wenzhou Zhang, Xiuyi Jia, and Zhiqiu Huang. 2020. Effort-aware semi-supervised just-in-time defect prediction. *Information and Software Technology* 126 (2020), 106364.

[44] Zhiqiang Li, Xiao-Yuan Jing, and Xiaoke Zhu. 2018. Progress on approaches to software defect prediction. *IET Software* 12, 3 (2018), 161–175.

[45] Dayi Lin, Chakkrit Tantithamthavorn, and Ahmed E Hassan. 2021. The Impact of Data Merging on the Interpretation of Cross-Project Just-In-Time Defect Models. *IEEE Transactions on Software Engineering* (2021).

[46] Jinping Liu, Yuming Zhou, Yibiao Yang, Hongmin Lu, and Baowen Xu. 2017. Code churn: A neglected metric in effort-aware just-in-time defect prediction. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 11–19.

[47] Ruchika Malhotra. 2015. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing* 27 (2015), 504–518.

[48] Shane McIntosh and Yasutaka Kamei. 2017. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. *IEEE Transactions on Software Engineering* 44, 5 (2017), 412–428.

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

[50] Keita Mori and Osamu Mizuno. 2015. An Implementation of Just-in-Time Fault-Prone Prediction Technique Using Text Classifier. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, Vol. 3. IEEE, 609–612.

[51] Mathieu Nayrolles and Abdelwahab Hamou-Lhadj. 2018. CLEVER: combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 153–164.

[52] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. 2019. Fine-grained just-in-time defect prediction. *Journal of Systems and Software* 150 (2019), 22–36.

[53] Chanathip Pornprasit and Chakkrit Tantithamthavorn. 2021. JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction. *arXiv preprint arXiv:2103.07068* (2021).

[54] Lei Qiao and Yan Wang. 2019. Effort-aware and just-in-time defect prediction with neural network. *PloS one* 14, 2 (2019), e0211359.

[55] Sophia Quach, Maxime Lamothe, Bram Adams, Yasutaka Kamei, and Weiyi Shang. 2021. Evaluating the impact of falsely detected performance bug-inducing changes in JIT models. *Empirical Software Engineering* 26, 5 (2021), 1–32.

[56] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. 2013. Software fault prediction metrics: A systematic literature review. *Information and software technology* 55, 8 (2013), 1397–1418.

[57] Foyzur Rahman and Premkumar Devanbu. 2013. How, and why, process metrics are better. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 432–441.

[58] Gema Rodriguez-Perez, Meiyappan Nagappan, and Gregorio Robles. 2020. Watch out for extrinsic bugs! A case study of their impact in just-in-time bug prediction models on the OpenStack project. *IEEE Transactions on Software Engineering* (2020).

[59] Christoffer Rosen, Ben Grawi, and Emad Shihab. 2015. Commit Guru: analytics and risk prediction of software commits. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*. 966–969.

[60] Shivkumar Shivaji, E James Whitehead, Ram Akella, and Sunghun Kim. 2012. Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering* 39, 4 (2012), 552–569.

[61] Sadia Tabassum, Leandro L Minku, Danyi Feng, George G Cabral, and Liyan Song. 2020. An investigation of cross-project learning in online just-in-time software defect prediction. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 554–565.

[62] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. 2015. Online defect prediction for imbalanced data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 99–108.

[63] Alexander Tarvo, Nachiappan Nagappan, Thomas Zimmermann, Thirumalesh Bhat, and Jacek Czerwonka. 2013. Predicting risk of pre-release code changes with checkinmentor. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 128–137.

[64] Hailemelekot Demtse Tessema and Surafel Lemma Abebe. 2021. Enhancing Just-in-Time Defect Prediction Using Change Request-based Metrics. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 511–515.

[65] Yuli Tian, Ning Li, Jeff Tian, and Wei Zheng. 2020. How Well Just-In-Time Defect Prediction Techniques Enhance Software Reliability?. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 212–221.

[66] Parastou Tourani and Bram Adams. 2016. The impact of human discussions on just-in-time quality assurance: An empirical study on openstack and eclipse. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 189–200.

[67] Alexander Trautsch, Steffen Herbold, and Jens Grabowski. 2020. Static source code metrics and static analysis warnings for fine-grained just-in-time defect prediction. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 127–138.

[68] Zhou Xu, Kunsong Zhao, Tao Zhang, Chunlei Fu, Meng Yan, Zhiwen Xie, Xiaohong Zhang, and Gemma Catolino. 2021. Effort-Aware Just-in-Time Bug Prediction for Mobile Apps Via Cross-Triplet Deep Feature Embedding. *IEEE Transactions on Reliability* (2021).

[69] Meng Yan, Xin Xia, Yuanrui Fan, Ahmed E Hassan, David Lo, and Shanping Li. 2020. Just-in-time defect identification and localization: A two-phase framework. *IEEE Transactions on Software Engineering* (2020).

[70] Meng Yan, Xin Xia, Yuanrui Fan, David Lo, Ahmed E Hassan, and Xindong Zhang. 2020. Effort-aware just-in-time defect identification in practice: a case study at Alibaba. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1308–1319.

[71] Limin Yang, Xiangxue Li, and Yu Yu. 2017. Vuldigger: a just-in-time and cost-aware tool for digging vulnerability-contributing changes. In *GLOBECOM 2017-2017 IEEE Global Communications Conference*. IEEE, 1–7.

[72] Xinli Yang, David Lo, Xin Xia, and Jianling Sun. 2017. TLEL: A two-layer ensemble learning approach for just-in-time defect prediction. *Information and Software Technology* 87 (2017), 206–220.

[73] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. 2015. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 17–26.

[74] Xingguang Yang, Huiqun Yu, Guisheng Fan, Kai Shi, and Liqiong Chen. 2019. Local versus global models for just-in-time software defect prediction. *Scientific Programming* 2019 (2019).

[75] Xingguang Yang, Huiqun Yu, Guisheng Fan, and Kang Yang. 2020. A differential evolution-based approach for effort-aware just-in-time software defect prediction. In *Proceedings of the 1st ACM SIGSOFT International Workshop on Representation Learning for Software Engineering and Program Languages*. 13–16.

[76] Xingguang Yang, Huiqun Yu, Guisheng Fan, Kang Yang, and Kai Shi. 2019. An Empirical Study on Progressive Sampling for Just-in-Time Software Defect Prediction.. In *QuASoQ@ APSEC*. 12–18.

[77] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. 2016. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 157–168.

[78] Steven Young, Tamer Abdou, and Ayse Bener. 2018. A replication study: just-in-time defect prediction with ensemble learning. In *Proceedings of the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*. 42–47.

[79] Abubakar Zakari, Sai Peck Lee, Rui Abreu, Babiker Hussien Ahmed, and Rasheed Abubakar Rasheed. 2020. Multiple fault localization of software programs: A systematic literature review. *Information and Software Technology* (2020), 106312.

[80] Zhengran Zeng, Yuqun Zhang, Haotian Zhang, and Lingming Zhang. 2021. Deep just-in-time defect prediction: how far are we?. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 427–438.

[81] Wenzhou Zhang, Weiwei Li, and Xiuyi Jia. 2019. Effort-Aware Tri-Training for Semi-supervised Just-in-Time Defect Prediction. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 293–304.

[82] Kunsong Zhao, Zhou Xu, Meng Yan, Yutian Tang, Ming Fan, and Gemma Catolino. 2021. Just-in-time defect prediction for Android apps via imbalanced deep learning model. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. 1447–1454.

[83] Kunsong Zhao, Zhou Xu, Meng Yan, Lei Xue, Wei Li, and Gemma Catolino. 2021. A compositional model for effort-aware Just-In-Time defect prediction on android apps. *IET Software* (2021).

[84] K. Zhao, Z. Xu, T. Zhang, Y. Tang, and M. Yan. 2021. Simplified Deep Forest Model Based Just-in-Time Defect Prediction for Android Mobile Apps. *IEEE Transactions on Reliability* (2021), 1–12. https://doi.org/10.1109/TR.2021.3060937

[85] Kun Zhu, Nana Zhang, Shi Ying, and Dandan Zhu. 2020. Within-project and cross-project just-in-time defect prediction based on denoising autoencoder and convolutional neural network. *IET Software* 14, 3 (2020), 185–195.

[86] Xiaoyan Zhu, Binbin Niu, E James Whitehead Jr, and Zhongbin Sun. 2018. An empirical study of software change classification with imbalance data-handling methods. *Software: Practice and Experience* 48, 11 (2018), 1968–1999.