

Towards Automated Test Adaptation in Fork Ecosystems via Large Language Models

Downloaded from: https://research.chalmers.se, 2026-11-08 14:10 UTC

Citation for the original published paper (version of record):
Mukelabai, M., Schurkus, K., Noller, Y. et al (2026). Towards Automated Test Adaptation in Fork Ecosystems via Large Language Models. Fse Companion 2026 Proceedings of the 34th ACM International Conference on the Foundations of Software Engineering: 1391-1395. http://dx.doi.org/10.1145/3803437.3805594

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology. It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004. research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)

Towards Automated Test Adaptation in Fork Ecosystems via Large Language Models

Mukelabai Mukelabai
Ruhr University Bochum
The University of Zambia
Germany
mukelabai.mukelabai@rub.de

Keanu-Wesley Schurkus
Ruhr University Bochum
Germany
keanu-wesley.schurkus@rub.de

Yannic Noller
Ruhr University Bochum
Germany
yannic.noller@acm.org

Thorsten Berger
Ruhr University Bochum
Chalmers | University of Gothenburg
Germany
thorsten.berger@rub.de

Abstract

Fork-based software ecosystems enable independent evolution, but fragment the quality assurance efforts. Although tests developed in one fork are often relevant to others, direct reuse is rare and can be difficult, due to API drift, refactorings, dependency evolution, or other changes. Prior work has shown that many tests are *potentially reusable* across forks, yet adapting them remains a largely manual task. This paper explores *LLM-assisted test case adaptation* as a new direction for automating test propagation in fork ecosystems. We propose a compiler-aware, iterative adaptation technique that integrates tests from one fork into another and leverages LLMs to repair incompatibilities revealed by build failures. We report empirical results on Java fork pairs, showing that LLM guidance can successfully adapt tests in 80% of cases where direct reuse fails due to modifications required to the test code itself. We outline research directions toward effective and scalable test propagation.

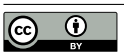
CCS Concepts

• **Software and its engineering** → **Reusability**; **Software testing and debugging**.

Keywords

forking, software ecosystems, test reuse, test adaptation, LLMs

ACM Reference Format:
Mukelabai Mukelabai, Keanu-Wesley Schurkus, Yannic Noller, and Thorsten Berger. 2026. Towards Automated Test Adaptation in Fork Ecosystems via Large Language Models. In *34th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE Companion '26), July 05–09, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3803437.3805594



This work is licensed under a Creative Commons Attribution 4.0 International License.
FSE Companion '26, Montreal, QC, Canada
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2636-1/2026/07
https://doi.org/10.1145/3803437.3805594

1 Introduction

Forking is a commonly used in modern software development to experiment with new ideas or customize software to meet varying requirements for specific market segments or environments [10, 32]. Large platforms such as GitHub host extensive ecosystems of forks that share a common ancestry, but diverge independently over time [8, 43]. While code reuse across forks is common, test cases are rarely propagated, even when they exercise relevant and overlapping functionality. Empirical studies on fork ecosystems show that reuse across divergent forks is infrequent and often handled in ad hoc ways, increasing maintenance costs and quality risks [7, 8, 35]. Moreover, integration conflicts during merging [15, 17, 19, 33] and reuse are further correlated with post-merge defects, underscoring the importance of robust validation assets, such as tests [6].

Bug-fix propagation between forks [2, 12, 13, 20, 24, 27–31, 37] or their integration into configurable platforms (a.k.a. software product lines [1, 4]) are active research areas. While these could address the issues in the first place, the former needs awareness of bugs, and the latter aim at a different goal [3] and would require a central organization to integrate changes. Instead, we advocate for better testing to improve the software quality of fork ecosystems and detect bugs early. In fact, recent work [22, 23] demonstrated that a substantial fraction of tests in fork ecosystems are *reusable in principle*, but require manual adaptation due to refactorings, API drift, and structural divergence, among other changes. Automating this adaptation step remains an open challenge.

Meanwhile, large language models (LLMs) have demonstrated promising capabilities in code transformation, refactoring, and automated repair, even for unit tests within single repositories [18, 25, 26, 36]. This raises the question: *To what extent can large language models assist in automatically adapting test cases across forked projects?* In fact, automated adaptation remains largely unexplored. We argue that LLMs enable a new, compiler-grounded design space for this problem, and propose an iterative, diagnostic-driven adaptation workflow, supported by early empirical evidence on 12 Java test cases from 9 fork ecosystems, along with an explicit discussion of limitations in soundness, cost, and semantic drift.

Our results show that test-case adaptation can be treated as an iterative process, where compiler errors guide large language models to make small, targeted fixes until the test compiles.

2 Background and Motivation

Test-Case Reuse in Fork Ecosystems. Fork ecosystems consist of projects that originate from a common codebase, but evolve independently over time. While forking facilitates experimentation and customization, it also leads to divergence in both functionality and quality assurance artifacts. Empirical studies show that changes—such as bug fixes or new test cases—are only sporadically propagated across forks [35], resulting in redundant effort and inconsistent quality [6–8]. Potentially, integrating forks into one codebase could address this issue. The field of software variability management (a.k.a. software product line engineering [1, 4]) has contributed methods to create and manage such codebases. However, fork ecosystems have different goals—such as isolated development, quick code reuse, or encouraging new contributions [3], with only sporadic integrations of changes across forks [35].

We conducted the first large-scale empirical study explicitly targeting *test-case reuse in fork ecosystems* [23]. Analyzing 305 GitHub fork ecosystems comprising 1,089 Java projects, they found that nearly half (48 %) of all test cases added during fork evolution were missing in at least one sibling project. For a substantial subset of these missing test cases, the corresponding units under test (UUTs) still existed in other forks—74% of them were type 1 or 2 clones—indicating significant unrealized potential for test-case propagation. Specifically, by manual inspection and transplantation of test cases, they identified three categories of applicability: (i) test cases that are *applicable as-is*, (ii) test cases that require *modifications to the test code* (WMTC), and (iii) test cases that require *modifications to the unit under test* (WMUUT). While the first category enables straightforward reuse, the latter two account for the majority of unrealized reuse opportunities.

Limitations of Manual and Static Adaptation. We further showed [23] that deciding test-case applicability and performing the necessary adaptations is cognitively demanding and difficult to automate. Even when UUTs exhibit high syntactic similarity, subtle differences in dataflow, context, or behavior often require non-trivial reasoning about intent and semantics. Their manual transplantation study revealed that static similarity metrics and conservative dataflow analyses are insufficient to reliably determine whether and how a test can be adapted.

Existing automated techniques [11, 16, 21, 42] for test reuse and adaptation typically rely on static analysis, clone detection, rule-based refactoring, or traditional machine learning. While such techniques work well for simple cases—such as type 1 or type 2 clones—they break down when confronted with co-evolution, API drift, dependency changes, or contextual shifts that are common in long-lived forks. As a result, test-case adaptation across forks remains largely a manual activity, limiting scalability and adoption.

Opportunities Enabled by LLMs. Prior work demonstrates that LLMs can support complex software maintenance tasks, including refactoring and structural code transformation [18, 34]. In adjacent areas of integration, learning-based techniques, such as MergeBERT and MergeGen, have shown that pretrained models can reconcile code differences and assist in resolving semantic merge conflicts beyond purely syntactic techniques [9, 36, 40]. Structured merge tools, such as Spork, further highlight the difficulty of handling moves and renames with static techniques alone [14].

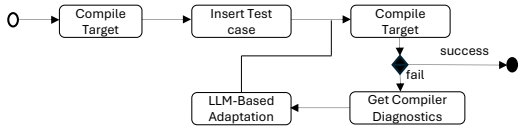


Figure 1: Compiler-aware LLM-assisted test adaptation

Recent work on LLM-based test repair provides strong evidence that LLMs can adapt tests in response to code evolution. UTFix [26], for example, demonstrates that LLMs can repair broken unit tests caused by API changes or behavioral drift, outperforming traditional rule-based approaches in complex scenarios. Complementary agentic approaches show that LLMs can autonomously prepare build environments and execute tests across previously unseen projects, bridging the gap between adaptation and execution [5].

While recent work [5, 26, 38, 39, 41] applies LLMs to generate or repair tests under code evolution and to automate project-level test execution, it does not address the distinct problem of *adapting and reusing existing tests across divergent forks*—where portability is constrained by cross-project API drift, refactorings, and ecosystem-specific context [23].

The Need for LLM-Assisted Test Adaptation. These insights motivate the use of LLMs for test-case adaptation in fork ecosystems. Unlike static techniques, LLMs can reason over compiler diagnostics, test code, and evolving implementation context, which potentially helps address the two main problematic scenarios of test-case applicability identified above: those applicable with modification to the test case (WMTC) and with modification to units under test (WMUUT). Our idea is to use LLMs as a mechanism to *operationalize test adaptation as an interactive, diagnostic-driven process*. In this paper, we explore this idea by investigating the extent to which LLMs can effectively assist developers in adapting test cases across forks when guided by concrete build feedback.

3 LLM-Assisted Adaptation

Our technique performs *test-case adaptation* as a compiler-guided, iterative repair process. Given a source test and a target fork that contains a (possibly diverged) UUT, we (i) validate that the target project builds in isolation, (ii) transplant the test into the target while preserving its package structure, and (iii) repeatedly compile the project and use build diagnostics to drive LLM-based edits until the test becomes compilable, or an attempt budget is exhausted.

We realized this technique in a Python-based prototype targeting Java projects. The prototype performs the pre-build validation, test insertion, iterative build-driven adaptation, and metrics collection, while interfacing with Maven-based builds and an LLM (Gemini Flash 2.5 in our evaluation). Figure 1 illustrates our workflow.

Design Goals. Our technique is guided by three goals: (i) grounding each adaptation step in concrete compiler feedback to drive minimal, diagnostic-informed changes, (ii) enforcing a conservative edit scope that restricts modifications to test code (and tightly guarded build-configuration fixes) to preserve production safety, and (iii) ensuring reproducible, cost-aware behavior through a bounded number of adaptation attempts with attempt-level diagnostics.

Pre-build Validation. Algorithm 1 describes the pre-build and test insertion stage. This takes as input the *source test code T* (obtained

Algorithm 1 Pre-build Validation and Test Insertion Workflow

Require: source test code T, source test path p_s, target root R, UUT relpath u, max attempts N

1: $bs \leftarrow$ DetectBuildSystem(R) ▷ expects Maven
2: $(c, out, err) \leftarrow$ Build(R, bs)
3: **if** $c \neq 0$ **then**
4: **if** ConfigFixAllowed() **then**
5: $P \leftarrow$ ConstructPomFixPrompt(err, `pom.xml`)
6: $\hat{X} \leftarrow$ QueryLLM(P)
7: **if** ExtractXML($\hat{X}$) **then**
8: ApplyPatch(`pom.xml`, $\hat{X}$)
9: $(c, out, err) \leftarrow$ Build(R, bs)
10: **end if**
11: **end if**
12: **if** $c \neq 0$ **then**
13: **return** RecordPreBuildFail()
14: **end if**
15: **end if**
16: $t \leftarrow$ InsertTest(T, p_s, R)
17: $success \leftarrow$ AdaptationLoop(t, R, u, bs, N)
18: **if** $\neg success$ **and** CleanupOnFailure() **then**
19: RestoreOrRemoveInsertedTest(t)
20: **end if**
21: **return** FinishTracking()

Algorithm 2 Iterative Build–Adaptation Loop

Require: test file t, target root R, UUT relpath u, build system bs, max attempts N

1: $C \leftarrow$ ReadFile(t)
2: **for** $i = 1$ **to** N **do**
3: $(c, out, err) \leftarrow$ Build(R, bs)
4: **if** $c = 0$ **then**
5: RecordAttempt(i, $success = true$); **return** true
6: **end if**
7: $E \leftarrow$ ParseErrors(out **or** err, bs)
8: $U \leftarrow$ ReadFile($R \| u$)
9: $P \leftarrow$ ConstructAdaptPrompt(C, E, U)
10: $\hat{C} \leftarrow$ QueryLLM(P); $C' \leftarrow$ ExtractJava($\hat{C}$)
11: **if** $C' =$ None **then**
12: RecordAttempt(i, $success = false$); **break**
13: **end if**
14: WriteFile(t, C'); $C \leftarrow C'$
15: RecordAttempt(i, $success = false$)
16: **end for**
17: **return** false

from the source fork), the source test path p_s (used to derive the package insertion location), the target project root R, the relative path to the target UUT u, and the attempt budget N.

Pre-build validation. Lines 1–14 in Algorithm 1 ensure that the target repository is in a buildable state *before* introducing the source test.

We first detect the build system. Note that our current implementation only supports Maven, while our general technique can be applied to other systems, such as Gradle. Thereafter, we invoke a clean compilation (e.g., `mvn clean test-compile -DskipTests`).

If the build fails for configuration reasons, the technique can optionally attempt a *minimal* build-file fix via the LLM (Lines 4–10). In Maven, build configuration files are represented in XML (`pom.xml`). Importantly, such fixes are guarded: a patch is applied only if it parses as XML, contains a single plugin/property change, and round-trips through an XML validator. If the target still fails after this phase, we record a pre-build failure and stop, avoiding confounding later errors with an already broken target project.

Test insertion. If the target passes the pre-build gate, we insert the source test (Line 16). The insertion is package-preserving: the declared `package` statement determines the destination under `src/test/java/...`1 we do not overwrite existing files unless explicitly necessary, and ensure that the resulting path is within the repository root to avoid path traversal from malformed packages.

Delegation to the adaptation loop. Finally, we invoke the iterative build-adaptation loop (Line 17). On persistent failure, the inserted test can be removed or restored depending on cleanup policy (Lines 18–20), and the run is finalized by persisting metrics (Line 21).

Iterative Build–Adaptation Loop. Algorithm 2 is the core of our technique. It repeatedly compiles the modified target project and uses compiler diagnostics to guide LLM-based edits to the test case.

Step 1: Compile and check outcome. At each attempt i, the technique compiles the target project (Line 3). If compilation succeeds, we record success and terminate (Line 4–5). We intentionally use compilation as the primary gate to (i) reduce flakiness from executing tests across heterogeneous environments and (ii) focus the technique on structural adaptation (imports, signatures, framework/API drift) that is a prerequisite for any further validation.

Step 2: Parse and structure diagnostics. If compilation fails, we parse diagnostics (Line 7). The parser extracts relevant build output that directly explains why compilation failed and where, focusing on errors that are likely in scenarios where the forked source code diverged: unresolved symbols, missing imports, incompatible types, changed method signatures, and package/class moves. This compiler output serves as input for the LLM.

Step 3: Assemble prompt context. The technique reads (i) the current test code C (Line 1; updated after each successful extraction) and (ii) the target UUT file U (Line 8). Because our focus is test case adaptation, we assume the target UUT has already been identified (e.g., via clone detection and AST differencing as in Mukelabai et al.,[23]). This keeps the prompt context small and relevant to the observed failures.

Step 4: Constrained LLM query. We construct a prompt P that includes four blocks: (a) a concise instruction framing the task as *adapting* the test rather than re-implementing it, (b) the failing test code C, (c) the UUT excerpt U, and (d) a truncated build log containing the most relevant error lines. The model is required to return exactly one complete Java class in a machine-parsable format, which enables robust extraction and prevents multi-file patches or interleaved narrative text.

Step 5: Validate and apply patch. After querying the LLM (Line 10), we attempt to extract a single class (Line 10). If extraction fails (e.g., due to multiple classes returned, or otherwise unparsable output), we record the attempt and break (Lines 11–12), preserving the current test for analysis. Otherwise, we overwrite the test file with the extracted revision C' (Line 14), update the in-memory code C, and proceed to the next compilation attempt.

Table 1: Adaptation outcomes across 120 experimental runs (10 repetitions per task)

Category	Tasks	Runs	Successes	Rate	Attempts	Time
WMTC	8	80	63	78.8%	2.62	85s
WMUUT	4	40	33	82.5%	2.15	618s
Total	**12**	**120**	**96**	**80.0%**	**2.46**	**262s**

Stopping conditions and bounded cost. The loop stops on first successful compilation or after N attempts. This bounded budget is intentional: in pilot runs we observed diminishing returns after a small number of iterations, while additional attempts increase time and cost and may encourage drifting away from the original intent.

4 Evaluation

Dataset and Setup. We evaluated our technique on the dataset of fork pairs manually transplanted by Mukelabai et al. [23]. From the original set, we excluded three test cases that were applicable as-is, leaving 12 adaptation tasks (from 9 unique ecosystems) that required manual intervention in the baseline study. Of these, eight required modifications to the test case only (WMTC), while four required modifications to the unit under test (WMUUT).

Recall that the prototype targets Java/Maven projects and uses Google Gemini 2.5 flash as the LLM with fixed parameters across all runs, chosen to balance latency, cost, and stability. Each task was allowed a maximum of three adaptation attempts per run. To account for LLM non-determinism, we executed the adaptation workflow ten times per task, resulting in a total of 120 experimental runs.

Design Constraints. To ensure reliable and analyzable behavior across diverse fork pairs, we constrained the design of our prototype as follows: (i) we restricted execution to Maven projects and fixed the Java toolchain to a stable JDK version (JDK 8) to reduce environmental confounds; (ii) we used a conservative edit policy, which limits its modifications to `src/test` and, optionally, a minimal pre-build `pom.xml` patch, preventing silent changes to production code; (iii) we only accepted responses with a single "'`java` fenced class with package consistency and ASCII-only output to avoid malformed or ambiguous patches; and (iv) generated compiler diagnostics for each attempt, recording error categories, response length, and timing to support post-hoc analysis and future cost-aware optimization.

Results. Table 1 summarizes the aggregated outcomes across all runs. A run is considered successful if the adapted test compiles in the target project within the attempt budget. Overall, we achieved successful compilation in 96 out of 120 runs (80.0%). Since the approach is restricted to modifying test code only, we are primarily interested in the WMTC category, while WMUUT cases provide insight into the behavior of the technique under strict constraints.

WMTC: Test-Code Adaptation. The WMTC category represents scenarios where manual reuse was achieved by refactoring the test code without modifying production code. Here, our technique succeeded in 7 of 8 tasks. Four tasks achieved 100% success across all ten runs, one achieved 90%, two achieved 70%, and one failed in all runs.

Successful cases typically involved API renames, package relocations, assertion refactoring, or test framework migration (e.g., JUnit 4 to JUnit 5). These patterns align closely with refactoring tasks previously shown to be within the capabilities of LLMs. The single complete failure involved deep structural divergence in third-party dependencies; the LLM repeatedly attempted to import non-existent classes, exhausting the attempt budget without converging.

WMUUT: Adaptation under UUT-Code Immutability. In the WMUUT category, the manual baseline achieved reuse by modifying the unit under test. Our technique, however, is restricted to test-code modifications. Despite this constraint, it achieved a success rate of 82.5%. Inspection of successful cases reveals a consistent strategy shift: instead of adapting the UUT to satisfy the test, the LLM adapted the test to tolerate the UUT by mocking missing dependencies, removing or weakening assertions, or reducing test scope. While these adaptations achieved compilability, they do not necessarily preserve the semantic intent or coverage of the original test.

These tasks were also substantially more expensive. For instance, one pair from the ecosystem called the *shiro-redis* pair averaged over 2,000 seconds per run, reflecting repeated compilation cycles over complex dependency conflicts that a human developer resolved directly by modifying the UUT or build configuration.

Comparison to the Manual Baseline. Our comparison highlights the following strengths and limitations.

Scope of Repair. The manual baseline operates at whole-repository scope, allowing changes to both `src/main` and `src/test`. Our approach intentionally restricts modifications to `src/test`, prioritizing production safety. This asymmetry enables automation but forces alternative solutions in WMUUT cases.

Consistency vs. Stochasticity. Manual adaptations are deterministic once identified. In contrast, the LLM-based approach exhibits stochastic outcomes: for instance, in one project, occasional hallucinated imports led to unrecoverable states in a subset of runs.

Efficiency Profile. For bounded divergences (WMTC), the technique is efficient, often converging in under one minute. For deep divergences, automation is substantially less efficient, consuming significant time and computation to reach outcomes that a human developer might quickly classify as requiring production-code changes.

Our results demonstrate that LLM-assisted adaptation can replicate a large fraction of human test refactoring behavior in fork ecosystems, while also exposing clear boundaries where semantic divergence or architectural constraints limit automation.

5 Conclusion

We presented an exploration of LLM-assisted test case adaptation in fork ecosystems. Our compiler-aware adaptation loop demonstrates that LLM guidance can automate substantial portions of test adaptation work previously requiring manual effort.

Our results open several research directions. While we use compilability as an initial criteria to study feasibility, future work can integrate execution-based validation (e.g., differential testing, metamorphic relations, or lightweight oracles) to assess whether adapted tests preserve their semantics, potentially enabling verifier-in-the-loop adaptation. The current adaptation loop relies on compiler diagnostics, test code, and localized UUT context. Augmentation with structured project knowledge (e.g., symbol indexes, call graphs, dependency manifests, or refactoring histories) may improve adaptation. Future test-case propagation and adaptation techniques could move beyond a single-model design to orchestrated or agentic workflows that separate build repair, test refactoring, and semantic validation, and evaluate scalability across languages and build systems.

References

[1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.

[2] Earl T Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 306–317.

[3] Thorsten Berger, Rolf-Helge Pfeiffer, Reinhard Tartler, Steffen Dienst, Krzysztof Czarnecki, Andrzej Wąsowski, and Steven She. 2014. Variability mechanisms in software ecosystems. *Information and Software Technology* 56, 11 (2014), 1520–1535.

[4] Thorsten Berger, Jan-Philipp Steghöfer, Tewfik Ziadi, Jacques Robin, and Jabier Martinez. 2020. The State of Adoption and the Challenges of Systematic Variability Management in Industry. *Empirical Software Engineering* 25 (2020), 1755–1797.

[5] Islem Bouzenia and Michael Pradel. 2025. You name it, I run it: An LLM agent to execute tests of arbitrary projects. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 1054–1076.

[6] Caius Brindescu, Iftekhar Ahmed, Carlos Jensen, and Anita Sarma. 2020. An empirical investigation into merge conflicts and their effect on software quality. *Empirical Software Engineering* 25, 1 (2020), 562–590.

[7] John Businge, Openja Moses, Sarah Nadi, Engineer Bainomugisha, and Thorsten Berger. 2018. Clone-Based Variability Management in the Android Ecosystem. In *34th IEEE International Conference on Software Maintenance and Evolution (ICSME), Industry Track*.

[8] John Businge, Openja Moses, Sarah Nadi, and Thorsten Berger. 2022. Reuse and Maintenance Practices among Divergent Forks in Three Software Ecosystems. *Empirical Software Engineering (EMSE)* 27, 2 (2022), 54.

[9] Jinhao Dong, Qihao Zhu, Zeyu Sun, Yiling Lou, and Dan Hao. 2023. Merge Conflict Resolution: Classification or Generation?. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1652–1663.

[10] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An exploratory study of cloning in industrial software product lines. In *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, 25–34.

[11] Stefan Fischer, Rudolf Ramler, Lukas Linsbauer, and Alexander Egyed. 2019. Automating test reuse for highly configurable software. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A*. 1–11.

[12] Mohammad Gharehyazie, Baishakhi Ray, and Vladimir Filkov. 2017. Some from here, some from there: Cross-project code reuse in github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 291–301.

[13] Naohiro Kawamitsu, Takashi Ishio, Tetsuya Kanda, Raula Gaikovina Kula, Coen De Roover, and Katsuro Inoue. 2014. Identifying source code reuse across repositories using lcs-based source code similarity. In *2014 IEEE 14th international working conference on source code analysis and manipulation*. IEEE, 305–314.

[14] Simon Larsén, Jean-Rémy Falleri, Benoit Baudry, and Martin Monperrus. 2022. Spork: Structured merge for Java with formatting preservation. *IEEE Transactions on Software Engineering* 49, 1 (2022), 64–83.

[15] Olaf Leßenich, Janet Siegmund, Sven Apel, Christian Kästner, and Claus Hunsen. 2018. Indicators for merge conflicts in the wild: survey and empirical study. *Automated Software Engineering* 25, 2 (2018), 279–313.

[16] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. 2019. Test transfer across mobile apps through semantic mapping. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 42–53.

[17] Wardah Mahmood, Moses Chagama, Thorsten Berger, and Regina Hebig. 2020. Causes of merge conflicts: A case study of elasticsearch. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*. 1–9.

[18] A. Mastropaolo, M. Ciniselli, L. Pascarella, and G. Bavota. 2024. An empirical study on the code refactoring capability of large language models. *arXiv preprint* (2024). arXiv:2401.02320 [cs.SE]

[19] Shane McKee, Nicholas Nelson, Anita Sarma, and Danny Dig. 2017. Software practitioner perspectives on merge conflicts and resolutions. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 467–478.

[20] Na Meng, Miryung Kim, and Kathryn S McKinley. 2013. LASE: locating and applying systematic edits by learning from examples. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 502–511.

[21] Mehdi Mirzaaghaei, Fabrizio Pastore, and Mauro Pezzè. 2012. Supporting test suite evolution through test case adaptation. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 231–240.

[22] M. Mukelabai, P. Borba, and T. Berger. 2021. Semi-automated test-case propagation in fork ecosystems. In *ICSE/NIER*.

[23] M. Mukelabai, C. Derks, J. Krüger, and T. Berger. 2023. To share, or not to share: Exploring test-case reusability in fork ecosystems. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering(ASE)*.

[24] Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N Nguyen, and Hridesh Rajan. 2013. A study of repetitiveness of code changes in software evolution. In *ASE*.

[25] Shengyi Pan, You Wang, Zhongxin Liu, Xing Hu, Xin Xia, and Shanping Li. 2024. Automating zero-shot patch porting for hard forks. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 363–375.

[26] S. Rahman, S. Kuhar, B. Cirisci, P. Garg, S. Wang, X. Ma, A. Deoras, and B. Ray. 2025. UTFix: Change Aware Unit Test Repairing using LLM. *Proceedings of the ACM on Programming Languages* 9, OOPSLA1, Article 85 (2025), Article 85 pages.

[27] Baishakhi Ray and Miryung Kim. 2012. A case study of cross-system porting in forked projects. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.

[28] Baishakhi Ray, Miryung Kim, Suzette Person, and Neha Rungta. 2013. Detecting and characterizing semantic inconsistencies in ported code. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 367–377.

[29] Baishakhi Ray, Meiyappan Nagappan, Christian Bird, Nachiappan Nagappan, and Thomas Zimmermann. 2015. The uniqueness of changes: Characteristics and applications. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 34–44.

[30] Luyao Ren. 2019. Automated patch porting across forked projects. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1199–1201.

[31] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. 2013. Managing cloned variants: a framework and experience. In *SPLC*.

[32] Julia Rubin, Andrei Kirshin, Goetz Botterweck, and Marsha Chechik. 2012. Managing forked product variants. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*. 156–160.

[33] Bowen Shen, Muhammad Ali Gulzar, Fei He, and Na Meng. 2023. A characterization study of merge conflicts in Java projects. *ACM Transactions on Software Engineering and Methodology* 32, 2 (2023), 1–28.

[34] A. Shirafuji, Y. Hoshikawa, Y. Watanobe, and K. Inoue. 2023. Refactoring programs using large language models with few-shot examples. *arXiv preprint* (2023). arXiv:2311.11690 [cs.SE]

[35] Ştefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. 2015. Forked and integrated variants in an open-source firmware project. In *2015 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 151–160.

[36] A. Svyatkovskiy, Y. Zhao, C. Fu, N. Sundaresan, and S. Chandra. 2021. MergeBERT: Program Merge Conflict Resolution via Neural Transformers. In *ESEC/FSE*.

[37] Birgit Vogel-Heuser, Alexander Fay, Ina Schaefer, and Matthias Tichy. 2015. Evolution of software in automated production systems: Challenges and research directions. *Journal of Systems and Software* 110 (2015), 54–84.

[38] Ahmadreza Saboor Yaraghi, Darren Holden, Nafiseh Kahani, and Lionel Briand. 2025. Automated test case repair using language models. *IEEE Transactions on Software Engineering* (2025).

[39] Hang Yin, Hamza Mohammed, and Sai Boyapati. 2024. Leveraging Pre-Trained Large Language Models (LLMs) for On-Premises Comprehensive Automated Test Case Generation: An Empirical Study. In *2024 9th International Conference on Intelligent Informatics and Biomedical Sciences (ICIIBMS)*, Vol. 9. IEEE, 597–607.

[40] Jialu Zhang, Todd Mytkowicz, Mike Kaufman, Ruzica Piskac, and Shuvendu K Lahiri. 2021. Can pre-trained language models be used to resolve textual and semantic merge conflicts? *arXiv preprint arXiv:2111.11904* (2021).

[41] Quanjun Zhang, Chunrong Fang, Siqi Gu, Ye Shang, Zhenyu Chen, and Liang Xiao. 2025. Large Language Models for Unit Testing: A Systematic Literature Review. *arXiv preprint arXiv:2506.15227* (2025).

[42] Tianyi Zhang and Miryung Kim. 2017. Automated transplantation and differential testing for clones. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 665–676.

[43] Shurui Zhou, Bogdan Vasilescu, and Christian Kästner. 2020. How has forking changed in the last 20 years? a study of hard forks on github. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 445–456.