The Fix Is Right at Hand: Fixing Incompatibility Errors Guided by Library Knowledge for Automatic Library Upgrade

ZHUOTONG ZHOU, Fudan University, China
SUSHENG WU, Fudan University, China
JUNPENG ZHAO, Fudan University, China
BIHUAN CHEN*, Fudan University, China
YENQIN HOO, Fudan University, China
YIHENG HUANG, Fudan University, China
YIHENG CAO, Fudan University, China
XIN PENG, Fudan University, China

Third-party libraries (TPLs) play critical roles in modern software development. Upgrading them is crucial for enhanced security and functionality, but often introduces incompatibility errors, caused by breaking changes in library APIs, in client code. Existing approaches rely on predefined migration patterns or API recommendation heuristics, which suffer from limited pattern coverage and ignore the usage context of broken API, leading to incorrect or incomplete fixes. To address these limitations, we propose Librarian, a novel LLM-based approach to automatically fix incompatibility errors when upgrading a dependent library in a client project. The core idea of Librarian is to extract context-aware fix hints from the library codebase, serving as semantic few-shot examples, enabling LLM to generate fixes without relying on predefined patterns. Since LLM may generate an incorrect or incomplete fix, Librarian performs fix refinement based on compilation feedback from the client project. Our evaluation has demonstrated that Librarian achieves a fixing success rate of 84.2%, outperforming the state-of-the-arts by at least 45.3%. Our evaluation has also indicated the practical usefulness of Librarian in fixing incompatibility errors in 32 real-world projects.

CCS Concepts: • **Software and its engineering → Software evolution**; **Software maintenance tools**.

Additional Key Words and Phrases: Library Upgrade, API Breaking Changes, Incompatibility Error Fixing

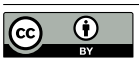
ACM Reference Format:
Zhuotong Zhou, Susheng Wu, Junpeng Zhao, Bihuan Chen, YenQin Hoo, Yiheng Huang, Yiheng Cao, and Xin Peng. 2026. The Fix Is Right at Hand: Fixing Incompatibility Errors Guided by Library Knowledge for Automatic Library Upgrade. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA025 (October 2026), 21 pages. https://doi.org/10.1145/3832116

1 Introduction

Third-party libraries (TPLs) enable developers to reuse common and well-tested functionality and thus significantly boost productivity. Despite these benefits, integrating TPLs can introduce both

*B. Chen is the corresponding author.

Authors' Contact Information: Zhuotong Zhou, Fudan University, Shanghai, China, zhouzt23@m.fudan.edu.cn; Susheng Wu, Fudan University, Shanghai, China, scwu24@m.fudan.edu.cn; Junpeng Zhao, Fudan University, Shanghai, China, jpzhao24@m.fudan.edu.cn; Bihuan Chen, Fudan University, Shanghai, China, bhchen@fudan.edu.cn; YenQin Hoo, Fudan University, Shanghai, China, 22302016004@m.fudan.edu.cn; Yiheng Huang, Fudan University, Shanghai, China, yihenghuang23@m.fudan.edu.cn; Yiheng Cao, Fudan University, Shanghai, China, caoyh23@m.fudan.edu.cn; Xin Peng, Fudan University, Shanghai, China, pengxin@fudan.edu.cn.



ACM 2994-970X/2026/10-ARTISSTA025
https://doi.org/10.1145/3832116

security vulnerabilities and functional defects. A widely recommended and cost-effective way to mitigate these risks is to upgrade the library to its latest released version [13, 19]. However, such upgrades often include breaking changes in library APIs, potentially causing incompatibility errors within client projects [12, 25]. Manually fixing such incompatibility errors is both labor-intensive and time-consuming, underscoring the importance of automatically fixing incompatibility errors.

Existing Approaches. To fix incompatibility errors arising from a library upgrade in a client project, automatic approaches have been proposed and can be classified into two categorizes, i.e., API migration approaches (e.g., [42, 44]) and API recommendation approaches (e.g., [4, 22, 31, 34, 38]). API migration approaches fix error-raising statements through *predefined* migration patterns. For example, MEDITOR [42] extracts migration patterns from historical migration examples in a set of client projects, and LIBCATCH [44] designs migration patterns based on compilation error types. In contrast, API recommendation approaches do not directly modify client code, but instead only recommend alternative APIs to replace the broken ones. Therefore, additional manual effort is still needed for code adaptations. In addition, not all incompatibility errors could be fixed simply by replacing a broken API with an recommended alternative [22]. For example, some incompatibility errors are fixed by modifying the context of the arguments passed to the broken API.

Limitations. We summarize two key limitations of the existing approaches in fixing incompatibility errors in the client project.

- *L1: Fix Pattern Insufficiency.* Existing approaches rely on predefined fix patterns. However, such patterns are often insufficient for fixing diverse type of errors, resulting in incorrect fixes.
- *L2: Broken API Usage Context Insensitivity.* Existing approaches do not consider the specific context in which the broken APIs are used in the client project, resulting in incorrect or incomplete fixes.

Our Approach. To address the limitations of existing approaches, we propose LIBRARIAN, a novel LLM-based approach that assists developers to upgrade dependent libraries in client projects by automatically fixing incompatibility errors introduced by the upgrades. One core idea of LIBRARIAN is to extract fix hints from the library source code itself, whose feasibility is evidenced by our preliminary empirical study. Specifically, the fix hints include the declaration changes of broken APIs as well as the usage context changes of broken APIs in the original and upgraded library versions. Such fix hints serve as semantic few-shot examples, enabling LLM to infer fixes without relying on predefined patterns, thereby addressing the limitation of *L1*.

Another core idea of LIBRARIAN is to perform inter-procedural data-flow slicing on the error-raising statement to obtain the full usage context of the broken API in the client project. Similarly, the usage context change of the broken API in the original and upgraded library versions is also obtained by the same slicing procedure. Such contexts keep the fixing-relevant statements while removing the fixing-irrelevant ones, easing the burden on LLM to infer the fix for the client project, thus addressing the limitation of *L2*. Since LLM may generate an incorrect or incomplete fix, LIBRARIAN also performs fix refinement based on compilation feedback from the client project.

Evaluation. We compared LIBRARIAN with four selected state-of-the-art approaches, SEMDIFF [4], REPFINDER [22], REFACTORINGMINER [34], and LIBCATCH [44], and six baseline LLMs. Our evaluation benchmark consists of 278 library upgrades collected from 190 projects, containing 2,871 incompatibility errors and covering 166 third-party libraries. On average, each library upgrade introduced ten errors. LIBRARIAN successfully fixed all incompatibility errors in 234 library upgrades, and successfully fixed 2,410 errors, while the best state-of-the-art fixed 161 library upgrades and 1,715 errors. This demonstrates an improvement of at least 45.3% and 40.5% over the state-of-the-arts. Moreover, we evaluated LIBRARIAN on 20 new library upgrades that postdate the LLM's training cutoff, and it fixed all incompatibility errors in 15 library upgrades, indicating that its performance is robust to

potential training-data leakage. Finally, to evaluate LIBRARIAN's practical usefulness, we applied it to real-world projects and submitted 57 pull requests, 32 of which were successfully merged.

Contribution. This work makes the following contributions.

- We proposed an LLM-based approach, LIBRARIAN, which extracts fix hints from the library codebase to guide the LLM in effectively fixing incompatibility errors introduced by library upgrades.
- We constructed a new benchmark consisting of 278 real-world library upgrades, each of which includes the introduced incompatibility errors and the developer-provided fixes. Our benchmark covers 166 distinct libraries, covering 33× more libraries than existing benchmarks.
- We conducted extensive experiments to evaluate the effectiveness, efficiency, generality and practical usefulness of LIBRARIAN.

2 Motivating Example

We introduce a breaking change example to illustrate the limitation of existing approaches and motivate the design of LIBRARIAN. As shown in Fig. 1a, upgrading `bcprov-jdk15on` from version 1.62 to 1.64 introduces two breaking changes. First, the method `convertPoint` in class `EC5Util` has been updated and no longer accepts the parameter `withCompression` (at Lines 2-3 in Fig. 1a). Second, the constructor of `X9ECParameters` now needs an instance of `X9ECPoint` instead of `ECPoint` as its parameter (at Lines 6-7 in Fig. 1a). The first breaking change results in a compilation error in the client project (at Line 7 in Fig. 1b). To fix it, the developer removes the argument `withCompression` from method `convertPoint`. Unfortunately, this fix introduces another compilation error due to the second breaking change. To further fix it, the developer wraps the return value of `convertPoint` with the constructor of class `X9ECPoint` (at Line 8 in Fig. 1b), providing the previously removed argument `withCompression` at this stage. As a result, all arguments passed to the constructor of class `X9ECParameters` match its updated parameter requirements. By performing these two sequential modifications, the developer can completely fix the incompatibility errors in the client project.

```
1   /* org.bouncycastle.jcajce.provider.asymmetric.util.EC5Util */
2 - public static ECPoint convertPoint(ECCurve curve, ECPoint point, Boolean withCompression);
3 + public static ECPoint convertPoint(ECCurve curve, ECPoint point);
4
5   /* org.bouncycastle.asn1.x9.X9ECParameters */
6 - public X9ECParameters(ECCurve curve, ECPoint g, X9ECPoint g, BigInteger n, BigInteger h, byte[] seed);
7 + public X9ECParameters(ECCurve curve, X9ECPoint g, BigInteger n, BigInteger h, byte[] seed);
```

(a) The breaking changes in the library during upgrade

```
1 public class BCUtil {
2     public static X962Parameters getDomainParametersFromName(ECParameterSpec ecSpec,
3                                                              boolean withCompression) {
4         ECCurvecurve=EC5Util.convertCurve(ecSpec.getCurve());
5         X9ECParameters ecP = new X9ECParameters(
6             curve,
7 -           EC5Util.convertPoint(curve, ecSpec.getGenerator(), withCompression),
8 +           new X9ECPoint(EC5Util.convertPoint(curve, ecSpec.getGenerator()), withCompression),
9             ecSpec.getOrder(),
10            BigInteger.valueOf(ecSpec.getCofactor()),
11            ecSpec.getCurve().getSeed()
12        );
13        /* Other Statements */
14    }
15}
```

[ERROR] no suitable method found for ***convertPoint(org.bouncycastle.math.ec.ECCurve, ja.security.spec.ECPoint, boolean)***
[ERROR] Incompatible types: ***org.bouncycastle.math.ec.ECPoint*** cannot be converted to ***org.bouncycastle.asn1.x9.X9ECPoint***

(b) Compilation errors and the developer fix in the client project

org.bouncycastle.jcajce.asymmetric.util.EC5Util.convertPoint(ECCurve, ECPoint, boolean) will be removed, use *@link* ***#convertPoint(ECCurve, ECPoint)*** instead. — **Library JavaDoc 1**

org.bouncycastle.asn1.x9.X9ECParameters(ECCurve, ECPoint, BigInteger, BigInteger, byte[]) Use constructor taking an ***X9ECPoint*** instead. — **Library JavaDoc 2**

(c) Fix hint from the library API JavaDoc

```
1   /* org.bouncycastle.jcajce.provider.asymmetric.ec.AlgorithmParametersSpi */
2   import org.bouncycastle.asn1.x9.X9ECPoint;
3
4   public class AlgorithmParametersSpi {
5       protected byte[] engineGetEncoded(String format) {
6 -         ECParameterSpec ecSpec = EC5Util.convertSpec(ecParameterSpec, false);
7 +         ECParameterSpec ecSpec = EC5Util.convertSpec(ecParameterSpec);
8           X9ECParameters ecP = new X9ECParameters(
9               ecSpec.getCurve(),
10 -             ecSpec.getG(),
11 +             new X9ECPoint(ecSpec.getG(), false),
12              ecSpec.getN(),
13              ecSpec.getH(),
14              ecSpec.getSeed());
15      }
16  }
```

(d) Fix hint from library API usage changes during upgrade

Fig. 1. Fixing compilation errors after upgrading dependent library `bcprov-jdk15on` from version 1.62 to version 1.64

API migration approaches cannot generate the complete fix for this client project. MEDITOR [42] does not collect the migration examples related to the library `bcprov-jdk15on`, and hence cannot fix the error. LIBCATCH [44] removes the argument `withCompression` and fixes the error caused by the first breaking change by applying predefined migration patterns. However, it cannot infer that the

return value of `convertPoint` should be wrapped in an instance of `X9ECPoint` with the argument `withCompression`, because this modification lies outside its migration patterns. Consequently, it cannot fix the error caused by the second breaking change.

API recommendation approaches cannot generate the fix for the client project. For the first breaking change, RefactoringMiner [34] correctly recommends the method `convertPoint (ECCurve, ECPoint)` based on the library API declaration changes. Rep-Finder [22] recommends this method by leveraging library JavaDoc (i.e., JavaDoc 1 in Fig. 1c). However, human effort is required to adapt the recommended method to the client project. Besides, these approaches fail to provide a fix for the second breaking change, because the error cannot be fixed by recommending an alternative method.

The fix pattern for the second breaking change is difficult to predefine. However, by examining the code changes between versions 1.6.2 and 1.6.4 of `bcprov-jdk15on`, we observe that in version 1.6.2, the method `engineGetEncoded` in the library invoked the constructor of `X9ECParameters` (at Line 8 in Fig. 1d), and in version 1.6.4, the usage of the constructor of `X9ECParameters` was changed (at Lines 6-7 and 10-11 in Fig. 1d). The semantics of this updated usage are similar to the developer's fix for the client project in Fig. 1b. This observation suggests that the fix pattern for a broken API (i.e., the API causing the compilation error) can be inferred by analyzing how the API's usage is updated in the library code itself. Moreover, recent studies [8, 11, 18] show that large language models (LLMs) have strong capabilities in understanding code semantics and editing code, therefore we attempt to provide LLMs with fix hints, such as these usage changes, to guide the generation of fixes for the client project.

Limitations. We summarize two limitations of existing approaches based on the above example.

L1: Fix Pattern Insufficiency. Existing approaches rely heavily on predefined fix patterns to fix errors. In *API migration approaches*, their predefined migration patterns are often insufficient to handle the diverse types of breaking changes. In *API recommendation approaches*, the recommended APIs only serve as replacements in the migration patterns for undefined APIs. However, not all incompatibility errors can be fixed by replacing a broken API with a recommended alternative. As a result, these approaches frequently either cannot generate a fix or produce an incorrect fix.

L2: Broken API Usage Context Insensitivity. Existing approaches attempt to fix broken APIs without adequately considering the usage context in client code. *API migration approaches* apply migration patterns directly to the error-raising statements without accounting for the API usage context within the client project. *API recommendation approaches* only recommend alternative APIs and entirely ignore how the API is used in the client project. Consequently, these approaches often fail to produce correct or complete fixes.

3 Preliminary Study

Motivated by the example in Sec. 2, we aim to leverage fix hints in the library to guide the fixing of incompatibility errors in client projects. However, before doing so, we first demonstrate that such fix hints are indeed available in real-world libraries, rather than being limited to a few specific libraries. Thus, we conduct a preliminary study to quantify how often such fix hints are available in practice.

Fix Hint Definition. We leverage three types of fix hints: (1) API JavaDoc, (2) API signature changes, and (3) API usage context changes. We define them as follows.

- **API JavaDoc (JavaDoc):** The official API documentation published alongside the library. For example, Fig. 1c shows the JavaDoc comments corresponding to the two APIs.
- **API Signature Change (Sign. Change):** The change to the signature of a library API. For a class, its signature includes the class name. For a method, its signature includes the method name, its parameter list, and its declaring class. For a field, its signature includes the field name and type, and its declaring class. For example, Fig. 1a illustrates two pairs of method signature changes.

- **API Usage Context Change (Usage. Change):** The change to how an API is used within the library itself. For example, Fig. 1d shows how the usage of the constructor `X9ECParameters` has changed in the library codebase.

API JavaDoc and API signature change have been widely used in prior work [2, 22, 34] and shown to be valuable for helping developers fix incompatibility errors. As broken APIs are often invoked by other APIs in the library or exercised in the library's test cases, their usage contexts can also change across versions, which may provide valuable hints for resolving these incompatibility errors.

Benchmark Construction. Existing library upgrade benchmarks are limited in both scale and diversity. For example, the benchmark used in [44] covers only five libraries, and the one in [42] includes only four libraries. To mitigate the potential bias caused by such limited coverage, we construct a new, large-scale benchmark consisting of real-world library upgrade samples that introduce incompatibility errors in client projects. By expanding far beyond those early works, we ensure that our benchmark captures a larger number of libraries and a more comprehensive spectrum of API breaking changes. Our construction process consists of the following three steps.

Step1: Library Upgrade Commit Extraction. We collected 425 Java Maven projects from GitHub with at least 5 stars or 10 commits. On average, each project contains 36.6K lines of code, and 27.0% of the projects consist of multiple modules. We then extracted all commits in the projects that modified the `<version>` element in the `pom.xml` file. Here, we exclued the version ranges (e.g., `[1.0.0, 2.0.0)`) because most developers specify a fixed version in `pom.xml` in practice [7]. A commit was classified as an upgrade if the version number increased according to semantic versioning; for non-semantic versioning change, we manually verified whether the change constituted an upgrade. This yielded 2,520 library upgrade commits.

Step2: Incompatibility Error Identification. Since not all library upgrades induce compatibility errors, for each library upgrade commit, we decomposed it into two parts of changes; i.e, *Part 1:* the version change in `pom.xml`, and *Part 2:* any additional modifications. We first reverted each Java project to the pre-upgrade version, applied only *Part 1* to the pre-upgrade version, and then manually compiled the project. A failed compilation indicated incompatibility errors. Using this procedure on the extracted library upgrade commits, we identified 2,871 incompatibility errors in 278 library upgrade commits across 190 projects. The remaining library upgrade commits were excluded as they did not introduce any compilation error or the project already failed to compile before the upgrade.

Step3: Fixing Change Extraction. Following prior work [35], we found fixing changes either in the upgrade commit or in the subsequent 3 commits. We then applied *Part 2* to the pre-upgrade version. If *Part 2* did not fix incompatibility errors, we incrementally applied later commits until the project compiled successfully. We regarded the changes in the commits as initial fixing for incompatibility errors. Since these commits often included unrelated changes, we iteratively removed and re-added modified statements, retaining only those whose removal reintroduced the errors. This yielded a minimal, sufficient set of fixing changes.

Following this process, we collected 278 library upgrade samples from 190 projects, covering 166 distinct libraries, which induced 2,871 compilation errors, thereby constructing our benchmark $\mathcal{B}$. Table 1 reports the 20 types of API breaking changes covered in $\mathcal{B}$ and their frequencies across the 278 library upgrades, spanning the class, method, and field levels. Formally, each sample $b \in \mathcal{B}$ is denoted as a six-tuple; i.e., $b = \langle client, snapshot, lib, v_o, v_u, pairs \rangle$.

- $client$: the identifier of the client project in which the upgrade causes incompatibility errors.
- $snapshot$: the snapshot of the client project at the precise commit just before the library upgrade.
- lib: the dependent library being upgraded.
- v_o: the original (pre-upgrade) version of the library.
- v_u: the upgraded (post-upgrade) version of the library.

Table 1. The types and frequencies of breaking changes in the benchmark

API Level	API Breaking Change Type and Frequency
Class-Level (5)	Class QualifiedName Modification (69) · Class Removal (33) · Class Accessibility Modification (5) · Class Type Modification (3) · Class Generic Type Modification (3)
Method-Level (12)	Method Removal (48) · Method Parameter Type Modification (47) · Abstract Method Addition (31) · Method Name Modification (27) · Method Return Type Modification (22) · Abstract Method Parameter Type Modification (10) · Method Exception Modification (8) · Method Accessibility Modification (7) · Abstract Method Removal (7) · Abstract Method Return Type Modification (6) · Method Ambiguity (3) · Abstract Method Name Modification (2)
Field-Level (3)	Field Removal (14) · Field Accessibility Modification (2) · Field Name Modification (1)

Table 2. The coverege and uniqueness for each type of fix hint

Type	JavaDoc	Sign. Change	Usage. Change	Total
Coverage (#)	30	76	242	254
Uniqueness (#)	1	5	96	102

- *pairs*: a set of $\langle e,f \rangle$ tuples, where e is a specific compilation error triggered by upgrading from v_o to v_u, and f is the minimal set of changes to fix e.

Metrics. To quantify the availability of each type of fix hint in the library, we introduce two metrics, i.e., *coverage* and *uniqueness*. Specifically, *coverage* measures the number of upgrade samples for which a fix hint of that type is available in the library, whereas *uniqueness* measures the number of upgrade samples for which only the fix hint of that type can be extracted from the library.

We compute both metrics on our constructed benchmark. For each type of fix hint, we apply the heuristic rules mentioned in prior work [2, 22, 34] to determine whether that fix hint is available in the library for each upgrade sample. For *JavaDoc*, we use regex patterns from RepFinder [22] to scan JavaDoc comments between the original and upgraded versions for deprecation annotations that reference the broken APIs. If such deprecation annotations are detected, we regard *JavaDoc* as available. For *Sign. Change*, we first locate the broken API in the original library version, and use RefactoringMiner [34] to identify the corresponding updated API in the upgraded version. If the updated API is identified, we regard *Sign. Change* as available. For *Usage. Change*, we use JDT [9] to extract the APIs that use the broken API in the original library version, denoting these extracted APIs as *shadow* APIs. For example, in Fig. 1d, the method `engineGetEncoded` is a shadow API of the broken API `X9ECParameters`. We then use RefactoringMiner [34] to map each shadow API to its counterparts in the upgraded library version. If a shadow API can be matched in the upgraded version, we regard *Usage. Change* as available.

Results. As shown in Table 2, *Usage. Change* exhibits the highest coverage of library upgrades and contributes the majority of unique fix hints, indicating that only this type of fix hint is available from the library. *Sign. Change* covers 76 library upgrades and contributes 5 unique fix hints. *JavaDoc* provides fix hints for only 30 library upgrades and offers just 1 unique fix hint. These results indicate that, in practice, for 91.4% of the library upgrades, at least one fix hint for upgrade-induced errors can be extracted from the library itself, thereby supporting the soundness of our approach (Sec. 4).

4 Approach

We propose Librarian, an library knowledge guided LLM-based approach that combines client-side broken API usage context with library-side broken API fix hints to fix incompatibility errors in client projects. It takes a client project that has upgraded a dependent library *lib* from the original version v_o to the upgraded version v_u as the inputs. It works in three phases to fix the incompatibility errors in the client project after upgrading the library. Fig. 2 provides an overview of Librarian.

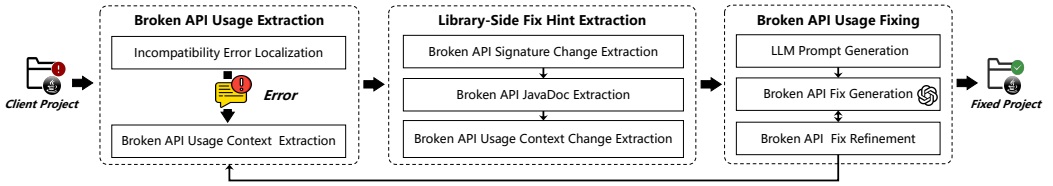


Fig. 2. Approach overview of Librarian

- **Phase 1: Broken API Usage Extraction.** Given the client project after library upgrade, Librarian first locates every compilation error in the client project and the broken API that causes it. Then, it applies a heuristic error-fixing order; i.e., it fixes errors in superclass declarations before subclass declarations, then fixes errors in a class declaration in the order of class-level, method-level, and field-level errors to minimize cascading issues (Sec. 4.1.1). Given an error, Librarian performs inter-procedural data-flow slicing, starting from the error-raising statement, to extract the usage context Err_{ctx} of the broken API within the client project (Sec. 4.1.2).
- **Phase 2: Library-Side Fix Hint Extraction.** Given the library's source code and test cases of the original and upgraded versions, Librarian extracts signature change of the broken API $Diff_{sig}$ (Sec. 4.2.1). Then, it traverses every intermediate library version from the original to the upgraded version to extract deprecation JavaDoc comment associated with the broken API Dep_{doc} (Sec. 4.2.2). Finally, it extracts usage context changes for the broken API between the original and upgraded library versions (Sec. 4.2.3). Specifically, it identifies all *shadow* APIs (i.e., APIs that uses the broken API) in the original version, and runs inter-procedural data-flow slicing on the call-site statements of the broken API to obtain the usage contexts. These shadow APIs are then mapped to their counterparts in the upgraded version, and inter-procedural data-flow slicing is again conducted to the updated call-site statements, extracting the updated usage contexts. However, these usage contexts extracted from the library may differ from those in the client project. Hence, Librarian computes semantic similarity between each library-side usage context and client-side usage context to select the most relevant library-side usage context change $Diff_{ctx}$. The extracted signature change of the broken API, deprecation JavaDoc comment and the usage context change of the broken API at the library-side serve as the fix hints in the following phase.
- **Phase 3: Broken API Usage Fixing.** Using the broken API usage context Err_{ctx} in the client project, Librarian prunes client code files $\mathcal{F}_{comp}$ that contain the usage context of the broken API and require fixing by the LLM. Then, $\mathcal{F}_{comp}$, together with the extracted fix hints, forms a prompt for the LLM to generate an initial fix (Sec. 4.3.1 and 4.3.2). After applying the fix, Librarian performs fix refinement based on the client project's compilation feedback (Sec. 4.3.3). If the original error persists, it re-instructs the LLM to generate a new fix. If the error is fixed but new errors emerge, it returns to *Phase 1* to fix each new error. Once the fix refinement process finishes, Librarian re-identifies the remaining errors in the client project, and proceeds to fix the next error until Librarian has attempted to fix all errors in the client project.

4.1 Broken API Usage Extraction

Given a client project after library upgrading, Librarian first localizes the incompatibility errors, and then extracts the broken API usage context for each incompatibility error.

4.1.1 Incompatibility Error Localization. Given a client project after upgrading a library, Librarian first retrieves all library JAR files based on the client project's dependency tree, and uses them to configure the AST parsing environment. Then it uses JDT to generate type-bound ASTs for each source file in the client project, and collects compilation errors reported in those ASTs. Since these ASTs are bound to the library's type definitions, Librarian can also extract the signatures of the

APIs responsible for each compilation error. We refer to the APIs causing compilation errors as broken APIs. There are three kinds of broken APIs: classes, methods, and fields. We define each error as e, which is a 4-tuple $\langle type_e, line_e, msg_e, sig_e \rangle$. Here $type_e$ denotes the type of the error (e.g., `Undefined Method`), $line_e$ denotes the line number in the source file where the error occurs, msg_e denotes the error message, and sig_e denotes the signature of the broken API causing the error.

Since a client project can have multiple errors, the error-fixing order may affect the effectiveness of the fixes. For example, if a superclass in the client project inherits from an undefined class, all of its subclasses will also encounter errors. Therefore, Librarian defines a heuristic error-fixing order. It identifies all class declarations that contain errors, and uses the implementation and inheritance relations among those class declarations to determine the fixing order. It first fixes errors in superclass or interface declarations, and then fixes errors in subclass declarations. Then, for errors that appear within a single class declaration, Librarian categorizes the errors into class-level, method-level, and field-level based on $type_e$. For example, if $type_e$ is `Undefined Method`, the error is method-level. Librarian follows the order of fixing class-level errors first, then method-level, and finally field-level errors. This is because class-level errors can cascade into additional method-level and field-level errors. Additionally, method-level errors have broad usage contexts that may involve field-level errors. Thus, method-level errors are prioritized for fixing over field-level errors. Librarian follows this heuristic error-fixing order to fix errors in the client project one by one.

4.1.2 Broken API Usage Context Extraction. Given an error e in the client project, Librarian first extracts the error-raising statement $stmt_e$ based on $line_e$. If $stmt_e$ is located outside any method body (e.g., field declaration), Librarian directly uses $stmt_e$ as the usage context of the broken API, since such statements usually have no dependency relations with other statements. If $stmt_e$ appears inside a method body, it typically depends on other statements in that body. In such cases, $stmt_e$ alone might not sufficiently capture the broken API usage context in the client project. To address this, Librarian uses Soot [16] to extract the usage context statements for $stmt_e$ by intra-procedural data-flow slicing. Librarian performs both forward and backward data-flow analyses on every identifier in $stmt_e$. The statement $stmt_e$ and statements reached by backward and forward data-flow analyses are collected into an initial statement set, denoted as S_b.

However, most API usage context spans multiple methods, creating a risk that inter-procedural data-flow through variables remains untracked [32, 36]. To address this, Librarian extends S_b with further inter-procedural data-flow slicing as follows.

- *Call Graph Generation.* Librarian leverages Soot to construct the call graph of the client project, and identifies all direct callers and callees of the method that contains $stmt_e$.
- *Extended Backward Analysis.* Librarian traces data-flow backward from each statement $s \in S_b$ into each caller, which yields a broader usage context in each caller, denoted as $S^i_{caller} (i = 1, \ldots, n)$, where n is the number of callers with a data flow to s.
- *Extended Forward Analysis.* Similarly, Librarian performs forward data-flow from each statement $s \in S_b$ into each callee, yielding a broader usage context in each callee, denoted as $S^j_{callee} (j = 1, \ldots, m)$, where m is the number of callees that use data from s.

The final usage context of the broken API in the client project is composed of S_b and the extended slices from the callers and callees, denoted as $Err_{ctx} = \left\{ S_b \ , \ \bigcup_{i=1}^{n} S^i_{caller} \ , \ \bigcup_{j=1}^{m} S^j_{callee} \right\}$, which captures a full usage context of the broken API in the client project.

4.2 Library-Side Fix Hint Extraction

Motivated by our motivating example and preliminary study, Librarian extracts the signature changes, JavaDoc comments, and usage context changes of the broken API from the library, and leverages them as fix hints during the fixing process.

4.2.1 Broken API Signature Change Extraction. Librarian first extracts the API signature change of the broken API in the original version and its corresponding updated API in the upgraded version. The definition of the API signature is given in Sec. 3. To achieve this, it leverages RefactoringMiner [34] to extract API signature changes between the original and upgraded library source code. Specifically, given the library source code in the original and upgraded version, RefactoringMiner analyzes their differences to identify which refactoring operation has been applied to the broken API (e.g., `Move and Rename Method`), and reports the corresponding updated API after the refactoring [15, 34]. The updated API is the counterpart of the broken API in the upgraded version, and its signature is denoted as sig_{post}. If RefactoringMiner detects multiple refactoring operations on the broken API, Librarian considers all reported updated APIs. For example, at Line 6 in Fig. 1a, RefactoringMiner detects that the constructor `X9ECParameters` has been refactored through a `Change Parameter Type` operation, and reports that its second parameter has been changed from `ECPoint` to `X9ECPoint`. The change from sig_e to sig_{post} captures the syntactic signature change of the broken API, denoted as $Diff_{sig} = \langle sig_e, sig_{post} \rangle$.

4.2.2 Broken API JavaDoc Extraction. Librarian incorporates JavaDoc comment extracted from the library to supplement the syntactic signature change. Specifically, Librarian uses JDT to locate the API declaration whose signature matches sig_e within the library source code from the original version to the upgraded version. If the matched API declaration is marked with the `@Deprecated` annotation, it then retrieves the associated JavaDoc comment, denoted as Dep_{doc}. Unlike RepFinder [22], which extracts keywords from comment using regular expression, Librarian preserves the entire textual content in Dep_{doc} as the fix hint, thus maintaining complete semantic context for API replacement recommendation. For example, JavaDoc 2 shown in Fig. 1c in Sec. 2 presents a deprecation comment. Although the comment does not explicitly state the exact name of the replacement API, it recommends to use the constructor which includes the parameter `X9ECPoint`. Essentially, the extracted $Diff_{sig}$ and Dep_{doc} capture the syntactic-level and semantic-level broken API signature change respectively, serving as the fix hint for the LLM to generate a fix for the error e.

4.2.3 Broken API Usage Context Change Extraction. Librarian extracts usage context changes of the broken API from the library, since such an API is often used by other APIs in the library or exercised in test cases. Compared to the fix hint extracted from the broken API signature changes and JavaDoc comments, the broken API usage context changes provide a more practical and flexible fix hint, as demonstrated by our preliminary study. The process unfolds in three steps. First, Librarian extracts shadow APIs by locating call-sites to the broken API in the original version and mapping shadow APIs to their upgraded counterparts. Next, for each shadow API, it extracts only those code relevant to the usage context of the broken API; and for its upgraded counterpart, it extracts the corresponding updated usage context, uncovering the usage context change. Finally, Librarian selects the shadow APIs to retain only those whose extracted usage contexts are semantically similar to the actual broken API usage context in the client project. By these three steps, Librarian extracts the usage context changes of the broken API, making it practically applicable for fixing the error e.

Step 1: Shadow API Extraction. Librarian first extracts all shadow APIs from the original version of the library source code and its test cases. To achieve this, Librarian uses JDT to construct the AST of every API in the original version. It then traverses each AST to locate nodes containing calls to other APIs. Specifically, Librarian locates three kinds of AST nodes, i.e., `Type` node, indicating a

call to class API; `MethodInvocation` node, indicating a call to method API; and `FieldAccess` node, indicating a call to field API. For each located node, Librarian extracts the signature of the API being called. If the extracted signature from the node matches sig_e, the node is identified as a call-site node, and the statement containing this node is identified as the call-site statement to the broken API, denoted as $stmt_c$. The API that contains $stmt_c$ is marked as a shadow API, denoted as API_{shadow}. After obtaining API_{shadow}, Librarian identifies updated counterpart of API_{shadow} in the upgraded version, following the procedure in Sec. 4.2.1. This updated shadow API is denoted as $API_{shadow}{}'$. If multiple updated shadow APIs are found, Librarian considers all of them. In this way, Librarian obtains a set of shadow API pairs. Each pair is denoted as a 2-tuple $\langle API_{shadow}, API_{shadow}{}' \rangle$. Our preliminary study shows that, for most broken APIs, such a shadow API pair can be found. If no matching API'_{shadow} can be found, the shadow API is deemed inapplicable and the pair is excluded.

Step 2: Usage Context Change Extraction. Simply treating the entire statements of API_{shadow} and $API_{shadow}{}'$ as representative of broken API usage context change can include irrelevant changes to the usage context of the broken API. To address this, Librarian extracts only the relevant statements of the shadow API that exhibit data-flow relations with the call-site of the broken API. Specifically, Librarian performs inter-procedural forward and backward data-flow slicing on $stmt_c$, as introduced in Sec. 4.1.2. As a result, for each API_{shadow}, Librarian extracts its usage context of the broken API, denoted as $\mathcal{S}_{shadow}$. To extract the updated API usage context from $API_{shadow}{}'$, Librarian first maps $stmt_c$ to the corresponding statement in $API_{shadow}{}'$. Specifically, given the API declarations of API_{shadow} and $API_{shadow}{}'$, Librarian uses GumTree [14] to compute an AST-level mapping between the two declarations by matching structurally similar AST nodes in their respective ASTs [10]. Based on this mapping, Librarian identifies the statement in $API_{shadow}{}'$ corresponding to $stmt_c$. However, $stmt_c$ might be removed in $API_{shadow}{}'$, which results in a mismatch. To address this, Librarian iteratively maps the dominant statement of $stmt_c$ to its corresponding statement in the $API_{shadow}{}'$ until reaching the entry point of API_{shadow}. The mapped call-site statement or the mapped dominant statement in $API_{shadow}{}'$ is denoted as $stmt_c{}'$. For example, given the `engineGetEncoded` method declarations in Fig. 1d (i.e., Lines 5-15) for both the original and upgraded versions, Librarian maps the call-site statement (i.e., Lines 8-14) in the original version to its corresponding statement in the upgraded version. After such mapping, Librarian then runs inter-procedural forward and backward data-flow slicing on $stmt_c{}'$ in $API_{shadow}{}'$, resulting in the updated usage context, denoted as $\mathcal{S}_{shadow}{}'$. The extracted $\mathcal{S}_{shadow}$ and $\mathcal{S}_{shadow}{}'$ constitute the usage context change of the broken API, denoted as $Diff_{ctx} = \langle \mathcal{S}_{shadow}, \mathcal{S}_{shadow}{}' \rangle$.

Step 3: Usage Context Selection. To select the broken API usage contexts extracted in the library that are semantically similar to the broken API usage context in the client project as the fix hint, Librarian performs a semantics-based ranking over all extracted usage contexts. Specifically, Librarian uses a CodeBERT-based model pretrained on the code clone detection task [3], which is demonstrated to have a good understanding of source code, to calculate the semantic similarity score between each usage context of the broken API extracted from the library and the broken API usage context in the client project. Compared with LLM-based models, this model is more lightweight and cost-efficient. Librarian ranks all library-side usage contexts of the broken API in descending order of their similarity, and selects only the top τ_{top} of them. The selected broken API usage contexts in the library $\mathcal{S}_{shadow}$ and their upgraded counterparts $\mathcal{S}_{shadow}{}'$ serve as high-quality usage context changes of the broken API for the LLM, thereby providing examples whose contexts are highly semantically similar to the broken API usage context of the client project.

4.3 Broken API Usage Fixing

Librarian uses LLM, which possesses a semantic understanding capability for code, to infer the fix pattern behind the extracted fix hints, and generate a fix for the broken API usage in client project.

Prompt Template

System Role:
You are an experienced **[% $lang$ %]** developer. After upgrading the dependent library **[% lib %]** in a client project from version **[% v_o %]** to version **[% v_u %]**, the client project encounters incompatibility errors.

Client-Side Broken API Usage Context:
1. **Broken File Code.** The files in the client project currently containing an error and requiring a fix: **[% $\mathcal{F}_{\text{comp}}$ %]**
2. **Broken Statement.** The exact statement within the broken files causing the error: **[% $stmt_e$ %]**
3. **Error Message.** The compilation error message: **[% msg_e %]**

Library-Side Broken API Fix Hints:
1. **Broken API Signature Change.** The signature change of the broken API: **[% $Diff_{sig}$ %]**
2. **Broken API Usage Context Change.** Examples illustrating how to fix the broken API usage: **[% $Diff_{ctx}$ %]**
3. **JavaDoc Comment.** Guidance from the library's JavaDoc describing how to adapt the broken API: **[% Dep_{doc} %]**

Task:
1. Fix the provided **Broken File Code** to fix the compilation error using the provided inputs.
2. Only return the fixed **Broken File Code** in a fenced code block without any additional explanation or comment.

Fig. 3. Prompt template for generating the fix

4.3.1 LLM Prompt Generation. Librarian generates a prompt to guide the LLM to generate a fix for the broken API usage. Fig. 3 shows the prompt template, denoted as P_{fix}. It consists of four parts, i.e., *system role*, *client-side broken API usage context*, *library-side broken API fix hints*, and the fixing *task*.

System Role. This part contains descriptions about the intended role of LLM and the background of the task, assisting LLM in better understanding the task. Specifically, *lang* is the programming language (e.g., Java) of the task. *lib* contains the library coordinate (e.g., group ID and artifact ID in Maven), and v_o and v_u are the original and upgraded versions of the dependent library, respectively.

Client-Side Broken API Usage Context. It provides the broken API usage context in the client project (Sec. 4.1.2). *Broken File Code* $\mathcal{F}_{\text{comp}}$ is a set of pruned code files that contain the broken API usage context that the LLM aims to fix. Librarian first identifies all code files in the client project that contain the sliced statement in Err_{ctx}. However, there are also code irrelevant to the broken API usage context in those files, which can mislead the LLM. To address this, Librarian constructs the AST of each file, locates each class declaration in the AST, and removes all the methods and field declarations within the class declaration that do not contain sliced statements in Err_{ctx}. Subsequently, Librarian wraps each processed, streamlined file with a delimiter, yielding the final pruned broken code files, denoted as $\mathcal{F}_{\text{comp}}$. *Broken Statement* $stmt_e$ is the error-raising statement that helps the LLM locate the error position in broken files. *Error Message* msg_e is the content of the error message, providing the detailed reason for the compilation error.

Library-Side Broken API Fix Hints. It provides the fix hints from the library (Sec. 4.2). *Broken API Signature Change* $Dif f_{sig}$ and *Broken API Usage Context Change* $Dif f_{ctx}$ are the signature change and usage context change of the broken API at the library side, respectively. *JavaDoc Comment* Dep_{doc} is the JavaDoc comment guiding how to adapt to the broken API.

Task. It specifies the detailed task and output format for the LLM.

4.3.2 Broken API Fix Generation. Librarian uses the prompt P_{fix} to instruct the LLM to fix the broken API usage in the pruned code files $\mathcal{F}_{comp}$. The fixed code files provied by LLM are denoted as $\mathcal{F}_{llm}$. It then traverses each fixed file in $\mathcal{F}_{llm}$, locates the corresponding original file in the client project by matching package name and class declaration in the file, and replaces the original broken file with the generated fixed file. To ensure consistency, Librarian restores any previously pruned method and field declarations to the class declarations in the fixed file. By this process, Librarian applies the fix generated by the LLM to the client project, resulting in an intermediate client project.

4.3.3 Broken API Fix Refinement. The LLM may not always generate a fully correct fix in one single attempt. For example, it correctly updates the return type of a method (e.g., changing `int` to `long`), but this modification can introduce type mismatches or other errors at call sites that invoke the updated method. These newly introduced errors must then be fixed as well. Therefore, Librarian iteratively refines the fix based on compilation feedback from the intermediate client project. During each refinement iteration, Librarian first checks whether the original error still persists. However, simply relying on error's line number to determine if an error has been fixed is not reliable, because the line number of the code can shift during the fix process. To address this, Librarian generates an error signature by concatenating the statement where the error occurred with its immediately preceding and following statements, after removing tabs and whitespaces. Using this error signature, Librarian re-identifies the error in the intermediate client project, and checks whether the same error is still present in the intermediate client project. If it finds an error with both a matched broken API and the same error signature, it concludes that the original error persists. In this case, it uses the original prompt to re-instruct the LLM to generate a new fix.

If the original error has been fixed but introduces new errors, Librarian re-executes the *broken API usage context extraction* and *library-side fix hint extraction* steps for each new error, and generates a new prompt for each new error. To ensure that the LLM is aware of the fix for the original error, Librarian adds the original error's fix into the new prompt, and provides this augmented prompt to the LLM to generate a fix for each new error. This enables the LLM incrementally to fix every new error and ultimately produce a complete fix.

The refinement process continues until the original error has been fixed without introducing additional new errors in the client project, or until a maximum iteration threshold τ_{max} is reached. After fixing this error, Librarian re-identifies the incompatibility errors in the client project. If there are still errors that Librarian has not yet attempted to fix, it continues to fix the next one according to the fixing order outlined in Sec. 4.1.1. Otherwise, it finishes the fixing process for the client project.

5 Evaluation

We first introduce the evaluation setup, and then report the results for each research question.

5.1 Evaluation Setup

We have implemented Librarian in 22.8 K of Java code. We used the GPT-5-mini [29], a powerful LLM, as the inference model in Librarian. To evaluate the effectiveness and efficiency of Librarian in fixing incompatibility errors for real-world projects, we designed six research questions. We ran our experiments on a Linux workstation with an Intel(R) Xeon(R) Silver 4316@2.30 GHz and 256 GB of RAM, running Ubuntu 22.04.4 LTS with JDK 17.

- **RQ1 Effectiveness Evaluation.** What is the effectiveness of Librarian?
- **RQ2 Ablation Study.** How is the contribution of each component in Librarian?
- **RQ3 Parameter Sensitivity Analysis.** How do the configurable parameters affect Librarian?
- **RQ4 Efficiency Evaluation.** What is the time overhead of Librarian?
- **RQ5 Generality Evaluation.** How is the generality of Librarian for new projects?
- **RQ6 Usefulness Evaluation.** How is the practical usefulness of Librarian in real world?

Benchmark. We evaluate Librarian on the benchmark introduced in Sec. 3. The benchmark consists of 278 real-world library upgrade samples collected from 190 Java libraries and covers a total of 2,871 incompatibility errors. Compared with existing benchmarks [42, 44], our benchmark offers 33× broader library coverage. It includes 20 distinct types of API breaking changes, spanning the class, method, and field levels, as detailed in Table 1.

Table 3. Results of our effectiveness evaluation compared to the state-of-the-art

	SemDiff	RepFinder	RefactoringMiner	LibCatch	**Librarian**
Fix_{up}	3.6% (10 / 278)	23.0% (64 / 278)	27.3% (76 / 278)	57.9% (161 / 278)	**84.2% (234 / 278)**
Fix_{er}	5.7% (164 / 2,871)	28.9% (831 / 2,871)	39.4% (1,132 / 2,871)	59.7% (1,715 / 2,871)	**83.9% (2,410 / 2,871)**

Table 4. Results of our effectiveness evaluation compared to the baseline LLMs

	GPT-5-mini	DeepSeek-V3.2	Gemini-2.5-flash	Grok-Code	Qwen3-Coder	Llama3-70B
Fix_{up}	56.5% (157 / 278)	40.6% (113 / 278)	41.0% (114 / 278)	43.2% (120 / 278)	43.5% (121 / 278)	23.7% (66 / 278)
Fix_{er}	57.2% (1,641 / 2,871)	45.5% (1,306 / 2,871)	51.5% (1,480 / 2,871)	42.0% (1,207 / 2,871)	49.6% (1,423 / 2,871)	25.1% (722 / 2,871)

State-of-the-Art Selection. We compared Librarian with four state-of-the-art approaches, including one *API migration approach*, i.e., LibCatch [44], and three *API recommendation approaches*, i.e., SemDiff [4], RepFinder [22], and RefactoringMiner [34]. Notice that we excluded Meditor [42] due to its heavy computational and manual overhead for large-scale repository corpus. In addition, we included six state-of-the-art LLMs as baselines, i.e., GPT-5-mini [29], DeepSeek-V3.2 [6], Gemini-2.5-flash [17], Grok-Code [41], Qwen3-Coder [30], and Llama3-70B-Instruct [27]. Concretely, for each code file containing errors in the client project, we followed the order described in Sec. 4.1.1 and used the prompt from [1] to directly instruct each LLM to generate a fixed version of the code file. We did not include program repair tools, as they rely on executing test cases to obtain failure traces and repair the identified bugs, which is different from our scenario.

Metric Selection. Following prior work [44], we define the *fixing success rate* at the error level (Fix_{er}) and the upgrade level (Fix_{up}) as follows.

$$Fix_{er} = \frac{\sum_{b\in\mathcal{B}}\left|\{\langle e,f\rangle \in b.pairs \mid e \text{ is fixed}\}\right|}{\sum_{b\in\mathcal{B}}\left|b.pairs\right|} \qquad Fix_{up} = \frac{\left|\{\, b \in \mathcal{B} \mid \forall \langle e,f\rangle \in b.pairs \text{ that } e \text{ is fixed}\}\right|}{\left|\mathcal{B}\right|}$$

Here, Fix_{er} denotes the proportion of individual errors in our benchmark that are successfully fixed, while Fix_{up} denotes the proportion of upgrade samples in which all errors are fixed. Specifically, for *API migration approaches* LibCatch and Librarian, an error is considered as fixed only if the original error is fixed and the fix introduces no new compilation error. For *API recommendation approaches* SemDiff, RepFinder, and RefactoringMiner, we regard an error as fixed if all API changes in the minimal fix are among the approach's recommendations, which assess the upper bound performance of the recommendation approaches.

5.2 Effectiveness Evaluation (RQ1)

We assessed Librarian on our benchmark, and compared its success fixing rates with the baseline approaches. As Table 3 shows, Librarian significantly outperforms all state-of-the-art on both the upgrade-level and error-level fixing success rates. Particularly, LibCatch only successfully fixes 57.9% of upgrade samples (161 / 278) and 59.7% of individual errors (1,715 / 2,871), while RepFinder fixes only 23.0% of upgrade samples (64 / 278) and 28.9% of errors (831 / 2,871). RefactoringMiner is slightly better than RepFinder by fixing 39.4% of errors (1,132 / 2,871), but still fixes just 27.3% of upgrade samples (76 / 278). SimDiff lags behind by fixing 3.6% of upgrade samples (10 / 278) and 5.7% of errors (164 / 2,871). In contrast, Librarian fixes 84.2% of all upgrade samples (234 / 278) and 83.9% of individual compilation errors (2,410 / 2,871), outperforming the best of the state-of-the-arts by 45.3% and 40.5% at the upgrade and error level, respectively. Table 4 presents the effectiveness of six baseline LLMs. Among them, GPT-5-mini achieves the best performance, fixing 56.5% of upgrade samples (157 / 278) and 57.2% of errors (1,641 / 2,871). In contrast, Llama3-70B fixes only 23.7% of upgrade samples (66 / 278) and 25.1% of errors (722 / 2,871). Librarian significantly outperforms the best baseline LLM by 49.0% and 46.9% at the upgrade and error level, respectively. This result demonstrates that LLM alone is insufficient in fixing incompatibility errors.

Table 5. Results of our ablation study

Ablated Module			Ablated LLM		
Ablated Version	Fix_{up}	Fix_{er}	Ablated Version	Fix_{up}	Fix_{er}
w/o Slicing	73.0% ↓11.2% (203 / 278)	65.1% ↓18.8% (1,870 / 2,871)	w/ DeepSeek-V3.2	80.2% ↓4.0% (223 / 278)	76.7% ↓7.2% (2,203 / 2,871)
w/o Refine	70.9% ↓13.3% (197 / 278)	72.8% ↓11.1% (2,089 / 2,871)	w/ Gemini-2.5-flash	75.2% ↓9.0% (209 / 278)	72.7% ↓11.2% (2,088 / 2,871)
w/o Usage. Change	68.7% ↓15.5% (191 / 278)	64.3% ↓19.6% (1,847 / 2,871)	w/ Grok-Code	78.8% ↓5.4% (219 / 278)	74.0% ↓9.9% (2,125 / 2,871)
w/o Sign. Change	81.7% ↓2.5% (227 / 278)	79.9% ↓4.0% (2,294 / 2,871)	w/ Qwen3-Coder	80.9% ↓3.3% (225 / 278)	77.2% ↓6.7% (2,217 / 2,871)
w/o JavaDoc	83.1% ↓1.1% (231 / 278)	80.0% ↓3.9% (2,297 / 2,871)	w/ Llama3-70B	70.1% ↓14.1% (195 / 278)	58.2% ↓25.7% (1,672 / 2,871)
w/o Fix Hint	66.9% ↓17.3% (186 / 278)	56.7% ↓27.2% (1,628 / 2,871)			

Case Studies. We conducted case studies to further demonstrate the effectiveness of LIBRARIAN. For the error shown in Fig. 1b, LIBRARIAN uses the broken API usage context change extracted from the library, as shown in Fig. 1d, to generate a fix that is identical to the developer's fix in Fig. 1b. Besides this, we present two additional case studies.

In case study 1, after upgrading the library `weixin-java-miniapp`, a client project encounters an error due to the removal of the class `WxMaTemplateMessage.Data` (at Line 9 in Fig. 4a). Fig. 4b shows the corresponding broken API usage context change extracted by LIBRARIAN, which indicates that the class `WxMaTemplateMessage.Data` should be replaced with `WxMaTemplateData`. Based on this extracted fix hint, LIBRARIAN successfully generates the fix, shown in Fig. 4a, which is consistent with the developer's fix.

```
1  import cn.binarywang.wx.miniapp.bean.WxMaTemplateData;
2
3  public class WxMaConfiguration {
4      private final WxMaMessageHandler templateMsgHandler = (…, service, …) ->
5              service
6              .getMsgService()
7              .sendTemplateMsg(WxMaTemplateMessage.builder()
8                      .templateId("template id").formId("form id")
9  -                   .data(Lists.newArrayList(new WxMaTemplateMessage.Data(...)))
10 +                   .data(Lists.newArrayList(new WxMaTemplateData(...)))
11                     .toUser(wxMessage.getFromUser()).build());
12 }
```

(a) The generated fix in case 1

```
1 +import cn.binarywang.wx.miniapp.bean.WxMaTemplateData;
2
3  @Test
4  public class WxMaMsgServiceImplTest {
5      public void testSendTemplateMsg() throws WxErrorException {
6          WxMaTemplateMessage templateMessage = WxMaTemplateMessage.builder()
7                  .toUser(config.getOpenid()).formId("FORMID").page("index")
8                  .data(Lists.newArrayList(
9  -                       new WxMaTemplateMessage.Data("keyword1", ...),
10 +                      new WxMaTemplateData("keyword1", ...),
11 -                     new WxMaTemplateMessage.Data("keyword2", ...),
12 +                     new WxMaTemplateData("keyword2", ...),
13 -                     new WxMaTemplateMessage.Data("keyword3", ...),
14 +                     new WxMaTemplateData("keyword3", ...),
15 -                     new WxMaTemplateMessage.Data("keyword4", ...)
16 +                     new WxMaTemplateData("keyword4", ...))
17                 )
18                 .templateId(config.getTemplateId()).emphasisKeyword("keyword1.DATA").build();
19     }
20 }
```

(b) The extracted broken API usage context change in case 1

```
1  public class JtsBinaryCodec {
2      public Shape readJtsGeom(DataInput dataInput) {
3          InStream inStream = new InStream() {
4              @Override
5  -           public void read(byte[] buf) throws IOException {
6  +           public int read(byte[] buf) throws IOException {
7                  if (first) {
8                      if (buf.length != 1) throw new IllegalStateException();
9                      buf[0] = WKBConstants.wkbXDR;
10                     first = false;
11 +                    return 1;
12                 } else {
13                     dataInput.readFully(buf);
14 +                   return buf.length;
15                 }
16             }
17         };
18         /* Other Statements */
19     }
20 }
```

(c) The generated fix in case 2

Fig. 4. Results of two case studies

In case study 2, after upgrading the library `jts-core`, the return type of the overridden method `read(byte[] buf)` changes from `void` to `int` (at Line 5 in Fig. 4c). Existing state-of-the-art tools are unable to handle this type of breaking change. In contrast, LIBRARIAN uses an LLM to interpret the compilation error message, successfully updates the return type of the overridden method (at Line 6 in Fig. 4c), and further inserts an appropriate `return` statement at the end of the method body by understanding the code's semantics (at Lines 11 and 14 in Fig. 4c). The generated fix is semantically equivalent to the developer's fix, illustrating the advantage of using LLMs for API migration when dealing with semantic changes.

In-Depth Analysis. Failed fixes of LIBRARIAN typically arise from two aspects. The first is *hallucinated fixes*. Even with good fix hints, the LLM sometimes hallucinates over irrelevant statements, introducing silent semantic bugs that raise from JDT. The second is *complex semantic changes*. Some library upgrades do not just rename or remove APIs, they change behaviors which require the client

project to add complex pre- or post-processing logic. Together, these gaps in LLM hallucination, and complex changes explain why a minority of cases still lead to incorrect fixes by Librarian.

Context Length Analysis. We explored the impact of prompt context length on Librarian's effectiveness. Specifically, we employed the `cl100k_base` tokenizer [23] to calculate the average token count for the prompts used in fixing each library upgrade sample. Fig. 5 presents Librarian's upgrade-level fixing success rate across different prompt context length, where the histogram reports the distribution of library upgrade samples within each prompt context length range, while the line plot depicts the average fixing success rate achieved by Librarian for each corresponding prompt context length range. We observed that there was no significant correlation between the fixing success rate and prompt context length. This is because the average prompt token length in Librarian is 1.1K, which is still far from reaching the maximum limit of LLM context window.

Cost Measurement. The total cost for fixing 278 upgrade samples in our benchmark was 4.30$, averaging only 0.015$ per sample. Specifically, our slicing strategy reduced the average token count per prompt from 2.5K to 1.1K, a 56.0% reduction, significantly lowering the cost of fixing.

Summary. On 278 upgrade samples (2,871 errors), Librarian fixes 84.2% of all upgrades and 83.9% of all errors, substantially outperforming the best state-of-the-art by 45.3% and 40.5% at the upgrade and error level, respectively. It also surpasses the best baseline LLM by 49.0% at upgrade level and 46.9% at error level. The cost of fixing each upgrade sample is only 0.015$.

5.3 Ablation Study (RQ2)

We conducted ablation study on the modules in Librarian and the inference LLM used. For the ablated module, we created six ablated versions of Librarian. First, we removed the slicing module from Librarian (w/o Slicing). Second, we removed the refinement module from Librarian (w/o Refine). Next, we removed the broken API usage context change (i.e., $Diff_{ctx}$), broken API signature change (i.e., $Diff_{sig}$), and broken API JavaDoc (i.e., Dep_{doc}) from the fix hints, respectively (w/o Usage. Change, w/o Sign. Change, and w/o JavaDoc, respectively). Finally, we removed all fix hints (w/o Fix Hint). For the ablated LLM, we replaced GPT-5-mini in Librarian with other baseline LLMs.

Table 5 reports the results of our ablation study. In the ablation of each module in Librarian, removing the slicing module ("w/o Slicing") leads to a 11.2% drop in Fix_{up} and a 18.8% drop in Fix_{er}, indicating that reducing irrelevant code in the prompt helps LLM focus on the broken code and improves effectiveness. Removing the fix refinement module ("w/o Refine") reduces Fix_{up} by 13.3% and Fix_{er} by 11.1%, demonstrating that iterative LLM refinement is crucial for resolving complex or subtle compilation errors. Removing the broken API usage context change, broken API signature change, and broken API JavaDoc from the fix hints result in decreases of 15.5%, 2.5%, and 1.1% in Fix_{up}, and 19.6%, 4.0%, and 3.9% in Fix_{er}, respectively, indicating that all three types of fix hints contribute to improving effectiveness, with the broken API usage context change being the most significant. Removing all fix hints ("w/o Fix Hint") results in the largest decrease of 17.3% in Fix_{up} and 27.2% in Fix_{er}, highlighting the critical role of library-side fix hints in generating accurate fixes.

In the ablation of the inference LLM, replacing GPT-5-mini with Qwen3-Coder results in the smallest decline of 3.3% in Fix_{up} and 6.7% in Fix_{er} among the five ablated versions. Replacing it with Llama3-70B leads to the largest drop of 14.1% in Fix_{up} and 25.7% in Fix_{er}. This is mainly due to differences in the models' capabilities. Moreover, when comparing these results with those in Table 4, the addition of the modules we design to equip the baseline LLMs results in significant improvements, emphasizing the importance of each module in enhancing the effectiveness of Librarian.

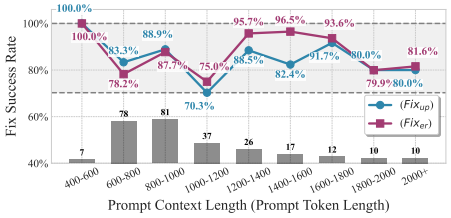
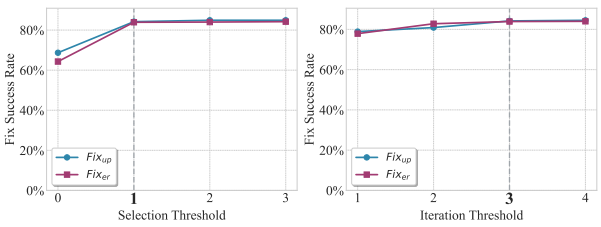


Fig. 5. Results of fixing success rate across different prompt context length



(a) Sensitivity of τ_{top} (b) Sensitivity of τ_{max}

Fig. 6. Results of our parameter sensitivity analysis

Table 6. Results of our efficiency evaluation (in seconds)

API Recommendation Approach			API Migration Approach				
				Librarian			
SemDiff	RepFinder	RefactoringMiner	LibCatch	Broken API Usage Extraction	Fix Hint Extraction	Broken API Usage Fixing	Total
1.4 s	0.5 s	152.5 s	496.4 s	225.6 s	274.4 s	58.5 s	558.5 s

Table 7. Results of our human study (in seconds)

RefactoringMiner	LibCatch	GPT-5-mini	Librarian
194.9 s	102.5 s	85.3 s	33.2 s

Summary. Removing slicing, fix refinement, and library-side fix hints drops upgrade-level fixing success rate by 1.1% to 17.3% and error-level fixing success rate by 3.9% to 27.2%, confirming every module's importance. Replacing GPT-5-mini with other baseline LLMs reduces fixing success rate by 3.3% to 14.1% at upgrade level and 6.7% to 25.7% at error level.

5.4 Parameter Sensitivity Analysis (RQ3)

Two key parameters are configurable in Librarian, namely τ_{top} (Sec. 4.2.3) and τ_{max} (Sec. 4.3.3). The default values for these parameters are set to 1 and 3, respectively. To assess the sensitivity of these parameters on Librarian's fixing success rates, we varied one parameter while keeping the other unchanged. Fig. 6 presents the results of our parameter sensitivity analysis. Librarian achieves the best effectiveness when τ_{top} is set to 1. Librarian demonstrates a significant improvement when τ_{max} is set to 3, with only a slight increase at 4. Consequently, the maximum refinement number is set to 3 to reduce computational cost of LLM. Finally, we set τ_{top} to 1 and τ_{max} to 3.

Summary. Performance peaks at top 1 hints ($\tau_{top} = 1$) and 3 refinements ($\tau_{max} = 3$), balancing fixing effectiveness with LLM cost.

5.5 Efficiency Evaluation (RQ4)

To evaluate the efficiency of Librarian, we measured the average execution time of each approach to fix errors in one upgrade. Table 6 presents the results of our efficiency evaluation. On average, Librarian takes 558.5 seconds per upgrade, most of which is spent in extracting fix hints from the library. This overhead can be further reduced by caching the fix hints for each broken API to avoid repeated extraction. Those API recommendation approaches are fast because they only suggest replacement APIs without code-level adaptation, but Librarian achieves fully automated, end-to-end fix generation that can drastically reduces manual effort. Although Librarian's execution time is still 12.5% higher than LibCatch, Librarian's much higher effectiveness makes it acceptable.

Human Study. To further evaluate Librarian's effectiveness in reducing manual effort, we conducted a human study with four developers, each with at least two years of Java development experience. We assigned each participant four tasks. Each task consisted of 10 distinct upgrade samples randomly selected from our benchmark, along with the output produced by one of four approaches, i.e., the best API recommendation approach RefactoringMiner, the API migration approach LibCatch, the best baseline LLM GPT-5-mini, and Librarian. To minimize confounding factors from uneven task difficulty, we ensured that the number of incompatibility errors across selected upgrade samples differed by no more than two. Participants were asked to resolve the incompatibility errors in the assigned upgrade samples using the provided outputs. For each task, we measured the average time required to fix an upgrade sample. Table 7 reports the results of this human study. Compared with the other three baseline approaches, Librarian reduces the manual fixing time by 83.0%, 67.6%, and 61.1%, respectively, thereby substantially lowering developers' manual effort.

Summary. Although Librarian's runtime overhead (558.5 seconds) exceeds lightweight recommenders, its fully automated fixing can significantly reduce manual effort.

5.6 Generality Evaluation (RQ5)

Large language models can inadvertently memorize training data, leading to artificially high performance [39]. To assess the generality of Librarian, we followed the procedure outlined in Sec. 3 to construct a new benchmark based on library upgrades that postdate the LLM's training cutoff. In total, this resulted in 20 previously unseen upgrade samples for evaluation, containing 47 errors.

Librarian successfully fixed all errors in 15 of the 20 upgrade samples, and corrected 37 of the 47 errors in total. These fixing success rates at both upgrade and error levels were similar to those observed in the effectiveness evaluation of Librarian (see Sec. 5.2), suggesting that data leakage from the LLM has little impact on Librarian's effectiveness, and Librarian is generalizable. In addition, as shown in Table 4, the fixing success rate drops substantially when directly using the baseline LLMs, which further indicates that Librarian's effectiveness does not stem from data leakage but from its carefully designed methodology.

Summary. On 20 new upgrade samples with 47 errors, Librarian fixes all errors in 75.0% of upgrade samples and 78.7% of errors, matching performance on the original benchmark and indicating minimal data-leakage and good generality.

5.7 Usefulness Evaluation (RQ6)

To evaluate the practical usefulness of Librarian in real world, we extracted the repositories of projects that had commit activity after June 1, 2025. We then cloned these projects locally, and upgraded their dependent libraries one by one. Whenever an upgrade caused a compilation error, we used Librarian to automatically fix it. Once the patched project successfully passed CI/CD, we submitted a pull request for the developers to review and merge our fix.

We submitted 57 pull requests to the developers of 57 projects, of which 32 were merged. Of the remaining pull requests, 2 are planned to be included in the next major version of the project, and have not yet been merged. The rest are still awaiting a response from the project developers. These results demonstrate the practical usefulness of Librarian in real-world library upgrade scenarios.

Summary. Of 57 real-world pull requests, 32 were merged immediately and 2 slated for upcoming releases, demonstrating Librarian's practical value in real-word projects.

6 Discussion

We first discuss the limitations of Librarian, and then elaborates the threats to validity.

6.1 Limitations

Shadow API Mismatches. Librarian utilizes the state-of-the-art refactoring detection approach RefactoringMiner to map shadow APIs from the original version to the upgraded version in order to extract the usage context changes in the library. However, during the library upgrade process, there are still some shadow APIs that undergo substantial refactoring, leaving their counterparts in the upgraded version mismatched. Despite this limitation, our effectiveness evaluation demonstrates that the current mapping approach remains highly effective. In the future, we plan to explore more precise API mapping techniques to further enhance the accuracy.

Run-Time Errors. Librarian is currently designed to fix compilation errors caused by broken APIs. It does not demonstrate the effectiveness of Librarian for runtime incompatibilities (e.g., behavioral changes that only surface during execution or within test cases), which account for under 2.3% according to [24]. We will explore runtime incompatibility errors in future.

Language Scope. Our implementation and evaluation are currently focused exclusively on Java. Although the core methodology, i.e., combining client-side context with library-side hints is language insensitive, we will adapt Librarian to different languages in our future work.

6.2 Threats to Validity

Effectiveness Comparison. In our evaluation of API recommendation baselines, we limited candidates to APIs within the upgraded library. In practice, developers may import alternative APIs from other libraries, which could narrow the observed performance gap.

Exclusion of Meditor. We did not reproduce Meditor at scale due to its high manual and computational overhead. Prior work [44] reported its fix rate remained below 1%, suggesting its omission has minimal impact on our effectiveness evaluation.

Benchmark Bias. Our benchmark may suffer from potential bias (e.g., it may favor libraries that have available annotations and test cases). To mitigate this threat, we construct our benchmark differently from prior work. Instead of first selecting well-maintained libraries and then searching for client projects that depend on them, we begin with a large corpus of real-world client projects and mine upgrade samples from these projects. This process helps reduce any preference for specific libraries. Moreover, our benchmark contains 278 upgrade samples spanning 166 libraries, which is substantially larger than the datasets used in prior work. In addition, our benchmark covers a wider variety of breaking change types, further mitigating potential bias in the evaluation.

7 Related Work

We review and discuss related work in two areas, i.e., library vulnerability risks, and library upgrade.

7.1 Library Vulnerability Risks

Reusing third-party libraries can significantly accelerate software development. However, it also introduces substantial security risks within ecosystems. Several works [21, 26, 43] have highlighted the extensive use and invocation of vulnerable libraries and functions, increasing the potential vulnerability exploitation. Similarly, Hu et al. [21] and Liu et al. [26] find that vulnerabilities can be propagated within libraries and across their transitive dependencies in the Go and NPM ecosystems, respectively. To better understand the impact of these vulnerabilities on downstream projects, Wang et al. [37] and Wu et al. [40] apply call graph analysis to assess function-level vulnerability reachability analysis, and report that 13.9% of vulnerable functions are reachable by downstream projects, indicating a significant potential for exploitation in client projects. Therefore, it is essential to mitigate the security risk propagation in software supply chain via automatic library upgrade.

7.2 Library Upgrade

Keeping libraries always up to date is essential for mitigating the risk of vulnerabilities in dependent libraries [5, 20]. However, Mirhosseini et al. [28] reveal that only 32% of suggested upgrades are accepted in practice, due to the fact that dependency upgrades usually introduce incompatibility errors [25]. To timely fix these errors after upgrades, there are currently two main approaches, i.e., API migration approaches [42, 44] and API recommendation approaches [4, 22, 31, 34, 38].

API migration approaches fix incompatibility errors either by using historical migration examples or by applying predefined migration patterns. Meditor [42] extracts the API migration changes from the existing migration examples from previously upgraded client projects. Based on the changes, Meditor designs heuristic patterns to migrate those changes to client projects. However, migration examples for specific broken APIs are often unavailable, and collecting these examples from a large number of projects is computationally infeasible. LibCatch [44] leverages compilation error message in client projects to guide the design of migration patterns. For each compilation error, it first identifies its type and then iteratively applies pre-defined migration patterns on error-raising statements until the error is eliminated. Both of them rely on predefined migration patterns, which can lead to incorrect fix and broken functionality. In contrast, Librarian extracts fix hints from library itself and leverages LLM to generate the fix, without depending on predefined fixing patterns.

API recommendation approaches do not directly modify client code but instead recommend alternative APIs to replace the missing or broken ones. Aura [38], RefDiff[31], and RefactoringMiner[34] use predefined rules to detect changes in API declarations between library versions and recommend corresponding alternatives in the upgraded version. SemDiff [4] identifies an alternative API in the upgraded library version by analyzing call differences. RepFinder [22] expands this scope by also incorporating JavaDoc comments. Nonetheless, as mentioned in RepFinder [22], not all incompatibility errors can be fixed simply by replacing a broken API with an recommended alternative. For example, some incompatibility errors are fixed by modifying the context of the input arguments used with the broken API. Moreover, even when alternative APIs are correctly recommended, additional manual effort is needed for further adaptation at the client code level.

8 Conclusions

We present Librarian, an LLM-driven approach that automatically fixes incompatibility errors from library upgrades by extracting fix hints from the library itself and leveraging inter-procedural dataflow slices, then iteratively refining fixes via compilation feedback. Our evaluation has demonstrates that Librarian significantly outperforms state-of-the-arts.

9 Data Availability

The benchmark and source code are available at our website [33].

Acknowledgment

This work was supported by the National Natural Science Foundation of China (Grant No. 62332005 and 62372114).

References

[1] Aylton Almeida, Laerte Xavier, and Marco Tulio Valente. 2024. Automatic library migration using large language models: First results. In *Proceedings of the 18th ACM International Symposium on Empirical Software Engineering and Measurement*. 427–433. doi:10.1145/3674805.3690746

[2] Aline Brito, Marco Tulio Valente, Laerte Xavier, and Andre Hora. 2020. You broke my code: understanding the motivations for breaking changes in APIs. *Empirical Software Engineering* 25 (2020), 1458–1492. doi:10.1007/s10664-019-09756-z

[3] Muslim Chochlov, Gul Aftab Ahmed, James Vincent Patten, Guoxian Lu, Wei Hou, David Gregg, and Jim Buckley. 2022. Using a nearest-neighbour, BERT-based approach for scalable clone detection. In *Proceedings of the 38th IEEE International Conference on Software Maintenance and Evolution*. 582–591. doi:10.1109/icsme55016.2022.00080
[4] Barthélémy Dagenais and Martin P Robillard. 2011. Recommending adaptive changes for framework evolution. *ACM Transactions on Software Engineering and Methodology* 20, 4 (2011), 1–35. doi:10.1145/2000799.2000805
[5] Andreas Dann, Ben Hermann, and Eric Bodden. 2023. UPCY: Safely updating outdated dependencies. In *Proceedings of the 45th IEEE International Conference on Software Engineering*. 233–244. doi:10.1109/icse48619.2023.00031
[6] DeepSeek. 2025. *DeepSeek*. Retrieved March 26, 2025 from https://www.deepseek.com
[7] Jens Dietrich, David Pearce, Jacob Stringer, Amjed Tahir, and Kelly Blincoe. 2019. Dependency versioning in the wild. In *Proceedings of the 16th ACM International Conference on Mining Software Repositories*. 349–359. doi:10.1109/msr.2019.00061
[8] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating large language models in class-level code generation. In *Proceedings of the 46th IEEE International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3639219
[9] Eclipse. 2025. *Eclipse JDT (Java development tools)*. Retrieved March 27, 2025 from https://projects.eclipse.org/projects/eclipse.jdt
[10] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM international conference on Automated software engineering*. 313–324. doi:10.1145/2642937.2642982
[11] Minying Fang, Xing Yuan, Yuying Li, Haojie Li, Chunrong Fang, and Junwei Du. 2025. Enhanced Prompting Framework for Code Summarization with Large Language Models. *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2025), 1630–1653. doi:10.1145/3728949
[12] Darius Foo, Hendy Chua, Jason Yeo, Ming Yi Ang, and Asankhaya Sharma. 2018. Efficient static checking of library updates. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 791–796. doi:10.1145/3236024.3275535
[13] Lukas Fruntke and Jens Krinke. 2025. Automatically fixing dependency breaking changes. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 2146–2168. doi:10.1145/3729366
[14] GitHub. 2025. *GumTree - An awesome code differencing tool*. Retrieved May 29, 2025 from https://github.com/GumTreeDiff/gumtree
[15] GitHub. 2025. *RefactoringMiner*. Retrieved May 29, 2025 from https://github.com/tsantalis/RefactoringMiner
[16] GitHub. 2025. *Soot - A Java optimization framework*. Retrieved May 29, 2025 from https://github.com/soot-oss/soot
[17] Google. 2025. *Google Gemini*. Retrieved March 27, 2025 from https://gemini.google.com/
[18] Priyanshu Gupta, Avishree Khare, Yasharth Bajpai, Saikat Chakraborty, Sumit Gulwani, Aditya Kanade, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. 2023. Grace: Language models meet code edits. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1483–1495. doi:10.1145/3611643.3616253
[19] Hao He, Bogdan Vasilescu, and Christian Kästner. 2025. Pinning Is Futile: You Need More Than Local Dependency Versioning to Defend against Supply Chain Attacks. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 266–289. doi:10.1145/3715728
[20] Joseph Hejderup and Georgios Gousios. 2022. Can we trust tests to automate dependency updates? a case study of java projects. *Journal of Systems and Software* 183 (2022), 111097. doi:10.1016/j.jss.2021.111097
[21] Jinchang Hu, Lyuye Zhang, Chengwei Liu, Sen Yang, Song Huang, and Yang Liu. 2024. Empirical Analysis of Vulnerabilities Life Cycle in Golang Ecosystem. In *Proceedings of the 46th IEEE International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3639230
[22] Kaifeng Huang, Bihuan Chen, Linghao Pan, Shuai Wu, and Xin Peng. 2021. REPFINDER: Finding replacements for missing APIs in library update. In *Proceedings of the 36th ACM International Conference on Automated Software Engineering*. 266–278. doi:10.1109/ase51524.2021.9678905
[23] Shantanu Jain. 2025. *tiktoken: A fast BPE tokeniser for use with OpenAI's models*. Retrieved August 13, 2026 from https://github.com/openai/tiktoken
[24] Dhanushka Jayasuriya, Valerio Terragni, Jens Dietrich, and Kelly Blincoe. 2024. Understanding the impact of APIs behavioral breaking changes on client applications. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1238–1261. doi:10.1145/3643782
[25] Dhanushka Jayasuriya, Valerio Terragni, Jens Dietrich, Samuel Ou, and Kelly Blincoe. 2023. Understanding breaking changes in the wild. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1433–1444. doi:10.1145/3597926.3598147
[26] Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. 2022. Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. In *Proceedings of the 44th IEEE International*

Conference on Software Engineering. 672–684. doi:10.1145/3510003.3510142
[27] Meta. 2025. *Llama 3*. Retrieved March 27, 2025 from https://www.llama.com/models/llama-3/
[28] Samim Mirhosseini and Chris Parnin. 2017. Can automated pull requests encourage software developers to upgrade out-of-date dependencies?. In *Proceedings of the 32nd ACM international Conference on Automated Software Engineering*. 84–94. doi:10.1109/ase.2017.8115621
[29] OpenAI. 2025. *ChatGPT*. Retrieved March 27, 2025 from https://chatgpt.com/
[30] Qwen. 2025. *Qwen Chat*. Retrieved March 27, 2025 from https://chat.qwen.ai/
[31] Danilo Silva, Joao Paulo da Silva, Gustavo Santos, Ricardo Terra, and Marco Tulio Valente. 2020. Refdiff 2.0: A multi-language refactoring detection tool. *IEEE Transactions on Software Engineering* 47, 12 (2020), 2786–2802. doi:10.1109/tse.2020.2968072
[32] Mifta Sintaha, Noor Nashid, and Ali Mesbah. 2023. Katana: Dual slicing based context for learning bug fixes. *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 1–27. doi:10.1145/3579640
[33] Librarian. 2025. *Librarian*. Retrieved May 25, 2025 from https://github.com/jxfzzzt/ISSTA2026-Librarian
[34] Nikolaos Tsantalis, Matin Mansouri, Laleh M Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. (2018), 483–494. doi:10.1145/3180155.3180206
[35] Daniel Venturini, Filipe Roseiro Cogo, Ivanilton Polato, Marco A Gerosa, and Igor Scaliante Wiese. 2023. I depended on you and you broke me: An empirical study of manifesting breaking changes in client packages. *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 1–26. doi:10.1145/3576037
[36] Chaozheng Wang, Shuzheng Gao, Cuiyun Gao, Wenxuan Wang, Chun Yong Chong, Shan Gao, and Michael R Lyu. 2024. A Systematic Evaluation of Large Code Models in API Suggestion: When, Which, and How. In *Proceedings of the 39th ACM International Conference on Automated Software Engineering*. 281–293. doi:10.1145/3691620.3695004
[37] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. 2020. An empirical study of usages, updates and risks of third-party libraries in java projects. In *Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution*. 35–45. doi:10.1109/icsme46990.2020.00014
[38] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. 2010. Aura: a hybrid approach to identify framework evolution. In *Proceedings of the 32nd IEEE International Conference on Software Engineering*. 325–334. doi:10.1145/1806799.1806848
[39] Yonghao Wu, Zheng Li, Jie M Zhang, and Yong Liu. 2024. Condefects: A complementary dataset to address the data leakage concern for llm-based fault localization and program repair. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 642–646. doi:10.1145/3663529.3663815
[40] Yulun Wu, Zeliang Yu, Ming Wen, Qiang Li, Deqing Zou, and Hai Jin. 2023. Understanding the threats of upstream vulnerabilities to downstream projects in the maven ecosystem. In *Proceedings of the 45th IEEE International Conference on Software Engineering*. 1046–1058. doi:10.1109/icse48619.2023.00095
[41] xAI. 2025. *Grok*. Retrieved March 27, 2025 from https://grok.com/
[42] Shengzhe Xu, Ziqi Dong, and Na Meng. 2019. Meditor: inference and application of API migration edits. In *Proceedings of the 27th IEEE International Conference on Program Comprehension*. 335–346. doi:10.1109/icpc.2019.00052
[43] Xian Zhan, Lingling Fan, Sen Chen, Feng We, Tianming Liu, Xiapu Luo, and Yang Liu. 2021. Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications. In *Proceedings of the 43rd IEEE International Conference on Software Engineering*. 1695–1707. doi:10.1109/icse43902.2021.00150
[44] Hao Zhong and Na Meng. 2024. Compiler-directed Migrating API Callsite of Client Code. In *Proceedings of the 46th IEEE International Conference on Software Engineering*. 1–12. doi:10.1145/3597503.3639084

Received 2026-01-29; accepted 2026-04-16