Detecting and Mitigating Inconsistencies Between Code, Documentation and Tests

Tobias Kiecker
tobias.kiecker@hu-berlin.de
Department of Computer Science, Software Engineering
Humboldt-Universität zu Berlin

***Abstract*—Inconsistencies between different software artifacts, such as source code, documentation, and tests, are a common and long-standing problem in software engineering. These misalignments can degrade software quality, slow down development, and hinder maintenance. Each of these artifacts provides a distinct yet overlapping perspective of the same software behavior, forming a triangular relationship in which any one artifact can, in principle, be used to regenerate the others.**

In this PhD project, we aim to exploit these relationships through regeneration-based techniques to detect and ultimately mitigate inconsistencies across software artifacts. The approach focuses on building an SE-tool that leverages original and regenerated versions of artifacts to triangulate inconsistencies and improve reliability through cross-validation. A particular emphasis is placed on reducing false positives, ensuring that reported issues are trustworthy and actionable, especially when presented to developers in real-world projects.

I. Introduction

In modern software development, inconsistencies between a system's implementation and its supporting artifacts remain a pervasive issue [1, 2]. Developers frequently update source code without making corresponding changes to documentation or tests, which leads to artifact drift as projects evolve. This not only degrades the overall quality of the software but also hinders comprehension, slows down users, and complicates maintenance [3, 4].

Any single software unit such as a method can be represented through three core artifacts: the executable source *code* itself, human-readable *documentation* describing intended behavior, and formal *tests* validating correctness. These three collectively define what a system is and does. Yet, despite this shared purpose, they often fall out of sync over time.

a) Code–Documentation Inconsistencies: This is perhaps the most prevalent form of drift. When functionality changes but developers neglect to update accompanying documentation, the result is often incorrect assumptions, misuse of APIs, and erroneous design decisions. Outdated documentation thus becomes a liability [3].

b) Code–Test Inconsistencies: These are typically the most immediately visible inconsistencies: they usually manifest as failing test cases. Such failures usually indicate that either the test is incorrect or, more likely, that the code no longer behaves as expected. However, not all mismatches are this explicit. Tests may continue to pass even when outdated or incomplete. For example, if new behaviors are added to the code but not covered by tests, or if assertions are too weak, subtle bugs can go undetected despite a passing test suite.

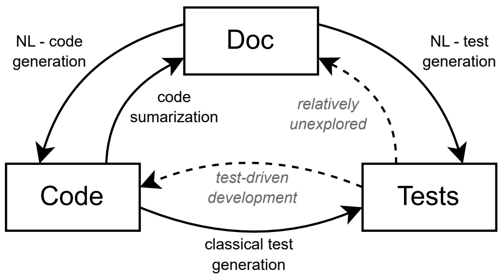

Fig. 1. The Code-Documentation-Tests triangle and its connections. Dotted arrows show relatively unexplored tasks regarding automation.

c) Test–Documentation Inconsistencies: This category is less studied than the other two, but such inconsistencies can still signal deeper problems. They occur when tests and documentation convey conflicting expectations. For instance, documentation may describe behavior that tests do not cover, or tests may validate scenarios not mentioned in the documentation, e.g., in error handling or edge cases. When tests and documentation are misaligned, it often hints at conceptual drift: either the intended behavior is unclear, or the implementation has evolved without proper reflection in auxiliary artifacts.

It is clear, that a misalignment of two of those artifacts can hint to a general problem with the software. This PhD Project will investigate how the interdependencies among code, documentation, and tests can be harnessed for automated inconsistency detection and resolution. These artifacts do not exist in isolation. They are inherently linked, often describing the same functionality from different perspectives.

In an idealized development workflow, any one of the three artifacts could be used to (re-)generate the other two, as illustrated in Figure 1. For instance, high-quality documentation enables automatic code generation [5] and conversely well-written code can be summarized by similar tools [6]; tests can be generated from either code [7] or NL specifications [5, 8], while in some cases, they can also serve as a basis for inferring intended code behavior, like it is done in test driven development. Although these techniques vary in maturity and feasibility, the bidirectional nature of these relationships is clear.

This provides a unique opportunity: by exploiting expected consistency, one can detect when an artifact becomes misaligned with the others. From this starting point, this PhD project will further explore mechanisms for detecting misalignments and proposing or generating repairs.

A major concern in developing such mechanisms is minimizing false positives. Since the goal is to assist developers by reporting inconsistencies directly to them, it is essential that reported issues are highly reliable. Studies have shown that developers quickly lose trust in tools that frequently raise incorrect alerts [9, 10]. Therefore, special emphasis will be placed on ensuring that any detected inconsistency is validated or cross-checked through multiple signals.

II. Background and Related Work

Most existing work on inconsistency detection focuses on the relationship between code and documentation [2, 11, 12]. This body of research serves as the conceptual foundation for the PhD project, which generalizes the detection problem to include tests as a third interdependent artifact.

Two principal strategies for detecting inconsistencies have emerged in the literature: *just-in-time* detection and *post-hoc* analysis [12]. Just-in-time methods aim to catch inconsistencies before they are introduced into the code base—typically at commit time. These approaches include machine learning models trained on historical code-comment changes to suggest documentation edits in response to code modifications [12, 13]. Other work enforces consistency using constraint systems or through mapping and monitoring links between documentation and code diffs [11].

Post-hoc techniques, in contrast, aim to identify inconsistencies in existing codebases. These range from language-specific static analysis tools [14] to general-purpose NLP-driven models trained to classify documentation-code pairs as consistent or inconsistent [2]. Some approaches infer contracts from comments and then validate them using generated tests [1].

While this prior work has largely been confined to code–comment inconsistencies, the goal of the PhD project is to extend those ideas to also cover the relationships between code, tests, and documentation. Thus, treating all three as equally authoritative views of software behavior.

Existing methods offer promising detection capabilities, however, they often lack robust mechanisms for filtering out false positives. This issue becomes even more pronounced when inconsistencies are automatically inferred. The proposed approach aims to build on these foundations by introducing cross-validation mechanisms that combine multiple signals across artifacts, with the specific goal of improving precision and in turn developer trust.

III. Proposed Solution

The core concept of the PhD project is based on the idea that inconsistencies between software artifacts can be detected by systematically leveraging the inherent relationships between them. The proposed methodology relies on the idea of *regeneration*: creating one artifact from another, and comparing the generated version with its existing counterpart. The intuition is: if we generate for example, documentation from code, or tests from documentation, and the regenerated artifact diverges significantly from the existing one, we obtain evidence of a possible inconsistency. Since each of the three core artifacts—code, documentation, and tests—can be used to describe the same software unit, this triangle of relationships becomes a powerful structure for both detecting and cross-validating inconsistencies.

In practice, we envision a system that, for any given software component, can build a space of mutually generated artifacts: original versions, regenerated versions, and hybrid comparisons. Through this redundancy, inconsistencies can be triangulated more reliably.

One of the guiding design principles of the system will be to minimize false positives. This can be achieved by leveraging the redundancy in the artifact space: for example, a documentation-test mismatch may be confirmed by the corresponding code-documentation inconsistency, increasing the confidence in both findings. Similarly, external artifacts, such as issue trackers or formal specifications, can be selectively incorporated to support or disregard a given inconsistency signal. This multi-perspective validation serves to ensure that developers are only notified of issues with a high likelihood of being real and impactful.

Although this work will primarily focus on unit-level inconsistencies, the methods could be extended to cover higher-level structures. While the triangle remains the central focus, we also acknowledge the existence of additional software artifacts such as issue trackers, architectural diagrams, or formal specifications. These may serve as useful future extensions to enrich detection strategies.

To guide the exploration and structuring of the project, we define the following research questions:

RQ1: What types of inconsistencies occur in practice, and how severe or prevalent are they? This question aims to establish a foundational understanding of real-world inconsistency patterns. It includes identifying a taxonomy of inconsistency types (e.g., omissions, contradictions, outdated descriptions) and analyzing their frequency and potential impact across large, open-source codebases. Further, it will help to solidify the terms code-test inconsistency and documentation-test inconsistency. We plan to conduct empirical studies and corpus analysis to inform both the modeling and evaluation of detection techniques.

RQ2: How can inconsistencies between code and documentation be detected reliably? While this problem has received prior attention, we seek to explore novel approaches based on regeneration. For instance, we aim to investigate how documentation generated from code using LLMs aligns with existing documentation, and how divergences can serve as inconsistency signals. We will also evaluate the precision and usefulness of such techniques in real-world contexts.

RQ3: How can inconsistencies between documentation and tests be uncovered? This question explores the lesser-studied relationship between natural-language descriptions and formal validation. We aim to generate tests from documentation and analyze whether their behavior aligns with existing test suites. Discrepancies—such as documentation describing cases that are not tested, or vice versa—can be framed and flagged as candidate inconsistencies.

RQ4: How can inconsistencies between code and tests be identified beyond simple test failures? While failing test cases are a clear signal of inconsistencies, more subtle forms of drift often go unnoticed. We aim to compare regenerated tests (e.g., derived from code behavior or specifications) with the actual test suite to detect gaps, redundancy, or behavioral mismatches. This includes regression test synthesis and differential test comparison.

RQ5: Can the methods designed for post-hoc detection also support just-in-time prevention? Most detection approaches assume an existing codebase, but we ask whether these methods can be adapted to pre-commit scenarios. For example, can regenerated artifacts or confidence metrics serve as pre-merge warnings for potential inconsistencies? This would bring our system closer to practical developer tooling that supports consistency in real time.

RQ6: How can false positives be reduced through cross-artifact validation? In order to be adopted in practical workflows, inconsistency detection tools must be highly precise. This question investigates how multiple perspectives, e.g., original and regenerated artifacts, external references, and different combinations of those, can be used to validate candidate inconsistencies before reporting them to developers.

RQ7: How can candidate repair suggestions be derived from detected inconsistencies? Rather than automatically modifying artifacts, we aim to develop mechanisms that propose actionable fixes to developers. These may include suggested documentation updates, missing test stubs, or highlighted code areas needing review. We will prototype such systems and, where possible, test them in real open-source repositories to gather feedback on usefulness and acceptance.

IV. Expected Contributions

The proposed research aims to advance the understanding and automated tooling around inconsistencies in software artifacts through the following contributions:

- **Novel algorithms for inconsistency detection and mitigation.** The core contribution will be the development of methods that leverage regenerated artifacts, using techniques such as: test generation, code summarization, or documentation synthesis, to detect, classify, and propose fixes for inconsistencies across code, documentation, and tests.
- **Open-source tools based on these algorithms.** Practical implementations of the developed techniques will be released as open-source tools, enabling automated or developer-assisted detection of inconsistencies in real-world codebases.
- **Curated datasets of real-world inconsistencies.** A dataset of annotated inconsistencies and corresponding human or tool-generated fixes will be constructed from open-source repositories. This resource will support benchmarking and future research on artifact misalignment.

Other contributions will include methodological advances in test, code, or documentation generation, especially when one is synthesized from another. These may lead to improvements in regression test synthesis, prompt engineering for LLM-based generation, or consistency-aware code summarization. In addition, the research may yield taxonomies and empirical insights into the prevalence and nature of inconsistencies across diverse codebases.

V. Results achieved so far

The project is still in an early phase, however, we already mined an initial dataset of code-documentation inconsistencies. Furthermore, we developed an LLM-based approach that generates unit tests based on the documentation of the method.

VI. Planned Evaluation

Evaluation will be conducted at the level of individual detection and generation tasks, with the overarching goal of demonstrating the effectiveness of regeneration-based methods for identifying and mitigating inconsistencies between software artifacts.

A central element of the evaluation strategy is the use of real-world datasets. Existing datasets for code–documentation inconsistencies often suffer from limitations such as reliance on heuristics [12] or synthetic modifications [15]. As part of the PhD project, we plan to curate a new dataset derived from real-world open-source projects. Candidate inconsistencies will be manually reviewed to ensure correctness, thereby enabling more reliable and representative benchmarking.

Where applicable, evaluations will be conducted against existing tools. For code–documentation inconsistency detection, several machine learning or static analysis-based tools are available and will be used as baselines [2, 11, 15, 16]. For code–test inconsistencies, we will draw from the broader domain of bug localization or explanation tools, comparing how effectively our methods identify or explain failed test cases in comparison to established techniques.

Each detection or generation subtask will be evaluated independently using appropriate metrics e.g., precision, recall, or top-k accuracy depending on the task. Since the larger system is composed of smaller regenerative pipelines, it is important to evaluate each link of the triangle (code, tests, documentation) in isolation before validating the whole. Precision is the most important metric here, as it relates to the amount of false positives. Where possible, feedback from developers (e.g., in GitHub issue threads) will be used to validate that reported inconsistencies are genuinely useful.

A strong and generic baseline for all subtasks will be direct prompting of a large language model. Since LLMs will likely be involved in our generation processes, it is essential to ask:

to what extent could consistency checking or regeneration be done directly by an LLM? This will serve as a baseline we aim to improve upon through structural validation and more targeted workflows. In addition, fine-tuning code-oriented language models on the specific inconsistency detection tasks may serve as an alternate baseline in settings where sufficient training data is available.

Overall, the evaluation will follow a modular approach, aligned with the architecture of the system itself. The final goal is not just to show that individual detection techniques work, but also to demonstrate how these components can be integrated into a broader tool chain that supports practical, developer-facing use cases.

VII. Plan for Demonstrating Research Contribution

This research aims to produce scientifically rigorous and practically valuable contributions to the software engineering community. To demonstrate its impact, the following strategies will be employed:

All developed tools, models, and datasets will be made publicly available under open-source licenses to ensure transparency, reproducibility, and future reuse. The dataset of real-world inconsistencies and fixes will serve as a benchmark resource for the broader research community.

Evaluation will be conducted using real-world data and standard metrics, and each component of the system will be compared against existing state-of-the-art approaches. These comparisons will highlight the novelty and practical advantages of regeneration-based detection techniques.

While the problem of inconsistencies in software artifacts has received prior attention—particularly in the context of code and documentation—the proposed approach introduces new techniques that systematically leverage regenerated artifacts. Additionally, one of the core problem areas, inconsistencies between documentation and tests, remains largely underexplored, further supporting the originality of the research direction.

Real-world relevance will be assessed by applying the tools to open-source projects and reporting detected inconsistencies to developers. Their feedback will serve as informal validation of practical utility. In this setting, minimizing false positives is critical: every reported issue must be highly reliable to avoid wasting developer time. Therefore, precision and cross-validation will be central to the design, ensuring that the system delivers not only novel insights but also trustworthy ones.

References

[1] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tcomment: Testing javadoc comments to detect comment-code inconsistencies," in *ICST*. IEEE Computer Society, 2012, pp. 260–269.

[2] A. T. V. Dau, N. D. Q. Bui, and J. L. C. Guo, "Bootstrapping code-text pretrained language model to detect inconsistency between code and comment," *CoRR*, vol. abs/2306.06347, 2023.

[3] G. Uddin and M. P. Robillard, "How API documentation fails," *IEEE Softw.*, vol. 32, no. 4, pp. 68–75, 2015.

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

[5] A. Ni, S. Iyer, D. Radev, V. Stoyanov, W. Yih, S. I. Wang, and X. V. Lin, "LEVER: learning to verify language-to-code generation with execution," in *ICML*, ser. Proceedings of Machine Learning Research, vol. 202. PMLR, 2023, pp. 26 106–26 128.

[6] Y. Zhu and M. Pan, "Automatic code summarization: A systematic literature review," *CoRR*, vol. abs/1909.04352, 2019.

[7] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *SIGSOFT FSE*. ACM, 2011, pp. 416–419.

[8] S. Kang, L. Milliken, and S. Yoo, "Identifying inaccurate descriptions in llm-generated code comments via test execution," *CoRR*, vol. abs/2406.14836, 2024.

[9] R. Croft, D. Newlands, Z. Chen, and M. A. Babar, "An empirical study of rule-based and learning-based approaches for static application security testing," in *ESEM*. ACM, 2021, pp. 8:1–8:12.

[10] R. Krishnamurthy, T. S. Heinze, C. Haupt, A. Schreiber, and M. Meinel, "Scientific developers v/s static analysis tools: vision and position paper," in *CHASE@ICSE*. IEEE / ACM, 2019, pp. 89–90.

[11] N. Stulova, A. Blasi, A. Gorla, and O. Nierstrasz, "Towards detecting inconsistent comments in java source code automatically," in *SCAM*. IEEE, 2020, pp. 65–69.

[12] S. Panthaplackel, P. Nie, M. Gligoric, J. J. Li, and R. Mooney, "Learning to update natural language comments based on code changes," in *Proceedings of the 58th ACL*, 2020, pp. 1853–1868.

[13] K. Liu, D. Kim, T. F. Bissyandé, T. Kim, K. Kim, A. Koyuncu, S. Kim, and Y. L. Traon, "Learning to spot and refactor inconsistent method names," in *ICSE*. IEEE / ACM, 2019, pp. 1–12.

[14] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "An empirical study of the non-determinism of chatgpt in code generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 2, pp. 42:1–42:28, 2025.

[15] H. Lee, G. An, and S. Yoo, "Metamon: Finding inconsistencies between program documentation and behavior using metamorphic LLM queries," pp. 120–127, 2025.

[16] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. D. Castellanos, "Translating code comments to procedure specifications," in *ISSTA*. ACM, 2018, pp. 242–253.