An Analytical Review of Bug Introduction and Prediction in Software Engineering: Foundations, Evolution, and Future Horizons

Introduction

The persistent challenge in software engineering is not merely the creation of functional systems, but their sustained maintenance and evolution in the face of inevitable defects. The lifecycle of these defects, from their inception to their eventual discovery and resolution, forms a critical area of study for both industrial practitioners and academic researchers. This review examines the two fundamental pillars of this domain: the retrospective identification of a bug's origin, known as **bug introduction**, and the proactive forecasting of future defects, known as **bug prediction**. These are not disparate fields of inquiry but rather a deeply interconnected ecosystem, where the accuracy and validity of the former directly govern the potential and utility of the latter.

This report traces the evolutionary arc of this ecosystem, beginning with the foundational, heuristic-based algorithms for mining software repositories. It follows the field's progression through successive refinements, critiques, and paradigm shifts, culminating in the current state-of-the-art, which leverages sophisticated machine learning, deep learning, and even large language models. This trajectory mirrors a broader maturation within empirical software engineering, reflecting a move towards more data-driven, methodologically rigorous approaches. Throughout this evolution, persistent challenges related to reproducibility, credibility, and practical utility have acted as catalysts for innovation, shaping the research landscape and pushing the community toward greater transparency and more robust scientific practices. The following sections will deconstruct the key advancements, methodological debates, and future horizons in both bug introduction and prediction, culminating in a synthesis that highlights their critical interdependencies.

Section 1: Unearthing Bug Origins: The SZZ Algorithm and Its Legacy

The ability to automatically identify the specific commit that introduced a bug is foundational to a vast array of empirical software engineering studies. The de facto standard for this task is the SZZ algorithm. This section deconstructs the SZZ algorithm, tracing its intellectual lineage from its inception, through numerous critiques and refinements, to its modern-day implementations. This history reveals an ongoing tension between the algorithm's undeniable utility and its inherent, persistent limitations.

1.1 The Foundational Heuristic: Śliwerski, Zimmermann, and Zeller (2005)

The SZZ algorithm, named after its originators Śliwerski, Zimmermann, and Zeller, was introduced as a method to identify "fix-inducing changes".[1] The central premise is that by analyzing a bug-fixing commit, one can trace the history of the modified code to find the earlier change that necessitated the fix.[3]

The original algorithm operates in a two-phase process, designed for version control systems like CVS and bug trackers like Bugzilla [3]:

1. **Phase 1: Linking Bug Reports to Fixes.** The first step is to establish a link between a bug report in an issue tracker and the specific commit in the version control system that resolves it. This is typically accomplished heuristically by parsing commit log messages for patterns that reference bug IDs, such as "Fixes #123" or "JENKINS-456," using regular expressions.[3]
2. **Phase 2: Tracing Bug-Introducing Commits.** Once a bug-fixing commit is identified, the algorithm examines the lines of code that were deleted or modified. For each of these lines, it uses the version control system's annotation command (e.g., cvs annotate or its modern equivalent, git blame) to determine which commit last altered that line prior to the fix. These identified prior commits are considered candidates for being the bug-introducing change.[5]

The initial application envisioned for this technique was to discover patterns in problematic changes, such as whether changes made on Fridays or larger changes were more likely to induce fixes, thereby enabling a deeper understanding of the development process.[1]

1.2 Early Critiques and Refinements (2006-2008)

Almost immediately after its introduction, the research community identified significant precision issues with the original SZZ heuristic. The algorithm could be easily misled by changes that were not semantically related to the program's logic, and its reliance on simple line-based annotation proved to be a critical weakness.[5]

Two key improvements emerged to address these shortcomings:

- **Annotation Graphs and Filtering (Kim et al., 2006):** Kim, Zimmermann, and others proposed an enhanced version, often referred to as AG-SZZ, to combat the problem of false positives. This approach introduced the use of annotation graphs to better track line history and, crucially, incorporated heuristics to filter out non-semantic or "cosmetic" changes, such as modifications to comments, whitespace, and code formatting.[7] This filtering was a significant step forward, with the authors reporting that their algorithms could remove between 38% and 51% of false positives and 14% to 15% of false negatives compared to the original SZZ approach.[7]
- **Line Number Mapping (Williams & Spacco, 2008):** A major limitation of annotation graphs is their imprecision when faced with large, contiguous blocks of modified code, where the ancestry of any given line becomes ambiguous.[5] To solve this, Williams and Spacco proposed replacing the annotation graph technique with a more precise "line-number mapping" approach.[8] Based on earlier work by Canfora et al., this method uses an edit distance algorithm, such as the normalized Levenshtein distance, to calculate the similarity between lines in successive revisions. By finding the best match, it can track the identity of a single line more accurately as it evolves over time.[5] Critically, this work also highlighted a fundamental, unverified assumption underlying all SZZ variants: that the lines identified by the algorithm are, in fact, the true source of the bug.[8]

1.3 The Reproducibility and Credibility Dilemma

As the SZZ algorithm became more widely adopted, a new, more systemic problem emerged, not with the algorithm's technical details, but with its implementation and use within the research community. A landmark 2018 systematic literature review by Rodriguez-Perez, Robles, and Gonzalez-Barahona exposed a potential crisis of credibility in the field.[11] Their analysis of 187 academic publications that used an SZZ variant revealed several alarming trends:

- **Lack of Public Implementations:** Researchers were overwhelmingly creating their own private, from-scratch implementations of SZZ instead of reusing or building upon existing work. This led to a proliferation of unverified and potentially buggy tools.[5]

- **Poor Reproducibility:** The vast majority of studies were not reproducible. Only 43 of 187 papers offered any form of replication package, and a mere 24 (13%) provided both a package and a detailed methodology. A full 39% of studies provided neither, making verification of their results impossible.[11]
- **Under-reporting of Limitations:** Despite the well-documented flaws of SZZ, a staggering 94 studies (50.3%) failed to mention any of its limitations as a threat to the validity of their results.[11]

The direct consequence of these findings was a challenge to the credibility of a significant body of empirical software engineering research. The very tool used to generate the ground-truth data (i.e., identifying which commits are buggy) was itself of unknown quality and its application was opaque.[4] Subsequent analysis confirmed that SZZ's performance is only moderate, with F-scores ranging from

0.44 to 0.77 and true positive rates not exceeding 63%.[13]

In direct response to this call for improved reproducibility, Borg et al. released **SZZ Unleashed** in 2019.[5] This public, open-source Java implementation of the SZZ algorithm, which incorporates the line-number mapping improvements from Williams and Spacco, was created to provide a standardized, tested, and extensible tool for the community. Its goal was to establish a trusted baseline, mitigate the "reinventing the wheel" problem, and allow research to build upon a more solid and transparent foundation.[5]

1.4 The Modern SZZ Ecosystem: Tackling Persistent Flaws

With a more stable and open foundation, recent research has focused on tackling some of the most persistent and nuanced flaws in the SZZ approach.

- **The Refactoring Problem:** One of the most significant sources of false positives for SZZ is code refactoring. When code is moved or renamed, SZZ may incorrectly blame the original commit that created the code, even though the change was behavior-preserving.[14] To address this,
  Refactoring-Aware SZZ (RA-SZZ) variants were developed. These tools integrate refactoring detection engines, such as RefDiff and RMiner, to identify and exclude refactoring operations from the blame analysis.[15] However, their

effectiveness is constrained by the limited number of refactoring types these underlying tools can reliably detect.[15]

- **The "Ghost Commit" and Context Problem:** The traditional SZZ assumption is that a fix involves deleting or modifying the faulty lines. This is not always the case. For instance, a fix might only add new code (e.g., adding a missing null-pointer check), leaving no deleted lines for SZZ to trace. These untraceable fixes are a form of "ghost commit".[17] To handle these cases, approaches like **Sem-SZZ** have been proposed, which trace back from *unmodified lines near the added lines* and use data flow analysis to improve precision.[17] Other work argues that SZZ ignores valuable context available in commit messages and bug report discussions, which could help identify related files and disambiguate complex, tangled commits.[19]
- **The Deep Learning and LLM Frontier:** The most recent evolution in the SZZ ecosystem represents a paradigm shift from syntactic tracing to semantic understanding.
  - **NeuralSZZ** employs a heterogeneous graph attention network to learn the semantic relationships between added and deleted lines in a fix. It then ranks the deleted lines by their likelihood of being the true root cause, aiming to improve SZZ's precision by feeding it only the most relevant input.[9]
  - **LLM4SZZ** leverages the power of Large Language Models (LLMs). Instead of syntactic tracing, it uses an LLM's ability to comprehend the natural language bug description and the semantics of the code changes to assess a set of candidate commits and identify the most likely bug-introducing one.[7] This marks a significant move towards automated reasoning in bug attribution.

The history of the SZZ algorithm serves as a compelling case study in the maturation of empirical software engineering. It began with a clever, useful, but ultimately flawed heuristic. Its widespread use revealed technical limitations, which the community addressed with targeted improvements. However, this uncoordinated adoption created a higher-level, systemic problem of non-reproducibility that threatened the field's credibility. The community responded again, this time with a socio-technical solution: a shared, open-source tool to establish a common baseline. This stable foundation has, in turn, enabled researchers to tackle the more subtle and difficult semantic challenges that were previously obscured by the noise of inconsistent and private implementations. This evolutionary arc provides a valuable roadmap for the responsible development and deployment of future research artifacts in software engineering.

Section 2: Forecasting Defects: The Evolution of Bug Prediction Models

Transitioning from the retrospective analysis of bug origins, this section examines the forward-looking discipline of bug prediction. The goal of these models is to forecast which parts of a software system are most likely to contain defects, thereby allowing for a more efficient allocation of limited quality assurance resources, such as code review and testing effort. This section details the key components of these models, the dominant paradigms for their application, and the critical methodological considerations required to produce valid and reliable results.

2.1 The Predictor's Toolkit: A Taxonomy of Metrics

The performance of any bug prediction model is contingent on the quality and predictive power of its input features, or metrics. The field has evolved from using simple, static code properties to incorporating more dynamic and semantically rich predictors that capture the nuances of the development process.

- **Code Churn Metrics:** One of the earliest and most enduring concepts in defect prediction is that the volume of code change, or "churn," is correlated with defect density. The seminal work by Nagappan and Ball (2005) demonstrated a crucial insight: *relative* code churn measures (e.g., lines of code added divided by total lines of code) are significantly better predictors of defect density than absolute churn measures.[21] These metrics, which capture the magnitude of change relative to the size of a component, remain a staple in modern prediction models and are often used as a baseline for comparison.[5]
- **Code Complexity Metrics:** The Chidamber & Kemerer (CK) suite of object-oriented metrics is a classic set of predictors that quantify various dimensions of code complexity, coupling, and cohesion.[25] The suite includes metrics such as Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Objects (CBO), Response for a Class (RFC), and Lack of Cohesion in Methods (LCOM). Numerous studies have investigated their effectiveness, often finding that metrics related to coupling (CBO, RFC) and complexity (WMC) are strong indicators of fault-proneness, while others like DIT and NOC may be less reliable.[25]
- **Change Coupling Metrics:** Going beyond the complexity of a single module,

change coupling metrics capture the implicit, evolutionary dependency between software artifacts that are frequently modified together in the same commit.[28] As proposed by D'Ambros et al. (2009), high change coupling can be a symptom of architectural decay or hidden dependencies, and it has been shown to correlate strongly with the presence of software defects, in some cases outperforming traditional complexity metrics.[28] The inclusion of coupling features in the illustrative study of the SZZ Unleashed paper demonstrates their continued relevance in modern prediction models.[5]

- **Process and Developer-Related Metrics:** Recognizing that software development is a human activity, researchers have incorporated metrics that capture aspects of the development process itself. These include the number of developers who have worked on a file, the overall experience of a committer, and the entropy or distribution of changes across the system.[5] More recent work has explored the concept of developer focus, providing evidence that changes made by developers who are scattered across many different parts of the codebase are more likely to be bug-prone than those made by focused developers.[32]

2.2 The Just-in-Time (JIT) Paradigm Shift (Kamei et al., 2013)

A pivotal development in the field was the shift from traditional, coarse-grained defect prediction to the more practical and actionable paradigm of Just-in-Time (JIT) defect prediction.[31]

Traditional models typically operated at the file or module level, predicting which components were likely to be buggy in a future release. While statistically sound, these predictions suffered from a "utility gap." A developer being told that a 10,000-line file is "buggy" receives little actionable guidance.[31]

JIT defect prediction, as popularized by Kamei et al. (2013), changes the unit of analysis from the file to the individual *change* or *commit*. The model aims to predict, at check-in time, whether a specific commit is likely to introduce a defect. This provides immediate, fine-grained feedback to the developer while the context of the change is still fresh in their mind, making it far more practical for integration into a continuous integration/continuous delivery (CI/CD) workflow.[31]

The large-scale empirical study by Kamei et al. across eleven open-source and commercial projects established the viability of the JIT approach, achieving an average accuracy of 68% and recall of 64%.[31] They also introduced the important concept of effort-aware evaluation, demonstrating that by focusing review effort on the top 20% of changes flagged as most risky, teams could identify 35% of all defect-inducing changes.[31] The JIT paradigm has since become the dominant approach in the field, framing the context for most modern research, including the illustrative study in the SZZ Unleashed paper and recent work on advanced machine learning models.[5]

2.3 Methodological Imperatives in Model Evaluation

The focus on practical utility brought about by the JIT paradigm also forced a re-examination of the methodologies used to evaluate prediction models. It became clear that standard machine learning practices were often ill-suited to the unique characteristics of software engineering data, leading to inflated and misleading performance claims.

- **The Time-Sensitive Evaluation Problem:** Software repository data is inherently time-ordered. A commit made today cannot be influenced by a commit made tomorrow. However, standard evaluation techniques like k-fold cross-validation violate this temporal dependency by randomly shuffling data, effectively allowing a model to be trained on "future" data to predict the "past." This leads to overly optimistic performance estimates that do not reflect how a model would perform in a real-world, forward-looking deployment.[5] The study in the SZZ Unleashed paper empirically confirms this, showing that F1 scores from a traditional cross-validation setup were consistently higher than those from a more realistic "Online Change Classification" setup that respects the timeline of commits.[5]
- **The Class Imbalance Problem:** Defect datasets are naturally and severely imbalanced. For example, in the Jenkins dataset analyzed by Borg et al., only 3.6% of commits were found to be bug-introducing.[5] When trained on such data, a naive classifier can achieve high accuracy simply by always predicting the majority class (i.e., "not buggy"), rendering it useless. It is therefore essential to employ techniques to handle this imbalance. Methods like oversampling the minority class (e.g.,
  SMOTE, Synthetic Minority Over-sampling Technique) or undersampling the majority class are critical for building a classifier that can successfully identify the rare, positive instances.[5] The SZZ Unleashed study demonstrates this clearly, showing that using SMOTE dramatically improved recall and overall F1 score

compared to a baseline approach with no sampling strategy.[5]

- **The Generalization Challenge (Cross-Project Defect Prediction):** A key desire for practitioners is a model that works "out of the box" for new projects that lack sufficient historical data for training. This has motivated research into **Cross-Project Defect Prediction (CPDP)**, where a model is trained on one or more source projects and applied to a different target project. However, empirical studies have consistently shown that this is a difficult challenge. The performance of CPDP models is often poor, and a model that works well in one direction (e.g., Project A predicting for Project B) may not work well in reverse.[34] This highlights the project-specific nature of defect patterns and the difficulty of building universally applicable models.
- **The Need for Rigorous Reporting:** A systematic literature review by Hall et al. (2012) emphasized that the credibility and utility of prediction studies are often hampered by incomplete reporting. Without comprehensive details on the context (e.g., project characteristics), methodology (e.g., data cleaning, feature selection, validation strategy), and performance metrics, it is impossible for other researchers to reliably analyze, compare, or build upon the results.[39]

The evolution of bug prediction modeling illustrates a clear trajectory from demonstrating statistical possibility to pursuing practical, developer-centric utility. This journey has been marked by a shift in granularity from files to commits, a deeper understanding of the importance of evolutionary and process-based metrics, and, critically, the development of more rigorous evaluation methodologies that reflect real-world deployment scenarios. The benchmark for a successful new prediction model is no longer just high accuracy on a static dataset, but also its timeliness, actionability, and the trustworthiness of its reported performance.

Section 3: Synthesis, Interdependencies, and Future Trajectories

The preceding sections have traced the parallel, yet deeply intertwined, evolutionary paths of bug introduction and bug prediction research. This final section weaves these threads together, making explicit the symbiotic and often fraught relationship between these two domains. It then looks to the future, analyzing the impact of the latest technological shifts in artificial intelligence and identifying the most pressing open questions and research gaps for the community.

3.1 The Symbiotic Cycle: How SZZ Underpins Defect Prediction

The most critical realization from this review is the foundational dependency of supervised defect prediction on bug attribution algorithms like SZZ. The entire field of building classifiers to predict buggy commits is predicated on having a labeled dataset for training, and it is the SZZ algorithm that provides these crucial labels ("buggy" or "clean").[4]

This relationship creates a direct and unavoidable pipeline where the quality of the output of one process becomes the quality of the input for the next. This leads to a classic "garbage in, garbage out" scenario. The documented limitations and inaccuracies of the SZZ algorithm—its susceptibility to noise from refactoring, its inability to handle ghost commits, and its moderate precision and recall—are not isolated problems for attribution research.[4] They represent a direct and significant threat to the validity of every defect prediction model that relies on SZZ-generated data. If the training labels are noisy or incorrect, the resulting prediction models risk learning spurious correlations and failing to capture the true characteristics of defect-inducing changes.

The practical impact of this dependency is substantial. The illustrative JIT prediction study in the SZZ Unleashed paper concluded that its achieved F1 score of approximately 0.15 was likely insufficient for practical industrial use. The authors explicitly speculated that a higher-quality, manually annotated training set—that is, one with more accurate labels—could lead to a more useful model.[5] This hypothesis was empirically supported by the work of Costa et al., who demonstrated that the accuracy of SZZ implementations improves by approximately 40% when they are provided with a pre-validated, cleaner set of bug-fixing changes as input.[15] This confirms that improving the foundational step of bug attribution is a prerequisite for advancing the practical utility of bug prediction.

3.2 The Next Frontier: Deep Learning and Large Language Models (2020-2025)

The field is currently in the midst of another major technological shift, driven by advancements in deep learning and, more recently, Large Language Models (LLMs). This is reshaping the approaches to both bug introduction and prediction.

The evolution of prediction models has progressed from traditional machine learning algorithms like Logistic Regression and Random Forest, which rely on hand-crafted features, to deep learning models.[44] Architectures based on Convolutional Neural Networks (CNNs) or Long Short-Term Memory (LSTMs) can automatically learn relevant features from raw data artifacts like commit messages and code diffs, reducing the dependency on manual feature engineering.[35]

The most recent and disruptive trend is the application of LLMs.

- **For Defect Prediction:** Researchers are fine-tuning LLMs to predict defects directly from code. These models have shown competitive performance but face new challenges, such as adapting to the rapid evolution of codebases without suffering from "catastrophic forgetting," where the model loses previously learned knowledge upon learning new information.[46]
- **For Improving SZZ:** As discussed previously, LLMs are also being applied to improve the SZZ algorithm itself. Approaches like LLM4SZZ aim to replace syntactic tracing with semantic comprehension, using the model's understanding of the bug report and code to make a more reasoned judgment about which commit introduced the defect.[19]

This new wave of technology introduces novel research challenges. For instance, a recent study highlights that while modern JIT models are becoming more accurate, their probabilistic outputs are often poorly calibrated.[38] This means a model might predict a commit is 90% likely to be buggy, but in reality, such predictions are only correct 60% of the time. Poor calibration undermines the trustworthiness of a model's confidence scores, which is a critical issue for practical adoption where developers need to prioritize which warnings to investigate.

3.3 Open Problems and Research Gaps

Despite decades of research, several fundamental challenges and research gaps remain.

- **Improving SZZ Accuracy:** The core problem of accurately and reliably identifying bug-introducing commits is still far from solved. The persistent issues surrounding refactoring operations, ghost commits (especially those that do not modify the same files as the fix), and the need for deeper semantic context remain active and important areas of research.[15]
- **Bridging the Practicality Gap:** A significant gap persists between the performance of prediction models reported in academic studies and their actual

utility in industrial practice. The modest F1 scores often seen in realistic, time-sensitive evaluations [5] and the formidable challenge of cross-project prediction [41] indicate that substantial work is needed to make these tools robust, reliable, and trustworthy enough for integration into daily developer workflows.

- **Fine-Grained Prediction:** While JIT prediction at the commit level was a major step forward in practicality, some research has attempted to provide even finer-grained predictions at the method or line level. However, a recent replication study demonstrated that when method-level models are evaluated using a realistic, time-sensitive strategy, their performance drops dramatically to little better than a random classifier. This suggests that achieving useful prediction at such a fine granularity is still an open and difficult challenge.[35]
- **Fairness and Interpretability:** As machine learning models are increasingly deployed to assess code quality, they are, by extension, assessing the work of developers. This raises critical socio-technical questions of fairness, bias, and interpretability. It is essential to ensure that prediction models do not unfairly penalize certain developers or coding styles and that their predictions are explainable to the developers who must act on them. Early work on fairness testing for machine learning systems in software engineering is an important first step in this emerging and vital research direction.[48]

Conclusion

This review has charted the parallel and co-dependent evolution of research into bug introduction and bug prediction. The journey reveals a clear maturation process within empirical software engineering, moving from a reliance on simple, brittle heuristics toward an embrace of more methodologically rigorous, data-driven, and AI-powered approaches. The history of the SZZ algorithm—from a simple script, to a complex ecosystem of variants wrestling with issues of reproducibility and accuracy, to its latest incarnation augmented by LLMs—is a testament to this progression. Simultaneously, the field of bug prediction has transformed from a theoretical statistical exercise on static files to a practical, developer-centric endeavor focused on providing just-in-time feedback on individual commits, with an increasing emphasis on realistic evaluation and trustworthiness.

The central theme emerging from this analysis is the undeniable symbiosis between these two domains: the quality of bug prediction is fundamentally constrained by the quality of bug introduction data. The future of the field therefore depends on progress on multiple fronts. First, resolving the foundational challenge of accurate and reliable bug attribution remains paramount. Second, efforts must continue to bridge the gap between academic models and industrial utility, focusing on robustness, generalization, and the emerging challenges of model calibration and interpretability. Finally, as AI becomes more deeply embedded in the software development lifecycle, the community must proactively address the critical socio-technical issues of fairness and transparency to ensure these powerful tools are deployed responsibly and effectively.

Appendix: A Chronological and Annotated Literature Review

The following table provides a chronological summary of the key literature discussed in this review. It is designed to serve as a high-value reference for researchers, detailing the core contributions, findings, and limitations of each seminal work.

Year	Authors	Title	Venue	Link / DOI	Keywords	Abstract	Findings	Conclusion	Room for Improvements / Limitations	Remarks
2005	Śliwerski, J., Zimmermann, T., Zeller, A.	When do changes induce fixes?	MSR '05	Link	fix-inducing changes, version control, bug database, metrics	Analyzes CVS archives to find "fix-inducing changes" by linking version archives to bug databases. Found that such changes show distinct patterns related to size and day of the week.	Fix-inducing changes can be automatically located by linking CVS and Bugzilla. These changes exhibit patterns; for example, changes made on certain days of the week are more likely to induce fixes.	The history of a software system contains valuable information for identifying problematic changes. This automatic identification can be used to filter changes for other analyses, like architecture extraction.	The initial approach is a heuristic. It relies on parsing commit messages and using cvs annotate, which has known precision issues with large or complex changes. The validity of the core assumption (blamed lines = bug cause) is not verified.	The seminal paper that introduced the SZZ algorithm. It established the foundational technique for a new class of mining software repositories (MSR) studies by providing a method to label commits as bug-introducing. 1
2005	Nagappan, N., Ball, T.	Use of Relative Code Churn Measures to Predict System Defect Density	ICSE '05	10.1145/1062455.1062514	code churn, defect density, prediction, regression models	Presents a technique for predicting system defect density using relative code churn measures. Shows that while absolute churn is a poor predictor, relative churn is highly predictive.	Relative code churn measures (e.g., churned LOC / total LOC) are highly predictive of defect density, while absolute measures are not. The proposed metric suite could discriminate between faulty and non-faulty binaries with 89% accuracy in a case study on Windows Server 2003.	Relative code churn measures are valid early indicators of system defect density and can be used to build effective prediction models.	The study was performed on a single, large-scale proprietary system (Windows). The generalizability to other types of projects (e.g., open source) was not explored. The definition of "defect" and the data collection process are specific to Microsoft's internal infrastructure.	Foundational work in defect prediction that established the importance of using relative change metrics over absolute ones. Code churn has since become a standard baseline feature in most defect prediction studies. 21
2006	Kim, S., Zimmermann, T., Pan, K., Whitehead, E.J.	Automatic identification of bug-introducing changes	ASE '06	10.1109/ASE.2006.23	bug-introducing changes, annotation graphs, false positives, false negatives	Presents algorithms to more accurately identify bug-introducing changes by removing false positives/negatives using annotation graphs, ignoring non-semantic changes, and validating fixes manually.	The proposed algorithms, which filter out cosmetic changes (e.g., comments, whitespace) and use annotation graphs for more precise tracing, can remove 38%-51% of false positives and 14%-15% of false negatives compared to the original SZZ algorithm.	By systematically filtering noise and using more sophisticated tracing, the accuracy of identifying bug-introducing changes can be significantly improved, enabling more reliable downstream analyses.	The approach still relies on heuristics for filtering and annotation graphs, which can be imprecise. It does not address fundamental issues like refactoring or fixes that only add code. The manual validation was done by researchers, not the original developers.	A critical early improvement to SZZ (often called AG-SZZ). It was the first major attempt to systematically address the "noise" in bug-fixing commits, making the output of SZZ more precise and reliable. 7

2008	Williams, C., Spacco, J.	SZZ Revisited: Verifying when changes induce fixes	DEFECTS '08	Link	SZZ, annotation graphs, line-number maps, verification	Outlines improvements to SZZ by replacing imprecise annotation graphs with line-number maps and using a syntax-aware diff tool. Begins the process of verifying if SZZ-identified lines are truly bug-introducing.	Annotation graphs are imprecise for large blocks of modified code. Line-number maps, based on edit distance, provide more precise tracking of source lines across revisions. Using a syntax-aware diff tool helps ignore non-semantic changes like comments.	The precision of SZZ can be improved with better line tracking techniques. However, the fundamental question of whether the SZZ-identified change is the true source of a bug remains an open and critical area for verification.	The line-mapping approach can still fail (false negatives) if edits to a line are very large. The verification of SZZ's output was preliminary and not exhaustive.	A key refinement of SZZ that directly addressed the imprecision of annotation graphs. The line-number mapping approach has been adopted by modern implementations like SZZ Unleashed. Crucially, it was among the first to explicitly question the ground-truth validity of SZZ's output. 8
2009	D'Ambros, M., Lanza, M., Robbes, R.	On the Relationship Between Change Coupling and Software Defects	WCRE '09	10.1109/WCRE.2009.19	change coupling, software defects, bug prediction, complexity metrics	Analyzes the relationship between change coupling (artifacts that change together) and software defects on three large systems, investigating if this correlation can improve bug prediction models.	Change coupling correlates with the number of defects, and this correlation is stronger than that of traditional object-oriented complexity metrics (like the CK suite). The performance of defect prediction models can be improved by including change coupling information.	Change coupling is a tangible indicator of design issues that lead to defects. It is a valuable metric that provides information beyond standard complexity metrics and should be considered when building defect prediction models.	The study was limited to three open-source Java systems, which may limit the generalizability of the findings to industrial or other language contexts. The bug linking process relies on conventions in commit messages.	Introduced and validated "change coupling" as a powerful evolutionary metric for defect prediction. It shifted focus from the properties of a single file to the relationships between files as they evolve, capturing a key aspect of system-level design decay. 28
2012	Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S.	A Systematic Literature Review on Fault Prediction Performance in Software Engineering	TSE	10.1109/TSE.2011.103	fault prediction, systematic literature review, performance, methodology	A systematic review of 208 fault prediction studies to investigate how context, independent variables, and modeling techniques influence model performance.	Models that perform well tend to use simple techniques (e.g., Naive Bayes, Logistic Regression) and combinations of variables with feature selection. Many studies lack sufficient methodological detail to be reliably analyzed or compared.	The methodology used to build and evaluate models is highly influential on performance. More rigor and comprehensive reporting of context and methodology are needed for the field to mature and for users to have confidence in prediction models.	The review's search terms may have missed some relevant studies. The synthesis is based on only 36 of 208 papers that met the quality criteria, and even these may have unreported issues (e.g., with imbalanced data).	A landmark systematic review that highlighted the methodological weaknesses and lack of standardized reporting in the defect prediction field. Its call for more rigor influenced subsequent research to adopt more transparent and reliable evaluation practices. 39
2013	Kamei, Y., Shihab, E., Adams, B., et al.	A Large-Scale Empirical Study of Just-in-Time Quality Assurance	TSE	10.1109/TSE.2012.70	just-in-time prediction, defect prediction, mining software repositories	Proposes and evaluates "Just-In-Time (JIT) Quality Assurance," a paradigm that predicts defect-prone changes (commits) instead of files, providing more actionable feedback to developers.	JIT models can predict defect-inducing changes with an average accuracy of 68% and recall of 64%. By focusing on the 20% of changes deemed most risky, 35% of all defect-inducing changes can be identified, demonstrating an effort-reducing approach.	JIT Quality Assurance is a viable and more practical alternative to traditional file-level defect prediction. It provides timely, fine-grained feedback that can be integrated into developer workflows to improve quality and reduce costs.	The study relies on the SZZ algorithm to identify defect-inducing changes, inheriting its limitations. The measure of review effort (lines of code) is a proxy. The findings may not generalize to all projects, though a diverse set of 11 systems was used.	A paradigm-shifting paper that moved the focus of defect prediction from coarse-grained files to fine-grained, actionable changes. It established the JIT prediction model and the concept of effort-aware evaluation, which have become standard in the field. 31
2018	Rodriguez-Perez, G., Robles, G., Gonzalez-Barahona, J.M.	Reproducibility and Credibility in Empirical Software Engineering: A Case Study based on a Systematic Literature Review of the use of the SZZ algorithm	IST	10.1016/j.infsof.2018.03.009	reproducibility, credibility, SZZ algorithm, systematic literature review	A systematic review of 187 studies using the SZZ algorithm to assess the state of reproducibility and credibility in empirical software engineering.	Reproducibility is poor; only 13% of studies provide both a replication package and detailed methods. Over 50% of studies do not mention SZZ's known limitations. Researchers tend to implement their own SZZ versions rather than use improved ones.	There is significant room for improvement in reproducibility and credibility in ESE. The research community needs to prioritize open science practices, including sharing tools and data, to increase the reliability of research results.	The study is a case study of SZZ, so results may not generalize to all of ESE. The assessment of reproducibility is based on the availability of artifacts, not on actually reproducing the studies.	A critical and influential paper that exposed a systemic "reproducibility crisis" in a major area of MSR. Its findings directly motivated the creation of open-source tools like SZZ Unleashed and raised community awareness about the importance of open and transparent research. 11
2019	Borg, M., Svensson, O., Berg, K., Hansson, D.	SZZ Unleashed: An Open Implementation of the SZZ Algorithm	ArXiv	1903.01742	SZZ, defect prediction, mining software repositories, open source	Presents SZZ Unleashed, an open-source Java implementation of the SZZ algorithm, created in response to calls for improved reproducibility. Includes an illustrative study on JIT bug prediction for Jenkins.	The paper provides a public, open implementation of SZZ incorporating line-number mapping. The illustrative JIT prediction study on Jenkins data shows that oversampling (SMOTE) is essential for imbalanced data and that time-sensitive evaluation is more realistic than cross-validation.	The lack of a public SZZ tool hampers reproducibility and leads to wasted effort. SZZ Unleashed provides a community resource to address this. The illustrative study confirms key methodological findings for JIT prediction but achieves modest accuracy (F1 score ~0.15).	The JIT prediction accuracy achieved is not sufficient for practical use, possibly due to SZZ limitations or the need for better features. The empirical study is illustrative and not an exhaustive analysis of JIT prediction.	A direct and practical response to the reproducibility crisis identified by Rodriguez-Perez et al. It provides a much-needed standardized tool for the community, establishing a common baseline for future SZZ-based research. 5

2019	Costa, D. A., et al.	Revisiting and Improving SZZ Implementations	ESEM '19	10.1109/ESEM.2019.8870177	SZZ algorithm, refactoring, bug-introducing change	Re-evaluates and improves SZZ implementations (RA-SZZ) using the Defects4J dataset. Investigates the impact of refactoring and proposes improvements using a better refactoring detection tool (RMiner).	SZZ implementations are ~40% more accurate when given a pre-validated, cleaner set of bug-fixes as input. The improved RA-SZZ* is more precise, and a median of 44% of its identified bug-introducing lines are correct. Undetected refactorings remain a major source of error.	Preprocessing the input for SZZ can significantly improve accuracy. Detecting refactoring operations is crucial for SZZ precision, but current tools are still limited. There is still much room for improvement in accurately identifying bug-introducing changes.	The study is limited to the Defects4J dataset. The refactoring detection tools used still cannot detect all types of refactoring. Manual analysis is subject to researcher bias.	A rigorous study that quantifies the impact of both input data quality and refactoring on SZZ's performance. It provides strong evidence that improving the data fed into SZZ is as important as improving the algorithm itself. 15
2023	Feng, B., et al.	NeuralSZZ: A Neural-based SZZ Algorithm	ASE '23	10.1109/ASE56229.2023.00118	SZZ Algorithm, Deep Learning, Graph Attention Network	Proposes NeuralSZZ, a deep learning approach that uses a heterogeneous graph attention network to rank deletion lines in a bug-fix by their likelihood of being the root cause, aiming to improve SZZ precision.	NeuralSZZ outperforms baselines in identifying the root cause of bugs. Using only the top-ranked deletion lines as input to the SZZ algorithm significantly improves the F1-score (by 40.7%) and precision (by 121.8%) compared to previous SZZ variants.	Static methods for improving SZZ have limitations. A deep learning approach that understands the semantic relationships between code changes can more effectively identify the true root cause of a bug, leading to more precise bug-introducing commit identification.	The approach relies on a manually annotated dataset for training, which is costly to create. The study was limited to Java projects. The model is complex compared to traditional SZZ heuristics.	Represents the state-of-the-art in improving SZZ, shifting from syntactic heuristics and static analysis to deep learning-based semantic analysis. It demonstrates the potential of AI to address the core precision problems of SZZ. 9

Works cited

1. When Do Changes Induce Fixes? - CiteSeerX, accessed June 23, 2025, https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=6688ea87c6a6d0a93594bbb988b080e1b7e88393
2. When do changes induce fixes? - ResearchGate, accessed June 23, 2025, https://www.researchgate.net/publication/221656991_When_do_changes_induce_fixes
3. When Do Changes Induce Fixes?, accessed June 23, 2025, https://www.st.cs.uni-saarland.de/papers/msr2005/msr2005.pdf
4. When do changes induce fixes? On Fridays - ResearchGate, accessed June 23, 2025, https://www.researchgate.net/publication/248427624_When_do_changes_induce_fixes_On_Fridays
5. SZZ unleashed.pdf
6. Identifying Defect-Inducing Changes in Visual Code - Abram Hindle, accessed June 23, 2025, https://softwareprocess.es/pubs/eng2023ICSME-SZZ-visual-code.pdf
7. Automatic Identification of Bug-Introducing Changes | Request PDF, accessed June 23, 2025, https://www.researchgate.net/publication/220884078_Automatic_Identification_of_Bug-Introducing_Changes
8. SZZ revisited: verifying when changes induce fixes - ResearchGate, accessed June 23, 2025,

https://www.researchgate.net/publication/220854597_SZZ_revisited_verifying_when_changes_induce_fixes
9. Neural SZZ Algorithm - Lingfeng Bao, accessed June 23, 2025, https://baolingfeng.github.io/papers/ASE2023.pdf
10. SZZ revisited: verifying when changes induce fixes - SciSpace, accessed June 23, 2025, https://scispace.com/pdf/szz-revisited-verifying-when-changes-induce-fixes-djobrejyfw.pdf
11. (PDF) Reproducibility and Credibility in Empirical Software ..., accessed June 23, 2025, https://www.researchgate.net/publication/323843822_Reproducibility_and_Credibility_in_Empirical_Software_Engineering_A_Case_Study_based_on_a_Systematic_Literature_Review_of_the_use_of_the_SZZ_algorithm
12. SZZ Unleashed: An Open Implementation of the SZZ Algorithm - arXiv, accessed June 23, 2025, https://arxiv.org/pdf/1903.01742
13. Reproducibility and Credibility in Empirical Software Engineering - It Will Never Work in Theory, accessed June 23, 2025, https://neverworkintheory.org/2021/10/02/reproducibility-and-credibility-in-empirical-software-engineering.html
14. Identifying Defect-Inducing Changes in Visual Code | Request PDF - ResearchGate, accessed June 23, 2025, https://www.researchgate.net/publication/373753750_Identifying_Defect-Inducing_Changes_in_Visual_Code
15. Revisiting and Improving SZZ Implementations - ResearchGate, accessed June 23, 2025, https://www.researchgate.net/profile/Daniel-Costa-62/publication/336637580_Revisiting_and_Improving_SZZ_Implementations/links/5db420d04585155e270176e1/Revisiting-and-Improving-SZZ-Implementations.pdf
16. (PDF) Revisiting and Improving SZZ Implementations - ResearchGate, accessed June 23, 2025, https://www.researchgate.net/publication/336637580_Revisiting_and_Improving_SZZ_Implementations
17. Enhancing Bug-Inducing Commit Identification: A Fine-Grained Semantic Analysis Approach, accessed June 23, 2025, https://www.computer.org/csdl/journal/ts/2024/11/10711218/20UdDwBlp2U
18. The Ghost Commit Problem When Identifying Fix-Inducing Changes - Software REBELs - University of Waterloo, accessed June 23, 2025, https://rebels.cs.uwaterloo.ca/papers/tse2021_rezk.pdf
19. LLM4SZZ: Enhancing SZZ Algorithm with Context-Enhanced Assessment on Large Language Models - arXiv, accessed June 23, 2025, https://arxiv.org/html/2504.01404v1
20. On Refining the SZZ Algorithm with Bug Discussion Data - PMC, accessed June 23, 2025, https://pmc.ncbi.nlm.nih.gov/articles/PMC11269517/

21. Use of Relative Code Churn Measures to Predict System Defect ..., accessed June 23, 2025, https://www.microsoft.com/en-us/research/publication/use-of-relative-code-churn-measures-to-predict-system-defect-density/
22. Use of Relative Code Churn Measures to Predict System Defect Density, accessed June 23, 2025, https://www.st.cs.uni-saarland.de/edu/recommendation-systems/papers/ICSE05Churn.pdf
23. Use of relative code churn measures to predict system defect density - ResearchGate, accessed June 23, 2025, https://www.researchgate.net/publication/4200542_Use_of_relative_code_churn_measures_to_predict_system_defect_density
24. An Enhanced Fault Prediction Model for Embedded Software based on Code Churn, Complexity Metrics, and Static Analysis Results - UPV, accessed June 23, 2025, https://personales.upv.es/thinkmind/dl/conferences/icsea/icsea_2019/icsea_2019_8_10_10121.pdf
25. Software bug prediction using object-oriented metrics - Indian Academy of Sciences, accessed June 23, 2025, https://www.ias.ac.in/article/fulltext/sadh/042/05/0655-0669
26. Source Code Metrics for Software Defects Prediction - arXiv, accessed June 23, 2025, https://arxiv.org/pdf/2301.08022
27. Software Bug Prediction using Machine Learning Approach - The Science and Information (SAI) Organization, accessed June 23, 2025, https://thesai.org/Downloads/Volume9No2/Paper_12-Software_Bug_Prediction_using_Machine_Learning.pdf
28. On the Relationship Between Change Coupling and Software Defects - ResearchGate, accessed June 23, 2025, https://www.researchgate.net/publication/221200492_On_the_Relationship_Between_Change_Coupling_and_Software_Defects
29. On the Relationship Between Change Coupling and Software Defects, accessed June 23, 2025, http://www.inf.unibz.it/~rrobbes/p/WCRE2009-changecoupling.pdf
30. (PDF) An Empirical Study of the Relation Between Strong Change Coupling and Defects Using History and Social Metrics in the Apache Aries Project - ResearchGate, accessed June 23, 2025, https://www.researchgate.net/publication/282692989_An_Empirical_Study_of_the_Relation_Between_Strong_Change_Coupling_and_Defects_Using_History_and_Social_Metrics_in_the_Apache_Aries_Project
31. A Large-Scale Empirical Study of Just-in-Time Quality Assurance, accessed June 23, 2025, https://posl.ait.kyushu-u.ac.jp/~kamei/publications/Kamei_TSE2013.pdf
32. A Developer Centered Bug Prediction Model - FABIO PALOMBA, accessed June 23, 2025, https://fpalomba.github.io/pdf/Journals/J5.pdf

33. Just-In-Time Defect Prediction on JavaScript Projects: A Replication Study - Xin Xia, accessed June 23, 2025, https://xin-xia.github.io/publication/tosem221.pdf
34. Studying Just-In-Time Defect Prediction using Cross-Project Models - POSL, accessed June 23, 2025, https://posl.ait.kyushu-u.ac.jp/~kamei/publications/Kamei_EMSE2015.pdf
35. Re-evaluating Method-Level Bug Prediction - Fabio Palomba, accessed June 23, 2025, https://fpalomba.github.io/pdf/Conferencs/C25.pdf
36. (PDF) A large-scale empirical study of just-in-time quality assurance (2013) | Yasutaka Kamei | 715 Citations - SciSpace, accessed June 23, 2025, https://scispace.com/papers/a-large-scale-empirical-study-of-just-in-time-quality-oqb4b5ys1y
37. An Empirical Study on JIT Defect Prediction Based on BERT-style Model - arXiv, accessed June 23, 2025, https://arxiv.org/html/2403.11158v1
38. On the calibration of Just-in-time Defect Prediction (MSR 2025 ..., accessed June 23, 2025, https://2025.msrconf.org/details/msr-2025-technical-papers/25/On-the-calibration-of-Just-in-time-Defect-Prediction
39. A Systematic Review of Fault Prediction approaches used in Software Engineering - Lero, accessed June 23, 2025, https://lero.ie/sites/default/files/Lero-TR-2010-04.pdf
40. Empirical Study on Software Bug Prediction - ResearchGate, accessed June 23, 2025, https://www.researchgate.net/publication/324046883_Empirical_Study_on_Software_Bug_Prediction
41. A Systematic Literature Review and Meta-analysis on Cross Project Defect Prediction, accessed June 23, 2025, https://bura.brunel.ac.uk/bitstream/2438/15341/1/Fulltext.pdf
42. Early Software Bug Prediction: A Literature Review and Current Trends - GRENZE Scientific Society, accessed June 23, 2025, https://thegrenze.com/pages/servej.php?fn=459.pdf&name=Early%20Software%20Bug%20Prediction:%20A%20Literature%20Reviewand%20Current%20Trends&id=3402&association=GRENZE&journal=GIJET&year=2024&volume=10&issue=2
43. (PDF) A Systematic Review of Fault Prediction Performance in ..., accessed June 23, 2025, https://www.researchgate.net/publication/224260840_A_Systematic_Review_of_Fault_Prediction_Performance_in_Software_Engineering
44. Advancements in Predictive Models for Software Defects: A ..., accessed June 23, 2025, https://ijsret.com/wp-content/uploads/2024/07/IJSRET_V10_issue4_282.pdf
45. Overview of modern software bug prediction approaches - ResearchGate, accessed June 23, 2025, https://www.researchgate.net/publication/371488121_Overview_of_modern_software_bug_prediction_approaches
46. LLMs for Defect Prediction in Evolving Datasets: Emerging Results ..., accessed

June 23, 2025, https://conf.researchr.org/details/fse-2025/fse-2025-ideas-visions-and-reflections/1/LLMs-for-Defect-Prediction-in-Evolving-Datasets-Emerging-Results-and-Future-Directio

47. Evaluating SZZ Implementations: An Empirical Study on the Linux Kernel | OpenReview, accessed June 23, 2025, https://openreview.net/forum?id=LwIJzJkkOF
48. (PDF) Search-Based Fairness Testing for Regression-Based Machine Learning Systems, accessed June 23, 2025, https://www.researchgate.net/publication/359502210_Search-Based_Fairness_Testing_for_Regression-Based_Machine_Learning_Systems
49. Thomas Zimmermann - Research Statement - CiteSeerX, accessed June 23, 2025, https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=799f311c53d90a5992af7777d811d809bca018f4