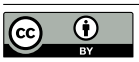
Break to Adapt: Knowledge-Based Updates of Breaking Dependencies in JavaScript

YIFAN XIA*, Zhejiang University, China
CHENGWEI LIU*, Nankai University, China
ZIFAN XIE, Chongqing University, China
LYUYE ZHANG, Nanyang Technological University, Singapore
PEIYU LIU, Zhejiang University, China
KANGJIE LU, University of Minnesota, USA
YANG LIU, Nanyang Technological University, Singapore
WENHAI WANG†, Zhejiang University, China
SHOULING JI†, Zhejiang University, China

Modern software libraries continually evolve to maintain activeness, improve performance, and enhance security. This evolution frequently introduces major version updates that include incompatible modifications (*breaking changes*), which often cause downstream projects to fail and significantly increase the effort required for maintenance. To reduce this burden on developers, recent studies propose automated approaches that leverage API characterization techniques and compilation error reports to guide LLM-based updates. However, these methods rely on static typing and compilation feedback, which are not available in dynamic languages such as JavaScript. As a result, when applied to JavaScript, they degrade to a knowledge-free setting and cannot effectively support automated dependency updates. In this paper, we propose a knowledge-based approach that mines explicit update knowledge, referred to as *breaking change records*, from library repositories to support automated dependency updates in the presence of breaking changes (i.e., breaking dependency updates). We first conduct an empirical investigation of how breaking change records are maintained in top-ranked JavaScript libraries. Our study reveals that 83.8% of libraries maintain such records within their repositories, often spread across multiple locations. However, the quality of these records varies considerably, with issues such as implicit references to changed objects and vague adaptation instructions. To address this issue, we systematically characterize breaking change records by identifying recurring patterns in both their locations and the types of information they convey. Building on these insights, we develop BDUpdater, an agent-based framework that aggregates repository sources and refines raw records into structured breaking change lists, thereby mitigating the absence of API differencing tools. Leveraging this mined knowledge, BDUpdater pinpoints changed objects and statically identifies the affected client code, while supplying fine-grained change information to LLMs to guide accurate client code migration in the absence of compilation error messages. Our evaluation on 13 popular JavaScript libraries and 84 clients shows that BDUpdater recovers 90.5% of the breaking dependency updates with high semantic equivalence to developer-authored adaptations, while incurring an average cost of only $0.05 per client.

*Equal Contribution, † Corresponding authors.

Authors' addresses: Yifan Xia, Zhejiang University, Hangzhou, China, yfxia@zju.edu.cn; Chengwei Liu, Nankai University, Tianjin, China, chengwei.liu@nankai.edu.cn; Zifan Xie, Chongqing University, Chongqing, China, xzff@cqu.edu.cn; Lyuye Zhang, Nanyang Technological University, Singapore, zh0004ye@e.ntu.edu.sg; Peiyu Liu, Zhejiang University, Hangzhou, China, liupeiyu@zju.edu.cn; Kangjie Lu, University of Minnesota, Minneapolis, USA, kjlu@umn.edu; Yang Liu, Nanyang Technological University, Singapore, yangliu@ntu.edu.sg; Wenhai Wang, Zhejiang University, Hangzhou, China, zdzzlab@zju.edu.cn; Shouling Ji, Zhejiang University, Hangzhou, China, sji@zju.edu.cn.



XXXX-XXXX/2026/7-ARTFSE109
https://doi.org/10.1145/3808116

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

ACM Reference Format:
Yifan Xia, Chengwei Liu, Zifan Xie, Lyuye Zhang, Peiyu Liu, Kangjie Lu, Yang Liu, Wenhai Wang, and Shouling Ji. 2026. Break to Adapt: Knowledge-Based Updates of Breaking Dependencies in JavaScript. 3, FSE, Article FSE109 (July 2026), 23 pages. https://doi.org/10.1145/3808116

1 INTRODUCTION

Modern software ecosystems evolve rapidly, with libraries frequently updated to deliver new features and security fixes [1, 2]. While staying current is essential for software quality and security, dependency updates are often blocked by breaking changes, which are incompatible modifications to public interfaces [3]. These changes can cause version bumps to fail, requiring costly and error-prone refactoring [4], and leaving up to 82% of projects lagging behind on dependency updates [5].

In light of this issue, the automation of dependency updates has emerged as an active area of research [6, 7]. Tools such as Dependabot[8] and Renovate [9] perform well for routine updates but are ineffective when updates involve breaking changes. This limitation arises because resolving such changes requires not only identifying the differences in the updated library but also modifying the dependent project's code appropriately. Some academic prototypes attempt to address this issue by analyzing the API changes between library versions [10–13]. However, these approaches rely on manually defined patterns for structural modifications and have been developed primarily for statically typed languages. As a result, current automated approaches for handling JavaScript breaking dependency updates still require substantial manual effort [14].

Recent advances in Large Language Models (LLMs) have opened new avenues for addressing software engineering challenges. Preliminary studies have investigated the application of LLMs to automate dependency updates in Java projects [15, 16]. However, their effectiveness in handling dynamic languages remains limited. This limitation is primarily due to two critical challenges:

Issue 1: Incapability of Dynamic API Change Characterization. Existing approaches [15, 16] rely on automated tools such as APIDiff [10] to generate structured representations of API changes across library versions. As highlighted by Lukas et al., such structured information is a critical input for enabling LLM-based analysis and adaptation [15]. These techniques work for statically typed languages like Java, where each class member has a fully qualified name, making it feasible to compute type-level differences between API versions. In contrast, JavaScript libraries expose their public interfaces dynamically and lack access modifiers, which makes static API signature differencing infeasible. As a result, when applied to JavaScript dependency updates, existing LLM-based techniques are forced to operate without explicit knowledge of the actual API changes, thereby fundamentally limiting their effectiveness.

Issue 2: Mitigation Reliance on Post-conditions. Prior LLM-based work often adopts a post-mitigation workflow, where the client project is first built, and then fixed iteratively using compiler error messages [15, 16]. This strategy is effective for statically typed languages, where compilation provides precise diagnostics, but it is less applicable to dynamic languages. Since JavaScript lacks a compilation phase, developers cannot rely on compiler feedback to localize breaking changes [17]. In addition, many breaking changes are behavioral rather than syntactic, producing no explicit errors but altering runtime functionality (e.g., asynchronous rewrites [18]). As a result, post-failure mitigation provides limited and unreliable guidance for addressing JavaScript breaking changes.

In this paper, we propose BDUpdater, a system designed to automate JavaScript breaking dependency updates by leveraging the change documentation. Given an upstream library and its downstream client, BDUpdater first uses a *record mining* agent to extract and formalize the change information directly from the library's repository. It then performs a static search on downstream clients to localize potentially affected code snippets. Finally, the *client adaptation* agent generates

targeted code modifications, which can either be integrated into an automated update workflow or serve as reference for human. BDUpdater is specifically designed to address the two challenges:

Addressing Issue 1: Existing automated techniques struggle in the absence of explicit API change knowledge, as they rely on structured information to locate and adapt affected code. A review of empirical studies on API evolution shows that prior work typically depends on manually extracting breaking changes from library repositories [19], which motivates us to automate this process. To this end, we conduct an empirical study to investigate the *availability* and *characteristics* of breaking change records in popular JavaScript libraries. Our results indicate that 83.8% of libraries document breaking changes across multiple repository locations; however, the quality of these records varies considerably, often suffering from issues such as implicit object references or vague adaptation instructions. From these findings, we identify a set of crucial recurring fields that make breaking change records actionable. Guided by these insights, BDUpdater employs an agent-based process that aggregates breaking change records from heterogeneous sources, incrementally refines them until all crucial fields are captured. This approach enables the generation of descriptive breaking change knowledge without API differencing tools.

Addressing Issue 2: Because JavaScript lacks a compilation phase and provides limited runtime feedback, post-condition based approaches are ineffective. To address this limitation, BDUpdater introduces a hybrid *pre-mitigation* workflow. In this workflow, the structured breaking change records produced by our system guide a repository-level static search in downstream projects to proactively identify potentially affected code locations, without relying on build failures. For each candidate location, LLMs are then used to filter false positives and synthesize precise code modifications based on the records collected during the change identification phase.

Our experimental evaluation shows that BDUpdater effectively automates breaking dependency updates in JavaScript. Across 13 libraries and 84 client–library pairs, BDUpdater achieves a 90.5% build success rate, with over 95% of generated adaptations being semantically equivalent to developer-authored updates. Compared with zero-shot LLM baselines, the success rates improve from 26.9% in the knowledge-free setting and 61.5% with coarse-grained records to 92.3% when equipped with fine-grained breaking change records. Furthermore, efficiency analysis indicates that both record mining and client adaptation incur modest costs (under 100 seconds and $0.05 per case on average), making the approach practical for large-scale deployment. Overall, these results demonstrate that BDUpdater provides accurate and cost-effective support for reducing developer effort in managing breaking dependency updates. This paper makes the following contributions:

- We conduct an empirical study of top-ranked JavaScript libraries to investigate how breaking change records are maintained. Our analysis identifies recurring patterns in both their locations and informational content, offering insights for automated dependency management.
- We design BDUpdater, a hybrid agent-based framework that addresses the incapability of dynamic API change characterization and automates breaking dependency updates in JavaScript.
- We evaluate BDUpdater on real-world library–client pairs involving actual breaking changes, including both historical and newly released versions, serving as a resource for future research. The implementation and experimental data of this paper are available at [19].

2 PRELIMINARIES

2.1 Definitions

To ensure clarity throughout this work, we define the key terminology used:

Client/Library: This describes a relationship between two JavaScript packages. The **client** is a repository or a package that integrates the **library** package as one of its dependencies. When clients

upgrade to a newer library version with breaking changes (due to syntax or semantic alterations), downstream clients may encounter runtime failures.

Breaking Dependency Update: In this paper, a *dependency update* refers to modifying the build specification file (e.g., *package.json* in npm) to increment the major version of a library defined by Semantic Versioning (SemVer [20]). We focus primarily on major-version updates, as they typically introduce more significant compatibility changes and represent the highest-priority maintenance task for developers, consistent with prior studies on JavaScript breaking changes [14]. Such an update becomes a *breaking dependency update* when the client invokes APIs that have changed incompatibly in the new version, leading to deprecated or invalid usages and subsequent failures.

Breaking Change Records: A *breaking change record* refers to a concrete piece of information that describes a potential breaking change introduced in a library's new major version. Such records may appear in diverse repository sources, including release notes, documentation, and project events (e.g., commits or issues). *Coarse-grained records* denote raw text extracted from a single source, which often lacks sufficient detail. In contrast, *fine-grained records* integrate information across multiple sources to provide a more complete view. When addressing a breaking dependency update, fine-grained records can offer actionable and comprehensive guidance. As shown in our study, they typically capture key dimensions such as the affected object, the characteristics of the change, and the recommended migration strategy.

2.2 Impact of JavaScript Dependency Updates

Keeping dependencies up-to-date is essential for both feature enhancements and security, as outdated libraries are a major source of vulnerabilities in modern software projects [21]. However, updating dependencies often exposes projects to compatibility issues that can lead to crashes or unexpected behavior. To mitigate this, the semantic versioning (semver) convention [20] recommends that bug fixes and security patches be released as *patch versions*, which should not introduce breaking changes. In practice, however, this convention is not always followed, which largely hinders the remediation of vulnerabilities from dependencies in practice. For instance, vulnerability fixes released in major updates take an average of 107.6 days to reach affected npm projects [22].

To examine the prevalence of this issue in the JavaScript ecosystem (npm [23]), particularly in cases where clients must adopt a major version update to address vulnerabilities, we analyze 3,921 reviewed GitHub security advisories [24] for npm packages. For each advisory, the release gap is determined by comparing the version that introduces the vulnerability with the version that provides the fix. As shown in Table 1, 72.0% of security patches are released in a new *major* version, which means that client developers have to examine the implications of breaking changes before adopting secure versions. This finding underscores that vulnerability remediation in JavaScript projects frequently requires non-trivial adaptation efforts, highlighting the practical urgency of our research objective—automating dependency upgrades in the presence of breaking changes.

Table 1. Distribution of release gaps between vulnerability-introducing and -fixing versions in npm packages.

Maximum Update Type	Count	Percentage
Major	1,865	72.0%
Minor	630	24.3%
Patch	96	3.7%
Total	2,591	100%

2.3 Motivation Example

The necessity of dependency updates underscores the importance of automated techniques. However, existing solutions are often limited to predefined transformation patterns and struggle to

handle the wide variety of breaking changes, such as method signature alterations and class removals [10, 25]. The recent emergence of LLMs offers new opportunities to address these challenges, yet current LLM-based approaches [15, 16] rely heavily on features of statically typed languages. In dynamic ecosystems, the absence of static API differencing tools and compilation error messages causes these approaches to degrade into a knowledge-free setting. This limitation motivates the need for alternative knowledge sources to guide the breaking dependency update process.

Figure 1 illustrates our motivation using a real example observed in our experiments with `Async`, one of the most widely used JavaScript libraries with over 30,000 client projects. When `Async` was updated to version 3.0, it introduced several breaking changes. One example is the function `doWhilst`, whose semantics were modified: it no longer accepts a synchronous test function. Instead, developers must adapt the test function to an asynchronous form or a callback style. This modification silently breaks client code by altering execution logic without raising explicit runtime errors. Moreover, the `Async` changelog merely announces the change without providing migration steps or an explanation of the altered execution semantics, leaving client developers responsible for further investigation.

To illustrate how current LLMs struggle in such cases, we provide OpenAI's GPT-4.1 [26] with the client code and the requirement to update the dependency. The model hallucinates the nature of the change and produces an inapplicable analysis. When supplemented with coarse-grained descriptions of the breaking change extracted from the repository changelog, the model correctly identifies the root cause but still fails to generate a valid adaptation, as it lacks usage-level details of the API. Only when supplied with fine-grained breaking change records derived from multiple repository sources (e.g., commits and test case modifications), the model succeeds in both diagnosing and handling the issue, thereby completing the update. This example highlights that breaking change records in library repositories can play a critical role in enabling successful dependency updates. In particular, fine-grained breaking change records substantially improve the ability to address compatibility issues during version updates. However, such detailed information must be specifically gathered, underscoring the need for systematic mining methods as proposed in this work.

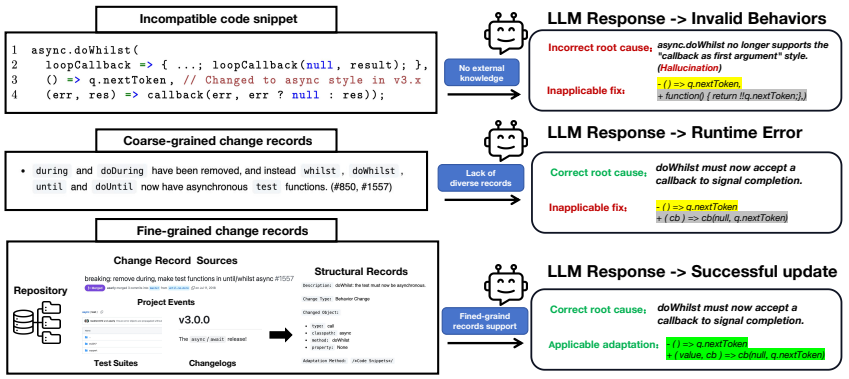


Fig. 1. Motivating example from the `Async` library showing that LLMs succeed in adapting updates only when supplied with fine-grained breaking change records.

3 EMPIRICAL STUDY ON BREAKING CHANGE RECORDS

To better collect these fine-grained breaking change records, we first conduct an empirical study to investigate how such knowledge is documented in upstream repositories, so that it can be further collected and utilized to guide the mitigation of compatibility issues when conducting major version

updates. Although prior studies have examined the types and impacts of breaking changes [27, 28], little is known about how such records are actually maintained across libraries. To build a practical pipeline for automated dependency updating, we focus on two key questions:

- **Availability.** How frequently are breaking change records maintained in library repositories, and where are they recorded?
- **Characteristics.** What kinds of information do breaking change records convey, and which key elements recur most frequently?

3.1 Breaking Change Records Availability

We first assess the availability of breaking change records across projects, as this forms the foundation for our later insights. In practice, records of breaking changes may appear in various locations, both inside and outside repositories. To investigate this, we examine the top 1,000 most depended-upon npm libraries, each used by an average of 15,830 downstream clients, and analyze how frequently breaking change records are maintained across different repository locations. To ensure coverage, we examine all major communication channels identified in prior work [27, 28] and expand the scope to include established documentation practices. By reviewing 15 relevant studies [10, 27–38], we confirm that the following sources consistently capture the dominant documentation practices for breaking changes: (1) in-repository documentation files, (2) project events [39] such as commit messages or GitHub issues, (3) repository release notes, and (4) externally maintained project homepages or wikis. Other sources, such as mailing lists or blogs exist, but are rarely linked to the repository and thus difficult to access systematically.

To reduce manual effort in detecting breaking changes, we employ a curated set of keywords[1]. The selection process draws on three sources: empirical studies that classify recurring change types and codify them as lexical tokens [10, 28], established community conventions such as conventional commits [40], and heuristic expressions observed in commit logs and issue reports. This layered approach ensures validity by grounding the terms in both research and practice, while also achieving breadth by covering structural changes, semantic refactorings, and community-specific terminology. For project events, we rely on the Conventional Commits standard [40] to detect entries explicitly flagged as breaking changes (e.g., messages prefixed with `BREAKING CHANGE`). All detected contents are manually verified by the first two authors over a period of three weeks, each with over five years of experience in the software engineering domain. Inter-annotator disagreements occur in 6.8% of cases (95% Wilson CI) and are resolved through discussion until consensus is reached.

Table 2. Distribution of breaking change records sources.

Source	Coverage	Common Sources
In-Repo Docs	560 (56.0%)	*CHANGELOG.md*, *HISTORY.md*, *Migration_GUIDE.md*
Release Notes	591 (59.1%)	Release notes, version tag notes
Project Events	343 (34.3%)	Github issues, pull requests, commit messages
External	30 (3.0%)	GitHub Wiki, Project websites
All	**838 (83.8%)**	–

Results. As shown in Table 2, 83.8% of repositories contain at least one source documenting breaking change records, demonstrating most popular JavaScript libraries provide some explicit guidance, which could be leveraged for breaking dependency updates. The statistics also reveal that this information is dispersed across multiple channels rather than centralized in a single location. Release notes (59.1%) and in-repository documentation (56.0%) emerge as the dominant forms of

[1]The set includes: (1) conventional identifiers: `breaking`, `major`; (2) common categories of breaking modifications: `remove`, `rename`, `move`, `inline`, `deprecate`, `replace`; and (3) heuristic terms: `backward-incompatible`, `drop`, `migrate`, `refactor`.

documentation, while project events (34.3%) contribute additional but less consistent coverage. Notably, only 20 repositories (2%) rely exclusively on events for breaking change documentation. External sources (3.0%) appear only rarely, meanwhile, only 1 library relies on external sources as its sole source of breaking change records. This shows that developers are more likely to maintain breaking change records in the repository. For the remaining 16.2% of libraries without identified sources, 133 (13.3%) provide no breaking change documentation in their repositories, 27 (2.7%) preserve only the latest major version while omitting historical records, and 2 (0.2%) explicitly state that the package is stable and therefore unlikely to introduce breaking changes. These results highlight both the prevalence and the fragmentation of breaking change records, underscoring the necessity of systematic mining across heterogeneous sources to ensure comprehensive coverage.

3.2 Breaking Change Records Characteristics

Despite the availability of breaking change records, their usefulness for both human developers and knowledge-based techniques depends heavily on quality. As discussed in Section 2.3, the absence of fine-grained information often causes LLMs to generate incorrect responses. Therefore, to effectively mine and utilize these records, it is essential to understand what kinds of information are most important for conveying breaking changes. To ensure representativeness while keeping manual review feasible, we apply a 95% confidence level with a 5% margin of error to the set of repositories in which we identified at least one documented source of breaking change records in the records availability study. This calculation yields a required sample of 263 repositories. For each sampled repository, we examine the identified breaking change–related documentation files (178) and release notes (213). Since the number of project events (e.g., GitHub issues) varies from only a few entries to several hundred, we randomly sample up to five events per repository, yielding a total of 173 events from 55 repositories that provide this type of record for analysis.

Prior work by Tian et al. [41] highlights the types of information that improve communication in commit messages. However, unlike commit messages, which primarily document the history and rationale of internal modifications, breaking change records are written for client developers who must quickly determine whether a change affects their code and how to adapt it. Building on this distinction, our goal is to identify the content elements that make breaking change records effective in practice. To this end, we conduct an empirical study of popular JavaScript libraries to examine recurring patterns in how breaking changes are documented. We analyze 263 sampled repositories and apply thematic analysis [42] to characterize the way upstream developers record breaking changes, following the process outlined by Tian et al. [41].

(1) **Familiarization.** Read each document to understand how changes are described and highlight segments with substantive information about the breaking change or required adaptations.
(2) **Initial Coding.** Assign descriptive codes to each segment reflecting the information conveyed.
(3) **Theme Development.** Cluster related codes into higher-level themes that capture recurring patterns in content and expression.
(4) **Refinement.** Merge overlapping themes and distinguish similar ones; two authors independently refine the themes and resolve disagreements through discussion until reaching consensus.

Based on this two-week process, we derive a structured classification to determine which elements of breaking change records provide the most useful information. Our results show that the informative records contain two dimensions: (1) a description of the changed entity (What), and (2) actionable guidance for adaptation (How). These findings highlight which information fields should be prioritized when mining breaking change records to support automated methods. Below, we summarize the major categories, their prevalence, and their implications.

Describing the Object of Change (#321, 56.9%). These records summarize breaking changes at the code-object level, capturing attributes such as API name, scope, and related identifiers. They provide the first step for client developers to determine whether their code is affected:

- *API Name Identification (#294, 52.1%)* — Specifies the exact name of the affected entity (e.g., method or property). Over half of the records include this, offering the most direct view for clients. Example: *"Updates to mapModules"*.
- *API Scope (#136, 24.1%)* — Provides scope-related details such as namespace, file path, which pinpoints specific APIs in large libraries. Example: *"Deprecated require('karma').server.start()"*.
- *Change List (#124, 22.0%)* — Groups multiple related changes to give an overview. (e.g., *Drop toBePresent, toBeEmpty. Add toExist*). Only a minority of records provide such clarity; others use descriptive phrases (e.g., *selectors like data-toggle is now data-bs-toggle*).

Describing Characteristics of the Change (#466, 82.6%). This dimension focuses on the nature of the modification and how it affects program behavior and client usage. It clarifies whether the change alters structure, semantics, or runtime behavior, and highlights the legacy effects: The implementation and experimental data used in this study are available at [19].

- *Structural Change (#248, 44.0%)* — Captures API modifications such as removal, renaming, replacement, or signature changes. Helps developers determine how to adjust prior API usage. Example: *"Drop toBePresent, toBeEmpty. Add toExist"*.
- *Behavioral or Semantic Change (#253, 44.9%)* — Describes runtime or semantic changes not visible from the interface, such as altered defaults, return types, or error handling. Example: *"The 'timeout' in 'waitFor(callback, { interval, timeout })' now uses the same clock as 'interval' "*.
- *Effect on Legacy Usage (#148, 26.2%)* — Explains the impact if updates are not applied, including failures, silent logic changes, or regressions. Example: *"If you had the parser explicitly set in your .prettierrc, update it to: parser: slang"*.
- *Change Reference (#323, 57.3%)* — Links to commits, issues, or pull requests documenting rationale or implementation, giving developers traceability. Example: *"For more details, see #1015"*.

Describing Environment or Dependency Changes (#311, 55.1%). These records capture compatibility changes beyond the API surface, including adjustments in dependencies or execution environments. Such information is critical since client systems often rely on assumptions that can silently break when requirements shift.

- *Dependency Version Requirement (#195, 34.6%)* — Specifies new constraints on dependent libraries or frameworks, such as raising minimum versions. Example: "*[Breaking] update svgo to v2*".
- *Environment or Platform Support (#228, 40.4%)* — Records changes in supported runtimes, such as Node.js versions. Example: "*Updates the engines from $\hat{1}$6.14.0 || >=18.0.0 to $\hat{1}$8.17.0 || >=20.5.0*".

Describing Migration Strategy (#252, 44.7%). These records capture the extent to which changelogs assist developers in mitigating the impact of a breaking change, going beyond stating the modification to provide actionable guidance for adapting client code or environments.

- *Textual Migration Instruction (Code-Level) (#160, 28.4%)* — Provides prose instructions describing how client code should be adapted. Example: "*Use Object.assign instead of xtend*".
- *Textual Migration Instruction (Environment-Level) (#46, 8.2%)* — Specifies required adjustments to configuration files, build settings, or environment setup. Example: "*Change engines field to require node>=12*".
- *Code Snippet or Diff Example (#88, 15.6%)* — Presents concrete migration support in the form of before/after code snippets or diffs. Example: "*-import { take } from 'redux-saga', +import { take } from 'redux-saga/effects*"".
- *External Reference (#139, 24.7%)* — Refers developers to additional resources such as migration guides. Example: "*Upgrade guide: {url}*".

Table 3. Information types conveyed in breaking change record across documentation sources.

Category	Docs	Notes	Project Events
Object of Change			
API Element Identification	56.7%	39.0%	63.6%
API Scope	25.3%	6.1%	45.1%
Change List	28.7%	8.9%	31.2%
Characteristics of Change			
Type of Structural Change	48.3%	35.7%	49.7%
Behavioral or Semantic Change	53.4%	32.9%	50.9%
Effect on Legacy Usage	29.8%	16.4%	34.7%
Change Reference	52.8%	39.0%	84.4%
Dependency and Env. Support			
Dependency Version Requirement	20.2%	52.1%	27.8%
Platform or Runtime Constraint	23.0%	63.9%	29.5%
Adaptation Guidance			
Textual Migration (Code-Level)	34.8%	25.4%	25.4%
Textual Migration (Env-Level)	11.2%	5.2%	8.7%
Code Snippet or Diff Example	14.6%	8.9%	24.9%
Linked External Reference	28.1%	22.1%	24.3%
Implicit Change	16.9%	15.0%	12.7%

Implicit Change (#84, 14.9%). This notable category captures mentions of a breaking change that do not specify what part of the system is affected and how developers should adapt. Such descriptions provide minimal actionable information and often leave developers uncertain about the necessary modifications. Making the dependency updates unavailable. *Example:* "This will be a breaking change because the rules object has changed".

Our study shows that the most important fields for conveying breaking changes are the *object of change* (56.9%) and the *characteristics of the change* (82.6%). These elements allow developers to recognize which API entity is affected and understand how its structure or behavior has been modified. However, we find that 26.4% of records describe only the characteristics of the change without explicitly identifying the object (e.g., "Remove deprecated APIs"), and 14.9% of records are entirely implicit, merely stating that a breaking change exists without clarifying what was changed and how to adapt. Such gaps mean that, while most records convey what has changed, they often fall short of providing complete, actionable support for developers to effectively adapt their systems. This observation highlights the need to systematically recover missing fields to make records usable for automated techniques.

We also observe substantial variation across documentation sources. Release notes, as the most prevalent source, tend to provide broad descriptions of changes but often lack explicit identification of the affected API objects and concrete contrasts with prior behavior. In-repository documentation offers more detailed coverage, with higher rates of API element identification and behavioral characterization, but still omits key aspects such as change types (48.3%) and scopes (25.3%). In contrast, project events such as commits and pull requests are the richest source, capturing explicit object- and scope-level details, structural and semantic differences, impacts on legacy usage, and nearly universal references to related issues or pull requests. However, they are less direct for client developers and require additional effort to associate with a specific breaking change.

Finding

Analysis of breaking change records shows that most document change characteristics (82.6%), specify the affected object (56.9%), and nearly half provide migration strategies

(44.7%). However, 14.9% of the records mention the existence of breaking changes without clarifying what was actually modified. Source types also differ in emphasis: release notes and documentation often raise awareness but lack precision, whereas project events provide richer details but are harder to interpret. These results indicate that no single source is sufficient; automated methods should integrate heterogeneous sources and refine records to ensure that essential fields are captured.

4 APPROACH

Our study of breaking change records highlights both their prevalence and potential for mitigating the challenges of JavaScript dependency updates. However, the diversity of sources and the variable quality of records necessitate a comprehensive workflow that can systematically mine and refine information across library repositories. To this end, we design BDUpdater, an agent-based framework that automates JavaScript breaking dependency updates.

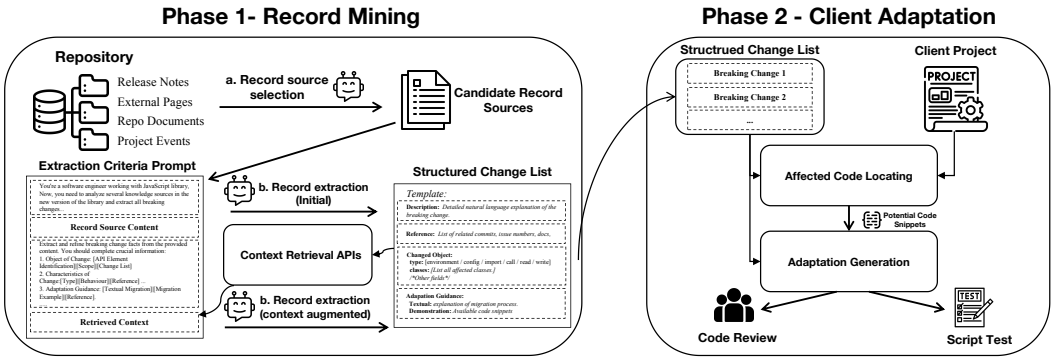


Fig. 2. The overview of BDUpdater.

BDUpdater operates on the client side, assisting downstream developers who intend to upgrade a dependency to its next major version. Given a client codebase C and a dependency library D to be updated from version $V_{target-1}$ to V_{target} (e.g., from v2.x to v3.x), BDUpdater autonomously mines breaking change records, detects affected locations in client code, and generates candidate adaptations that can be vetted and refined by developers. By automatically handling even a subset of breaking changes, BDUpdater reduces the manual effort required for dependency maintenance.

As shown in Figure 2, the workflow of BDUpdater consists of two main stages: *record mining* and *client adaptation*. In the *record mining* stage, an LLM agent selects candidate sources (Section 4.1.1) relevant to the new major version, guided by our study on record availability (Section 3.1). The agent then summarizes and extracts breaking changes according to the classification in Section 3.2, producing structured records that capture essential attributes. To improve reliability, we design a search-oriented MCP server that performs chained extraction. In this process, the agent iteratively retrieves additional sources or refines missing fields using a set of refinement tools until the extraction criteria are satisfied. The output of this stage is a structured list of breaking change records. In the *client adaptation* stage, for each breaking change record, BDUpdater applies a lightweight static analysis (Section 4.2.1) to identify candidate code snippets that may be affected. These snippets are examined by an adaptation agent, which uses the mined records to determine whether each location is truly impacted and, if so, generates a corresponding adaptation. The final output is an adaptation set that resolves the incompatibilities introduced by the dependency update.

4.1 Record Mining

In this stage, BDUpdater collects and structures breaking change records from repositories of the dependency being updated. The process consists of two steps: *source selection*, *record extraction*.

4.1.1 Source selection. Breaking change records are often distributed across heterogeneous sources. BDUpdater first identifies potential sources using the same keyword-based filtering criteria described in Section 3.1. However, this step alone may still preserve a large number of candidates, since the selected keywords can also appear in less relevant documentation, such as API references, usage examples, or general project descriptions. To further refine the search space, the *record mining* agent examines the path and file names of available candidates, as these structural cues often reveal a document's purpose (e.g., `update_to_v6.md` indicating migration guidance), allowing the agent to identify files more likely to contain breaking change information. The agent then produces a ranked list of the most promising source files for prioritized record extraction.

4.1.2 Record extraction. From the candidate sources, BDUpdater proceeds to extract explicit breaking change records. As illustrated by our studies, release notes and in-repository change documents often provide high-level summaries of breaking changes, while project events like GitHub issues capture more fine-grained details. Following this observation, BDUpdater first attempts to inspect the release notes of the target version and then extends to other sources such as changelogs. Once a high-level overview of the listed breaking changes is obtained, BDUpdater proceeds into all related project events to capture additional details. To avoid overwhelming the agent with excessive context, only event metadata (e.g., commit titles) is initially provided. The full content of these events is reserved for the subsequent refinement process, where missing attributes can be filled in as needed.

There is a typical scenario where record sources do not provide the explicit characteristics of a breaking change, which makes it difficult to precisely locate and adapt the affected code in client projects. For example, the changelog for Async 3.0 lists several breaking changes in the new version. However, as highlighted, it does not specify the changed objects or provide concrete migration guidance. Instead, vague expressions are used, such as "Most Async methods" and "the event-style methods like `q.drain`." In such cases, LLM-generated breaking change records are often incomplete, which leads to failures in identifying the affected client code.

To address this issue, BDUpdater not only extracts breaking change records from the candidate sources identified during *record selection*, but also builds a search-oriented MCP server that enables the record mining agent to validate and refine these records using a series of context retrieval tools. Specifically, BDUpdater enforces the agent outputs to follow a predefined template using a JSON schema together with a post-processing JSON repair module. The mandatory fields include: (1) a natural language description, (2) a *changed object* section specifying both change types (e.g., modules, functions, properties) and changed entities (e.g., module names, method names, class names), and (3) a *migration* field containing textual migration guidance or code examples. During the extraction process, the agent iteratively checks completeness and validity of each field against the classification defined in Section 3.2, with the available retrieval tools summarized in Table 4.

Table 4. List of context retrieval tools.

Tool name	Description	Output
`search_class(cls)`	Search for class `cls` in the codebase.	Properties and methods of the class prototype.
`search_method(m)`	Search for method `m` in the codebase.	Signature and body of the method.
`read_file(path)`	Retrieve the content of file.	The file contents
`search_pull_request(id)`	Retrieve the content of a pull request.	Pull request description and related commit titles.
`search_issue(id)`	Retrieve content of an issue.	Issue description and associated commit contents.
`search_commit(id)`	Retrieve content of a commit.	Commit messages and associated diff files.

When key fields lack information, the agent invokes additional tools to retrieve the necessary context, thereby completing the record. For instance, after extracting a breaking change record, BDUpdater collects related project events and employs API search tools to validate the existence of referenced APIs, thus completing the *changed object* field. Similarly, BDUpdater mines user discussions, tests, and examples in the project events to populate the *migration* field with both descriptive guidance and concrete code. This process ensures records are not only extracted but also enriched with sufficient structured details to support subsequent client-side adaptation. BDUpdater also implements an on-demand mechanism for handling large contexts. For example, when a pull request contains multiple commits, the agent is first prompted to rank commits by their textual relevance to the breaking change and then analyze them sequentially. This approach avoids arbitrary context retrieval and reduces resource consumption.

4.2 Client Adaptation

Once a structured list of breaking change records has been obtained, BDUpdater applies them to adapt client code. Specifically, BDUpdater processes each mined record individually rather than performing an all-in-one adaptation, in order to avoid excessively large LLM contexts. This stage consists of two steps: *locating affected code* and *adaptation generation*.

4.2.1 Affected code localization. The structural breaking change records provide explicit references to changed APIs. However, a simple string match in the client code would generate many false alerts due to variable aliasing and APIs with identical names. To address this, BDUpdater applies a lightweight static analysis of the client codebase based on the pattern language design of Tapir [43]. We omit the details of the pattern language here and refer the reader to Tapir for a full description. The objective is to identify candidate locations that may be affected by the mined breaking changes.

The static analysis consists of three components: (i) a lightweight alias analysis that approximates possible variable and property aliases, (ii) access path inference that reconstructs connections between client expressions and imported modules, and (iii) pattern matching to identify expressions whose access paths correspond to API access patterns of interest.

For each source file s, alias analysis constructs a map $\alpha_s : Var \cup Prop \rightarrow \mathcal{P}(Exp)$, where Var denotes declared variables, $Prop$ property names, and Exp the set of expressions in s. For $x \in Var$, $\alpha_s(x)$ denotes expressions that may alias x; for $f \in Prop$, $\alpha_s(f)$ denotes expressions that may alias property f. For example, from Listing 1, alias analysis yields $\alpha_s(\texttt{libAsync}) = \{\texttt{require('async')}\}$ and $\alpha_s(\_) = \{\texttt{require('lodash')}\}$. Access path inference then distinguishes the calls as $\langle\texttt{async}\rangle\texttt{.map}$ versus $\langle\texttt{lodash}\rangle\texttt{.map}$.

```
1 const libAsync = require('async');
2 const _        = require('lodash');
3
4 libAsync.map(files, worker); // uses Async's map
5 _.map(items, f);             // uses Lodash's map
```

Listing 1. Minimal client snippet with homonymous API

To ensure that all potentially affected code is located, we relax the analysis into two core checks that directly leverage the breaking change records. Formally, let n be an AST node in the client code. We consider n a candidate match if it satisfies: (1) $name(n) \in \{p, f, m\}$, where p, f, and m denote the property, function, or module names listed in the mined breaking change records; and (2) $accesspath(n) > D \vee accesspath(n) = U$, where D is the target dependency library and U denotes an unresolved access path. These conditions ensure that we capture both direct usages of the target library and ambiguous cases where aliasing information is unavailable. This conservative treatment improves recall but may introduce false positives in cases of unknown access paths. Such false

positives are subsequently filtered by the adaptation agent, which inspects the usage context (i.e., block-level syntactic unit) to determine whether the snippet is affected by the breaking change.

4.2.2 Adaptation generation. For each potentially affected code snippet, BDUpdater forwards it to an adaptation agent that operates in two phases. First, the agent verifies whether the snippet corresponds to a changed API and is incompatible with the new version by cross-checking the mined breaking change records with its usage context and the localized source file. If confirmed, the agent generates an adaptation that adapts the client code to the updated API, together with any required dependency update instructions. The output of this stage is an adaptation set that can be applied to the client project, intended for validation by developers through testing and review.

BDUpdater also supports in-place adaptation of the client code. In this mode, the adaptation agent iteratively adapts each affected location within the file, checking for side effects on the changed file before moving to the next site. Once all locations are resolved, BDUpdater updates the dependency version and re-executes the client's test suite to confirm success.

5 EXPERIMENT

We design a controlled experimental setup to evaluate the effectiveness of BDUpdater in handling breaking dependency updates. Our study is guided by the following research questions:

- **RQ1 (Effectiveness):** How effective is BDUpdater at automating breaking dependency updates?
- **RQ2 (Comparison with Zero-Shot Baseline):** How does BDUpdater perform relative to direct LLM use with different knowledge?
- **RQ3 (Efficiency and Impact of Different LLMs):** What are the time and monetary costs of applying BDUpdater? And how consistent is BDUpdater 's performance when integrated with different LLMs?

5.1 Datasets

Our experiments are based on real-world dependency updates across popular JavaScript libraries. In total, the dataset covers 13 libraries and 84 client–library pairs, spanning both contemporary and historical breaking dependency updates. To ensure both realism and reliability, we construct a dataset of real-world dependency updates across widely used JavaScript libraries. We begin with the top 1,000 `npm` packages ranked by libraries.io [44]. To mitigate data leakage (i.e., LLMs being pre-exposed to breaking-change records or client code), we restrict candidates to libraries that released a new major version after June 2024, which is beyond the knowledge cutoff of widely used models (e.g., GPT-4.1). For each candidate version, we manually verify that breaking changes were introduced, then search GitHub for downstream clients that depend on older versions of the library.

Since JavaScript correctness cannot be validated via compilation, we use runtime behavior as the primary oracle. A client–library pair is retained if updating the client to the target major version causes failures in its `package.json` scripts (e.g., `test`, example runners). We execute these scripts to confirm that post-update failures are reproducible, ensuring that the adapted client code must be runnable and functionally plausible under the same scripts. Although many libraries publish new major versions, their recency often limits the availability of affected downstream clients with runnable test suites, which explains the modest dataset size despite the large candidate pool. Following this procedure, we obtain 7 library updates with 44 affected client projects. To further enlarge the dataset, we incorporate part of the dataset from JSFIX [14], which contains historical breaking dependency updates. Because some updates introduce an exceptionally large number of breaking changes within a single release (e.g., Lodash [45] introduces hundreds of breaking changes), identifying, refining, and consolidating all change documents into structured BC records would exceed current LLM context limits and overwhelm the adaptation process. Therefore, we

filter out library–client pairs involving more than 20 breaking changes. This yields an additional 6 library updates with 40 affected client projects.

Table 5. Composition of evaluation dataset across different timeframes.

Time	Library	BC	Clients	Module	Property	Function	Env
< 2024	*uuid* 3.0.0	1	1	0	1	0	0
	redux 4.0.0	2	4	1	0	1	0
	chalk 2.0.0	3	10	0	0	3	1
	async 3.0.0	5	8	0	1	3	1
	bluebird 3.0.0	7	9	2	3	2	0
	node-fetch 2.0.0	9	8	1	2	6	0
2024	*execa* 9.0.0	14	7	0	2	11	1
	jsdoc-to-markdown 9.0.0	3	4	0	3	0	0
	replace-in-file 8.0.0	7	6	1	3	1	2
2025	*commander.js* 13.0.0	4	5	0	4	0	0
	ember-qunit 9.0.0	2	7	0	0	3	0
	svgo 4.0.0	5	9	2	0	1	2
	web-vitals 5.0.0	3	6	0	1	1	1
Total		**65**	**84**	**7**	**20**	**32**	**8**

The details of the dataset are shown in Table 5. Here, **BC** denotes the number of breaking changes introduced in the major version, and **Clients** refers to the number of open-source projects that fail after upgrading. The remaining columns summarize the types of breaking changes. *Module* changes refer to entire modules being removed or relocated. *Property* changes involve the removal or relocation of object properties (often methods). *Function* changes correspond to modifications in the signature or expected usage of individual functions. Finally, *Env.* changes apply at the package level rather than specific APIs, such as dropping support for outdated Node.js versions.

5.2 RQ1: Effectiveness

Evaluation setting. In this RQ, we evaluate the effectiveness of BDUpdater in adapting client projects to breaking dependency updates. For the 84 client–library pairs in our dataset, we update the dependency to the target major version and execute the `package.json` scripts (e.g., `test`, example runners) to identify scripts that fail after the update. We then apply BDUpdater, re-run the same scripts, and label the case a success if the previously failing scripts complete without error. Since passing tests may not ensure semantic correctness, we additionally compare BDUpdater 's adaptation with a developer-authored migration when available and conduct a manual evaluation in which two authors independently label adaptations as *identical*, *semantically equivalent*, or *plausible but not equivalent*. Specifically, semantically equivalent adaptations differ only syntactically from the ground truth or use APIs with the same behavior. Code that passes tests but is not behaviorally identical is plausible but not equivalent. All subsequent experiments are conducted on a server running Ubuntu 20.04, equipped with 251 GB of memory and two Intel Xeon Gold 6346 CPUs.

Overall results. The results of the overall success are shown in Table 6. Across 84 clients that initially fail to execute their built-in scripts after the update, BDUpdater restores successful execution in 76 cases, yielding an overall success rate of 90.5%. These results indicate that BDUpdater effectively generates adaptations that restore runnable project functionality. During the adaptation process, 91.8% of potentially affected locations are confirmed and resolved in a single query. For the remaining cases, BDUpdater queries file-level context to double-check the adaptation.

Record Mining Results Analysis. Besides adaptation success, we report the results of the record mining stage in Table 7, as the effectiveness of BDUpdater critically depends on the quality of the mined breaking change records. For each library update, two authors independently construct a reference set of breaking changes by manually inspecting official repositories and existing ground-truth datasets. All mined records are then mapped to this reference set. A mined record is classified

Table 6. Adaptation Effectiveness of BDUpdater (npm script execution).

Library	Clients	Success	Fail	Success Rate	Mod. Files	Mod. Hunks
uuid	1	1	0	100%	1.0	3.0
redux	4	3	1	75.0%	1.0	1.0
chalk	10	10	0	100%	1.6	4.7
async	8	8	0	100%	1.3	3.6
bluebird	9	7	2	77.8%	2.1	4.2
node-fetch	8	6	2	75.0%	1.9	3.9
execa	7	6	1	85.7%	1.3	5.3
jsdoc-to-markdown	4	4	0	100%	1.0	4.0
replace-in-file	6	6	0	100%	1.0	4.0
commander.js	5	4	1	80.0%	1.8	4.0
ember-qunit	7	7	0	100%	1.6	1
svgo	9	8	1	88.9%	1.0	1.1
web-vitals	6	6	0	100%	1.3	3.0
Total	**84**	**76**	**8**	**90.5%**	**1.3**	**3.4**

"Clients": the number of clients; "Success"/"Fail": outcomes of the relevant `npm` scripts after applying BDUpdater; "Success Rate": the fraction successfully upgraded; "Mod. Files"/"Mod. Hunks": the average numbers of modified files and edit hunks across successful adaptations.

Table 7. The Record Mining Effectiveness and Record Categories Extracted in Different Mining Stages.

Stage	Recall	Precision	Extracted Record Categories			
			C1	C2	C3	C4
Initial	97.0%	96.9%	69.7%	95.5%	4.5%	77.3%
Augmented	98.5%	97.0%	90.9%	97.0%	4.5%	90.9%

as a true positive (TP) if it corresponds to a ground-truth breaking change, and as a false positive (FP) if it corresponds to a non-existent breaking change. Since the availability of ground truth alone does not fully capture the quality of mined records, the last four columns of Table 7 report the distribution of extracted record categories. These statistics are computed as the proportion of records whose categories are correctly identified relative to the total number of mined records.

Across the evaluated library versions, BDUpdater successfully identifies the majority of documented breaking changes during the initial mining process, achieving a high recall of 97.0%. These results benefit from the exhaustive extraction of information from multiple sources and confirm the effectiveness of repository-based mining. The missed cases are all undocumented breaking changes that lack any explicit evidence in the repositories. Notably, BDUpdater is also able to mine undocumented breaking changes when analyzing unit tests associated with other documented breaking changes, as test code is often grouped within the same file and BDUpdater analyzes and confirms incompatibilities simultaneously. BDUpdater also achieves high precision, as its main execution flow focuses exclusively on breaking changes with explicit documentation. The few false positives arise from cases where BDUpdater mistakenly collects breaking changes from older versions. A typical example is a major release that announces a deprecation but does not actually remove the deprecated APIs (e.g., by implicitly replacing old functions with aliases). Motivated by our preliminary study in Section 3, we observe that record availability (i.e., precision and recall) alone does not fully reflect the effectiveness of the mined results. Although BDUpdater explores the main repository sources, the mined breaking change records may still lack essential information and require further retrieval to capture sufficient change context. As shown in Table 7, while the majority of records specify the characteristics of breaking changes, only 69.7% correctly identify the changed API objects from explicit sources, and only 77.3% provide adaptation guidance. Consequently, 11 out of the 13 evaluated libraries proceed to the subsequent context augmentation stage, which improves both metrics to 90.9%. The relatively low occurrence of category C3 is

expected, as only a small fraction of breaking changes require environment-level modifications, which typically occur when libraries drop support for long-unmaintained components.

Failure cases. We then examine the client projects that BDUpdater fails to update. One notable case involves a *node-fetch* client that re-wraps the official API. In this scenario, methods affected by breaking changes are overridden within the wrapper, yet BDUpdater still applies modifications because it lacks awareness of the client's internal re-implementation. This illustrates a fundamental limitation of BDUpdater's on-demand context retrieval: the LLM struggles to distinguish between APIs originating from the target library and those subsequently modified by the client. A second set of failures arises from breaking changes that are not explicitly documented in the corresponding repositories. For example, *commander.js* silently changed internal state variables without documentation or related tests, as these were not considered part of its public API. However, one client relied on this variable to structure its execution logic. Such cases highlight an inherent limitation of BDUpdater's mining process: when library maintainers omit seemingly minor internal changes from documentation, the resulting incompatibilities may still surface in client code. Finally, two clients fail not because of unhandled breaking changes, but due to test code that is indirectly affected by API modifications. For instance, another *node-fetch* client used a proxy library for testing. When the dependency is updated, the proxy continued mocking the old API version, causing the test suite to break despite the functional code being otherwise correct.

Semantic correctness. To validate the effectiveness beyond automatic metrics, we conduct an evaluation on the semantic correctness of the successful updates. For each client, we retrieve the developer-authored adaptations when available, yielding 24 ground-truth adaptations across 8 libraries. We then compare BDUpdater 's patches with these ground truths to assess semantic equivalence. The results in Table 8 show that BDUpdater achieves high effectiveness when compared against ground-truth developer migrations. Across all 24 collected cases, 33.3% of the generated adaptations are *identical*, 62.5% are *semantically equivalent*, and only one case (4.2%) is classified as *plausible* but not equivalent. In total, 95.8% of the adaptations restore functionality in a manner consistent with developer intent.

Table 8. Effectiveness evaluation by ground-truth comparison (RQ1).

	Identical	Semantically Equivalent	Plausible
All (24)	33.3% (8)	62.5% (15)	4.2% (1)
History (12)	50.0% (6)	41.7% (5)	8.3% (1)
New (12)	16.7% (2)	83.3% (10)	0.0% (0)

To assess the risk of data leakage, we break the dataset by time period. The *History* dataset (12 cases), collected before the knowledge cutoff of GPT-4.1, may partially overlap with training data. In this set, 50.0% of the adaptations are identical and 41.7% are semantically equivalent. However, 5 out of 7 identical cases involve only single-line changes, where identical adaptations are acceptable and expected. By contrast, the *New* dataset (12 cases), which includes only libraries with major releases after the LLMs' knowledge cutoff, provides a more realistic test of generalization. Here, BDUpdater produces fewer identical adaptations (16.7%), but the vast majority (83.3%) are semantically equivalent, indicating that while the surface form may differ, the intended functionality is correctly preserved. The results show that BDUpdater is capable of generating practical migrations comparable to those produced by human developers. Moreover, the dominance of semantically equivalent adaptations in the new dataset demonstrates that BDUpdater can reliably produce functionally correct updates, even for breaking changes that were previously unseen.

5.3 RQ2: Comparison with Zero-Shot LLM Baseline

Evaluation setting. Automating breaking dependency updates requires both locating and adapting affected client code, yet currently there is no established baseline existing for JavaScript. For

comparison, we implement a zero-shot approach inspired by Lukas et al. [15], which targets single file level modifications. The zero-shot baseline relies on natural language descriptions of the library update task, provides no in-context examples, and takes the potentially affected file as input. Since existing designs for Java [15, 16] rely on compilation errors to localize changes, a strategy infeasible for JavaScript, we adapt the setting by manually identifying client projects that require single file updates and supplying zero-shot LLMs with the affected API and update requirements. To avoid duplication, only one client is retained per unique breaking change, yielding 26 clients in total. Update success is then evaluated using the same npm script execution protocol as in RQ1.

Table 9. Comparison of adaptation success across baselines and BDUpdater (RQ2).

Setting	Success Rate	Type of Breaking Change		
		Module (4)	Property (8)	Function (14)
Zero-shot LLM (Knowledge-free)	26.9% (7/26)	2	2	3
Zero-shot LLM (Coarse-grained records)	61.5% (16/26)	4	5	7
BDUpdater	**92.3% (24/26)**	**4**	**8**	**12**

We evaluate 26 client–library cases under three settings: (i) *Knowledge-free*, where the affected client files are specified but no external knowledge is provided to the LLM. (ii) *Coarse-grained records*, where breaking change records from a single source (e.g., changelogs) are provided to the LLM, and (iii) *Fine-grained records*, where mined breaking change records guide BDUpdater.

Results. Table 9 shows a clear progression in adaptation effectiveness across the three evaluated settings. Even when the affected client files are explicitly specified, LLMs still struggle to perform breaking change adaptations without additional context (26.9% success rate). This finding is consistent with prior studies in other ecosystems [15, 16], which emphasize that multi-dimensional knowledge is often necessary for successful migration. In this regard, mining breaking change information from repositories provides substantial improvements. Providing coarse-grained records already improves performance to 61.5% (16/26), confirming that even high-level documentation can assist LLMs in generating more accurate updates. When equipped with fine-grained breaking change records, BDUpdater achieves the highest success rate at 92.3% (24/26).

By investigating the presence of different record categories under the baseline setting, we observe a clear relationship between the availability of required categories and successful adaptation. For failed cases in the zero-shot setting with coarse-grained records, every case lacks at least one essential record category (with the exception of C3). Among these failures, 70% lack adaptation guidance, and 50% fail to specify the affected API objects and the characteristics of the change. In contrast, all successful cases consistently contain a complete set of record categories. Across different change types, function-related modifications prove to be the most challenging. This may be due to the greater semantic complexity associated with function-level changes compared to module or property updates. Indeed, in the two function-related cases that remain unsolved, both BDUpdater and the baselines fail to produce correct adaptations.

5.4 RQ3: Efficiency and Impact of Different LLMs

Table 10. The average execution time and cost for two main phases of BDUpdater with different models.

Process	BDUpdater (GPT-4.1)		Claude-Sonnet 3.7		DeepSeek-Chat V3		Qwen-plus	
	Time (s)	Cost (USD)	Time (s)	Cost (USD)	Time (s)	Cost (USD)	Time (s)	Cost (USD)
Record Mining	92.0	0.049	285.4	0.435	290.6	0.016	1162.0	0.092
Client Adaptation	69.5	0.047	55.8	0.064	137.8	0.007	47.1	0.006

Evaluation setting. To evaluate the system efficiency, we measure both runtime and monetary cost when applying BDUpdater. Table 10 summarizes the average time and cost for the two

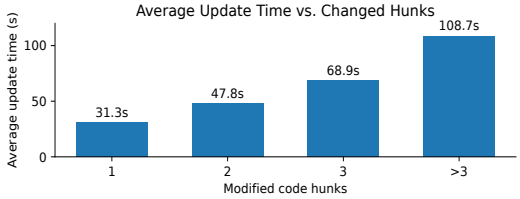


Fig. 3. Average update time by number of hunks.

main phases of BDUpdater over the libraries and clients in our dataset. To further present the impact of model choice, we run BDUpdater with flagship LLMs from multiple vendors. Specifically, we evaluate GPT 4.1 [26], Claude 3.7 Sonnet [46], DeepSeek–Chat V3 [47], and Qwen–Plus [48], selected for strong performance on code-related tasks. All models are evaluated under the same protocol and accessed via their official APIs using default temperature settings.

Cost Analysis. When executed with the default model of BDUpdater (i.e., GPT-4.1), the record mining phase requires 92.0 seconds and costs $0.049 per library version, while the client adaptation phase averages 69.5 seconds and costs $0.047 per client. Notably, the mining phase needs to be executed only once for each new library version. Once breaking change records are extracted, they can be cached or shared with the community, enabling subsequent client projects to reuse the results without incurring additional mining costs. Across different LLMs, the record mining phase takes between 285.4 and 1162.0 seconds and costs $0.016–$0.435, whereas the client adaptation phase requires 47.1–137.8 seconds and costs $0.006–$0.064. All evaluated models spend more time in the record mining phase, reflecting the inherent complexity of mining breaking changes from multiple repository sources. Monetary cost largely follows model pricing. For instance, Qwen–Plus has a token price that is approximately 6% of OpenAI's, resulting in the lowest adaptation cost, although it also exhibits the longest mining time (1162 seconds).

We observe that adaptation time is strongly correlated with the number of modified hunks. As shown in Figure 3, when the required modification involves only a single hunk, the update completes quickly, averaging 32.2 seconds. Adaptations with two or three hunks take moderately longer, averaging 47.8 and 68.9 seconds respectively. For larger changes involving more than three hunks, the adaptation time increases substantially to 109.1 seconds on average. This trend suggests that more complex updates require additional reasoning and validation, leading to higher latency. Overall, the efficiency analysis shows that both phases of BDUpdater incur modest computational and monetary costs. Furthermore, by reusing mined records across projects, the cost per adaptation can be reduced further, highlighting the feasibility of deploying BDUpdater at scale.

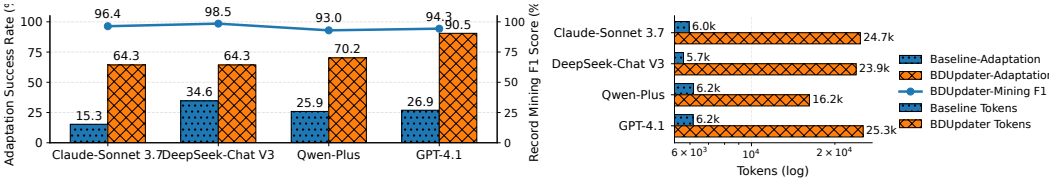


Fig. 4. Performance of BDUpdater across different LLMs compared to zero-shot baseline.

Comparison Results. Figure 4 shows that BDUpdater is effective across different LLMs. All evaluated models achieve high F1 scores in the record mining stage, and even the lowest-performing models reach an adaptation success rate of 64.3%. Claude-Sonnet 3.7 and DeepSeek-Chat V3 exhibit similar performance, which is likely constrained by their limited context lengths (100–200k tokens), reducing precision when mining complex repository records. Qwen-Plus achieves a higher success rate of 70.2%, while GPT-4.1 substantially outperforms all other models with a 90.5% success rate, reflecting its advanced code understanding and robustness in both record mining and client

adaptation. These results indicate that while BDUpdater can operate effectively with a range of LLMs, extended context capacity is critical for maximizing performance in automated dependency updates. Compared with the zero-shot knowledge-free baseline in RQ2, BDUpdater improves the success rate by 1.86× to 4.18×, suggesting that despite differences in model capabilities, BDUpdater consistently delivers significant improvement due to the mined records. However, BDUpdater also requires 2.61× to 4.19× more tokens for exploratory repository mining, since the zero-shot baseline is directly provided with the potentially affected files. This additional token consumption represents the necessary cost of locating client incompatibilities to enable effective adaptation.

6 DISCUSSION

6.1 Threats to Validity

While BDUpdater shows promising results, several threats to validity remain:

Internal Validity. (1) The preliminary study (Section 3) and semantic evaluation (Section 5.2) involve manual assessment; although consensus was used to reduce subjectivity, some inaccuracies may persist. Accordingly, we open-source the data for public evaluation. (2) To mitigate data leakage, the study focuses on newly released library versions, and therefore does not include a large-scale evaluation of historical versions. We leave this to future work.

External Validity. (1) The current implementation targets JavaScript. While the framework is conceptually portable, applying it to other dynamic languages (e.g., Python) would require adapting the static analysis components. However, the agent component could be language-agnostic. (2) Although the evaluation considers LLMs from multiple vendors, it does not cover all model variants, and outcomes may vary with specific implementations.

6.2 Limitations and Future Works

Although the insights of BDUpdater are shown to be effective on our dataset, several inherent limitations remain due to constraints in current LLM architectures and static analysis techniques:

Context Window. The model struggles to maintain global coherence in large-scale projects because of limited LLM context windows. For large library updates, such as core libraries (e.g., *lodash* [45]), a single major version may involve hundreds of breaking change documents. BDUpdater is currently less effective at generating precise breaking change records for such cases. Recent research on LLM trajectory reduction [49] offers a promising direction for mitigating this limitation by refining and compressing the processing context.

Semantics Misunderstanding. The client update agent relies on the code comprehension ability of LLMs. While LLMs can generally interpret the context to decide whether a modification is needed, they struggle in scenarios where clients rewrap or override official APIs. A potential mitigation here is to use dynamic analysis to distinguish library APIs and customizations.

Impacts of Documentation. BDUpdater operates on documented breaking changes and thus applies to both major and non-major updates, but may not capture undocumented changes. However, our study shows that 83.8% of the libraries in our dataset provide breaking change documentation. In addition, prior work reports that 78.1% of non-major breaking changes are documented in the dataset studied by [27], and 94.1% of surveyed breaking changes are documented in Tapir [43]. These findings indicate that documented breaking changes constitute the majority of real-world cases, supporting the practical applicability of BDUpdater. At the same time, maintainers are encouraged to provide complete breaking change records across different categories, with clearly identifiable locations in their repositories. Since our current prototype focuses primarily on in-repository documentation, undocumented changes remain an open challenge. Incorporating additional information from external sources, such as developer forums and technical blogs, may help mitigate this limitation, which we leave as an important direction for future work.

7 RELATED WORKS

Breaking Changes in Dependency Updates. Breaking dependency updates are common in software projects [34, 35, 50] and often discourage developers from upgrading their dependencies [51–53]. Nevertheless, keeping dependencies up to date is crucial for security, as outdated libraries may contain vulnerabilities [4, 54, 55]. Most prior work on breaking dependency updates has focused on analyzing their frequency, causes, and identifying breaking changes in library code [34, 35, 50, 56]. In contrast, research on automated breaking change update is scarce.

Automated Breaking Dependency Updates. Automatic client transformation was first introduced by Coccinelle [57, 58] for Linux drivers and later extended to Java [59]. Subsequent work automatically computed differences between library versions and, based on these differences, either suggested adaptations for client code or directly transformed it to accommodate API breaking changes. Such tools typically rely on expert-written templates to detect structural API modifications [11–13, 60], and generally do not address behavioral changes. For JavaScript, JSFIX [14] shows that semantic patches can capture JavaScript-specific update patterns. However, JSFIX requires experts to manually identify each breaking change and encode the corresponding fix patterns. Recent advances in LLMs explore applying LLMs to automate updates in Maven-based Java projects [15, 16]. These approaches depend on API differencing tools to compute structural changes between library versions and remain limited in handling semantic changes. Because Java APIs provide fully qualified names, such differencing is feasible in Java but not in JavaScript, where library interfaces are highly dynamic. In practice, detection tools for JavaScript can only report raw type-level changes, requiring additional human confirmation [61, 62]. BDUpdater is designed to enable automated dependency updates in the absence of reliable API differencing tools. Moreover, our approach could integrate detection methods to capture undocumented breaking changes and further improve effectiveness.

Patch porting. Unlike updating to a secure version, patch porting backports security fixes to older releases using either pattern-based approaches (e.g., FixMorph [63], TSBPORT [64]) or LLM-based methods (e.g., PPATHF [65], MYSTIQUE [66]). While it preserves overall code compatibility, this practice could lock projects into outdated versions and hinder future update.

8 CONCLUSION

Breaking dependency updates are particularly difficult in JavaScript, where techniques designed for statically typed languages fail without API differencing or compiler diagnostics. We propose BDUpdater, a hybrid agent-based framework that mines and refines breaking change records and applies them in a pre-mitigation workflow to localize and adapt affected code. An empirical study identified essential record elements, and evaluation on 13 libraries and 84 clients showed a 90.5% success rate with high semantic equivalence at modest cost. These results demonstrate that combining repository-level knowledge mining with LLM-guided adaptation provides an effective and practical solution for breaking updates in dynamic language ecosystems.

9 ACKNOWLEDGEMENT

This work was partly supported by NSFC under No. U2441239, 62302443 and U24A20336, the China Postdoctoral Science Foundation under No. 2024M762829, 2025M781523 and 2025M781522, Zhejiang Key Laboratory of Decision Intelligence under No. 2025E10006, Zhejiang Provincial Natural Science Foundation Exploration of China under No. LMS26F020003, State Key Laboratory of Cryptography and Digital Economy Security under No. KFYB2504, the Zhejiang Provincial Natural Science Foundation under No. LD24F020002, the Fundamental Research Funds for the Central Universities+2025ZFJH02, , the Pioneer and Leading Goose R&D Program of Zhejiang under No. 2025C02033 and 2025C01082 , and the NGICS platform.

REFERENCES

[1] A. Decan, T. Mens, and M. Claes, "An empirical comparison of dependency issues in oss packaging ecosystems," in *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*, pp. 2–12, IEEE, 2017.

[2] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, vol. 24, no. 1, pp. 381–416, 2019.

[3] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?," *Empirical Softw. Engg.*, vol. 23, p. 384–417, Feb. 2018.

[4] B. Chinthanet, R. G. Kula, S. McIntosh, T. Ishio, A. Ihara, and K. Matsumoto, "Lags in the release, adoption, and propagation of npm vulnerability fixes," *Empirical Softw. Engg.*, vol. 26, May 2021.

[5] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?," *Empirical Softw. Eng.*, vol. 23, p. 384–417, Feb. 2018.

[6] D. Jayasuriya, "Towards automated updates of software dependencies," in *Companion Proceedings of the 2022 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, SPLASH Companion 2022, (New York, NY, USA), p. 29–33, Association for Computing Machinery, 2022.

[7] A. Dann, B. Hermann, and E. Bodden, "Upcy: Safely updating outdated dependencies," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 233–244, IEEE, 2023.

[8] "Dependabot.." https://docs.github.com/en/code-security/dependabot.

[9] "Renovate.." https://www.mend.io/renovate.

[10] A. Brito, L. Xavier, A. Hora, and M. T. Valente, "Apidiff: Detecting api breaking changes," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 507–511, IEEE, 2018.

[11] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 4, pp. 1–35, 2011.

[12] L. Li, T. F. Bissyandé, H. Wang, and J. Klein, "Cid: Automating the detection of api-related compatibility issues in android apps," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 153–163, 2018.

[13] M. Lamothe, W. Shang, and T.-H. Chen, "A4: Automatically assisting android api migrations using code examples," *CoRR*, 2018.

[14] B. B. Nielsen, M. T. Torp, and A. Møller, "Semantic patches for adaptation of javascript programs to evolving libraries," in *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, p. 74–85, IEEE Press, 2021.

[15] L. Fruntke and J. Krinke, "Automatically fixing dependency breaking changes," *Proc. ACM Softw. Eng.*, vol. 2, June 2025.

[16] F. Reyes Garcia, M. Mahmoud, F. Bono, S. Nadi, B. Baudry, and M. Monperrus, "Byam: Fixing breaking dependency updates with large language models," 05 2025.

[17] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of javascript programs," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, (New York, NY, USA), p. 1–12, Association for Computing Machinery, 2010.

[18] "Async: 'breaking: remove during, make test functions in until/whilst async.'." https://github.com/caolan/async/pull/1557.

[19] "Online appendix." https://github.com/EvanXiaa/BDUpdater.

[20] "Semantic versioning 2.0.0." https://semver.org/.

[21] M. Lamothe, Y.-G. Guéhéneuc, and W. Shang, "A systematic review of api evolution literature," *ACM Computing Surveys (CSUR)*, vol. 54, no. 8, pp. 1–36, 2021.

[22] Y. Wang, P. Sun, L. Pei, Y. Yu, C. Xu, S.-C. Cheung, H. Yu, and Z. Zhu, " Plumber: Boosting the Propagation of Vulnerability Fixes in the npm Ecosystem ," *IEEE Transactions on Software Engineering*, vol. 49, pp. 3155–3181, May 2023.

[23] "npm. node package manager." https://www.npmjs.com/.

[24] Github advisary. https://github.com/advisories.

[25] Z. Xing and E. Stroulia, "Api-evolution support with diff-catchup," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 818–836, 2007.

[26] "Openai, gpt-4.1." https://platform.openai.com/docs/models/gpt-4.1.

[27] D. Venturini, F. R. Cogo, I. Polato, M. A. Gerosa, and I. S. Wiese, "I depended on you and you broke me: An empirical study of manifesting breaking changes in client packages," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, May 2023.

[28] D. Kong, J. Liu, L. Bao, and D. Lo, "Toward better comprehension of breaking changes in the npm ecosystem," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, Apr. 2025.

[29] A. Decan and T. Mens, "What do package dependencies tell us about semantic versioning?," *IEEE Transactions on Software Engineering*, vol. 47, no. 6, pp. 1226–1240, 2021.

[30] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "How to break an api: cost negotiation and community values in three software ecosystems," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of*

Software Engineering, FSE 2016, (New York, NY, USA), p. 109–120, Association for Computing Machinery, 2016.
[31] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "When and how to make breaking changes: Policies and practices in 18 open source software ecosystems," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, July 2021.
[32] G. Mezzetti, A. Møller, and M. T. Torp, "Type regression testing to detect breaking changes in node.js libraries," in *European Conference on Object-Oriented Programming*, 2018.
[33] L. Zhang, C. Liu, Z. Xu, S. Chen, L. Fan, B. Chen, and Y. Liu, "Has my release disobeyed semantic versioning? static detection based on semantic differencing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, (New York, NY, USA), Association for Computing Machinery, 2023.
[34] A. Brito, M. T. Valente, L. Xavier, and A. Hora, "You broke my code: understanding the motivations for breaking changes in apis," *Empirical Software Engineering*, vol. 25, no. 2, pp. 1458–1492, 2020.
[35] A. Brito, L. Xavier, A. Hora, and M. T. Valente, "Why and how java developers break apis," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 255–265, IEEE, 2018.
[36] J. Hejderup and G. Gousios, "Can we trust tests to automate dependency updates? A case study of java projects," *CoRR*, vol. abs/2109.11921, 2021.
[37] S. Mujahid, R. Abdalkareem, E. Shihab, and S. McIntosh, "Using others' tests to identify breaking updates," in *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, (New York, NY, USA), p. 466–476, Association for Computing Machinery, 2020.
[38] D. Dig and R. Johnson, "How do apis evolve? a story of refactoring: Research articles," *J. Softw. Maint. Evol.*, vol. 18, p. 83–107, Mar. 2006.
[39] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, (New York, NY, USA), p. 92–101, Association for Computing Machinery, 2014.
[40] "Conventional commits.." https://www.conventionalcommits.org/.
[41] Y. Tian, Y. Zhang, K.-J. Stol, L. Jiang, and H. Liu, "What makes a good commit message?," in *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, (New York, NY, USA), p. 2389–2401, Association for Computing Machinery, 2022.
[42] D. S. Cruzes and T. Dyba, "Recommended steps for thematic synthesis in software engineering," in *2011 international symposium on empirical software engineering and measurement*, pp. 275–284, IEEE, 2011.
[43] A. Møller, B. B. Nielsen, and M. T. Torp, "Detecting locations in javascript programs affected by breaking library changes," *Proc. ACM Program. Lang.*, vol. 4, Nov. 2020.
[44] "Libraries.io - security & maintenance data for open source." https://libraries.io/.
[45] "Lodash. a modern javascript utility library delivering modularity, performance & extras.." https://lodash.com/.
[46] "Anthropic." https://www.anthropic.com/.
[47] "Deepseek. into the unknown." https://www.deepseek.com/en.
[48] "Qwen." https://qwen.ai/home.
[49] Y.-A. Xiao, P. Gao, C. Peng, and Y. Xiong, "Improving the efficiency of llm agent systems through trajectory reduction," 2025.
[50] L. Xavier, A. Hora, and M. T. Valente, "Why do we break apis? first answers from developers," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 392–396, IEEE, 2017.
[51] J. Dietrich, D. Pearce, J. Stringer, A. Tahir, and K. Blincoe, "Dependency versioning in the wild," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 349–359, IEEE, 2019.
[52] L. Ochoa, T. Degueule, J.-R. Falleri, and J. Vinju, "Breaking bad? semantic versioning and impact of breaking changes in maven central: An external and differentiated replication study," *Empirical Software Engineering*, vol. 27, no. 3, p. 61, 2022.
[53] D. Venturini, F. R. Cogo, I. Polato, M. A. Gerosa, and I. S. Wiese, "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*, vol. 32, no. 4, pp. 1–26, 2023.
[54] E. Larios Vargas, M. Aniche, C. Treude, M. Bruntink, and G. Gousios, "Selecting third-party libraries: The practitioners' perspective," in *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pp. 245–256, 2020.
[55] P. Salza, F. Palomba, D. Di Nucci, C. D'Uva, A. De Lucia, and F. Ferrucci, "Do developers update third-party libraries in mobile apps?," in *Proceedings of the 26th conference on program comprehension*, pp. 255–265, 2018.
[56] S. Mujahid, R. Abdalkareem, E. Shihab, and S. McIntosh, "Using others' tests to identify breaking updates," in *Proceedings of the 17th international conference on mining software repositories*, pp. 466–476, 2020.
[57] Y. Padioleau, R. R. Hansen, J. L. Lawall, and G. Muller, "Semantic patches for documenting and automating collateral evolutions in linux device drivers," in *Proceedings of the 3rd workshop on Programming languages and operating systems: linguistic support for modern operating systems*, pp. 10–es, 2006.

[58] Y. Padioleau, J. L. Lawall, and G. Muller, "Smpl: A domain-specific language for specifying collateral evolutions in linux device drivers," *Electronic Notes in Theoretical Computer Science*, vol. 166, pp. 47–62, 2007.

[59] H. J. Kang, F. Thung, J. L. Lawall, G. Muller, L. Jiang, and D. Lo, "Semantic patches for java program transformation," in *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, vol. 134, pp. 22–1, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

[60] Z. Zhang, H. Zhu, M. Wen, Y. Tao, Y. Liu, and Y. Xiong, "How do python framework apis evolve? an exploratory study," in *2020 ieee 27th international conference on software analysis, evolution and reengineering (saner)*, pp. 81–92, IEEE, 2020.

[61] D. Kong, J. Liu, C. Ni, D. Lo, and L. Bao, "More effective javascript breaking change detection via dynamic object relation graph," *Proc. ACM Softw. Eng.*, vol. 2, June 2025.

[62] A. Møller and M. T. Torp, "Model-based testing of breaking changes in node. js libraries," in *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pp. 409–419, 2019.

[63] R. Shariffdeen, X. Gao, G. J. Duck, S. H. Tan, J. Lawall, and A. Roychoudhury, "Automated patch backporting in linux (experience paper)," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, (New York, NY, USA), p. 633–645, Association for Computing Machinery, 2021.

[64] S. Yang, Y. Xiao, Z. Xu, C. Sun, C. Ji, and Y. Zhang, "Enhancing oss patch backporting with semantics," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, (New York, NY, USA), p. 2366–2380, Association for Computing Machinery, 2023.

[65] S. Pan, Y. Wang, Z. Liu, X. Hu, X. Xia, and S. Li, "Automating zero-shot patch porting for hard forks," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, (New York, NY, USA), p. 363–375, Association for Computing Machinery, 2024.

[66] S. Wu, R. Wang, Y. Cao, B. Chen, Z. Zhou, Y. Huang, J. Zhao, and X. Peng, "Mystique: Automated vulnerability patch porting with semantic and syntactic-enhanced llm," *Proc. ACM Softw. Eng.*, vol. 2, June 2025.