Dependency Version Migration in Software Projects: A Rapid Review of Motivations, Techniques, Tools, and Challenges

Arthur V. B. da Silva
Instituto Federal de Pernambuco
Recife, PE, Brasil
arthurvini2703@gmail.com

Leopoldo Teixeira
Centro de Informática
Universidade Federal de Pernambuco
Recife, PE, Brasil
lmt@cin.ufpe.br

Wesley K. G. Assunção
North Carolina State University
Raleigh, NC, United States of America
wguezas@ncsu.edu

Bruno Cartaxo
Zup Innovation
São Paulo, SP, Brazil
Instituto Federal de Pernambuco
Paulista, PE, Brasil
email@brunocartaxo.com

Abstract

Software projects increasingly rely on third-party libraries and frameworks that evolve independently of client applications. When dependencies release new versions, client projects must decide whether and how to migrate, a decision that may require adapting client code to breaking changes, altered APIs, or shifted behaviors. Despite the practical importance of this activity, existing research on dependency version migration is scattered across ecosystems, motivations, and support approaches, with no prior synthesis consolidating what is known across all these dimensions. This paper presents a rapid review characterizing the state of research on dependency version migration. Guided by seven research questions, we retrieved 2,637 candidate papers. After applying predefined inclusion and exclusion criteria through a three-phase screening process, we selected 63 primary studies for data extraction and synthesis. The results show that dependency migration is not yet a widespread practice: studies consistently report that most projects rely on outdated dependency versions, and migration tends to be reactive rather than proactive. Where motivations are reported, bug fixes, compatibility requirements, and security vulnerabilities are the primary drivers. However, these findings derive from a limited number of primary studies and should be interpreted with caution. Breaking change detection relies predominantly on automated tooling, while identification of affected client code typically combines automated analysis with manual inspection. The broader repertoire of migration support techniques spans static and structural analysis, rule-based transformations, and AI-driven or example-based approaches. Each category presents recurring trade-offs between precision, ecosystem generalizability, and required developer effort. Across the 63 studies, no single technique demonstrates broad effectiveness across diverse migration scenarios. These findings reveal a fragmented research area characterized by ecosystem-specific solutions, limited cross-study comparability, and sparse empirical evaluation under realistic developer conditions.

Keywords

Library Migration, Code Migration Tools, Dependency Management, Software Evolution, Migration Challenges

1 Introduction

Modern software development heavily depends on third-party libraries, frameworks, and APIs to accelerate development, encourage reuse, and improve maintainability [10]. However, these dependencies evolve independently of the client projects that use them, and new versions often introduce breaking changes that require developers to adapt their code to maintain functionality [6]. In a large-scale study of the Maven repository, Raemaekers et al. [9] found that breaking changes occur even when library maintainers follow semantic versioning, meaning that dependency updates carry non-trivial adaptation risk regardless of versioning discipline.

Many systems continue to rely on outdated dependencies. Lauinger et al. [7] found that a substantial portion of websites include obsolete and vulnerable JavaScript libraries. While non-updating can be a rational decision, driven by concerns about unwanted new features, incompatible toolchain versions, or increased resource requirements, it can also expose systems to known vulnerabilities, indicating that dependency version migration is not a consistently adopted practice. This tension between the potential benefits of migrating and the costs or risks of doing so motivates a need to understand how migration decisions are made, how breaking changes are detected, and what tools and techniques exist to support the process.

The scale of this problem is substantial. Kula et al. [S29] found that 81.5% of Java systems on GitHub still depended on outdated libraries even after newer versions had been available for extended periods. Jayasuriya et al. [S61] reported that 71.6% of Maven artifact dependencies were not up-to-date, while Salza et al. [S30] observed that in 63% of Android app libraries, developers never updated a dependency after its initial introduction. When asked why, developers cite the cost of adaptation and the risk of introducing new failures as the primary deterrents [S58]. Venturini et al. [S37] showed that breaking changes do manifest in client packages in practice, causing test failures and requiring non-trivial code adaptation. Taken together, this evidence indicates that dependency version migration is both a widespread maintenance burden and a consistently deferred one — a combination that points to a gap between the

tools and techniques the research community has proposed and the support that developers are actually using.

Prior work has examined adjacent aspects of this problem in isolation: studies on API evolution, library update behavior, and automated refactoring have each contributed partial evidence. However, no existing synthesis consolidates what is known about dependency version migration as a whole, spanning motivations, detection strategies, support techniques, and their reported trade-offs. This gap in consolidated evidence is the starting point for the present study.

In this paper, we present a rapid review of the literature on dependency version migration, following the methodology of Cartaxo et al. [3, 4]. Rapid reviews offer a structured and time-efficient approach to evidence synthesis, particularly suited to contexts where a broad map of a research area is needed to support decision-making [3]. We searched Scopus in February 2025, retrieved 2,637 candidate papers, and selected 63 primary studies through a three-phase screening process guided by predefined inclusion and exclusion criteria. The study is organized around seven research questions covering migration adoption and motivations (RQ1–RQ2), strategies for identifying outdated dependencies and breaking changes (RQ3–RQ4), and the tools, techniques, and trade-offs that characterize existing migration support (RQ5–RQ7).

The main findings of this review can be summarized as follows. Dependency version migration is predominantly reactive: Developers migrate in response to concrete failures — security vulnerabilities, compatibility breakages, and bugs — rather than as a proactive maintenance practice, and the majority of projects in every studied ecosystem use outdated dependency versions at any given time. The set of available tools is large but fragmented: 45 distinct tools were identified across the 63 studies, yet virtually all are ecosystem-specific, with no tool demonstrating broad effectiveness across diverse migration scenarios. The underlying technique families — static and structural analysis, rule-based transformation, AI-driven and example-based approaches, and semantic analysis — each occupy a distinct performance niche, and the trade-offs between them mean that no single approach is adequate for the diversity of real-world migration contexts. These findings reveal a research area where tool proposals outpace rigorous, comparable evaluation, and where the developer-facing dimensions of migration support remain largely uninvestigated.

In summary, this paper makes the following contributions:

- Characterizes the extent to which developers migrate dependencies in practice and the motivations that drive such decisions (RQ1–RQ2);
- Surveys strategies used to identify dependencies that require updates and to detect version migrations that introduce breaking changes (RQ3–RQ4);
- Analyzes the tools proposed to support dependency version migration, the underlying techniques they employ, and the trade-offs reported in the literature (RQ5–RQ7).

The remainder of this paper is structured as follows. Section 2 reviews related work. Section 3 describes the review method. Section 4 presents the results for each research question. Section 5 discusses the implications of these findings. Finally, Section 6 concludes the paper and outlines directions for future work.

2 Related Work

Research on dependency evolution and API migration has grown substantially over the past decade, spanning empirical studies of developer behavior, proposals of automated support tools, and secondary studies synthesizing existing evidence. We organize the most relevant prior work into three groups and conclude by identifying the gap this study addresses.

Developer behavior and migration decisions. Several studies have investigated how often and why developers update their dependencies in practice. Kula et al. [S29] analyzed the influence of security advisories on library migration decisions, finding that many developers continue to rely on outdated versions despite known vulnerabilities, primarily due to concerns about migration effort and compatibility risk. In the context of mobile development, Salza et al. [S30] observed that developers frequently postpone library updates, attributing this behavior to uncertainty about breaking changes introduced in newer versions. These studies establish that migration avoidance is widespread, but each focuses on a specific ecosystem or trigger, leaving the broader picture of migration behavior across ecosystems uncharacterized.

Tools and techniques for automated migration. Other work has focused on building and evaluating tools to support automated or semi-automated migration. Dig et al. [S63] proposed using refactorings to automate component updates, presenting tools capable of transforming client code based on API change logs. Hora and Robbes [S13] surveyed techniques for inferring API changes, cataloguing static and dynamic analysis approaches and discussing their respective limitations. Bavota et al. [S32] reported on the practical use of automated migration tools in the Android ecosystem, identifying integration challenges and trade-offs between automation and developer effort. While these contributions are directly relevant, each addresses a specific technique, family, or ecosystem rather than providing a cross-cutting characterization of the available tools.

Secondary studies on software modernization. At the secondary study level, Althani and Khaddaj [1] conducted a systematic review of legacy system migration, identifying challenges related to data, architecture, and interface transformation. Assunção et al. [2] surveyed contemporary software modernization strategies, categorizing the driving forces behind modernization and highlighting the challenges of integrating new dependencies into existing systems. Both studies provide relevant context, but their scope is system-level modernization rather than the specific activity of migrating client code to newer versions of existing dependencies. Neither characterizes the detection strategies, tool techniques, or migration trade-offs that are central to dependency version migration as a development practice.

Rapid reviews in software engineering. Our study follows the rapid review methodology proposed by Cartaxo et al. [4], which offers a structured and time-efficient alternative to full systematic literature reviews, particularly suited to contexts where timely evidence synthesis is needed. Cartaxo et al. [3] have argued that, when carefully planned, rapid reviews can provide relevant insights without compromising methodological rigor. Their complementary proposal of

evidence briefings [5] further supports the transfer of synthesized findings to practitioners. These methodological contributions underpin both the feasibility and the design choices of the present study.

The gap this study addresses. Taken together, the studies above each contribute partial evidence on dependency version migration, but none spans the full activity, from motivations through detection strategies to migration support techniques and their trade-offs. This paper addresses that gap through a rapid review consolidating evidence across all these dimensions.

3 Research Method

We performed a rapid review following the guidelines of Cartaxo et al. [3, 4] to provide timely evidence to support decision-making in a real-world software development context.

3.1 Practical Problem

As highlighted by Cartaxo et al. [4], rapid reviews should be performed in close collaboration with practitioners, bounded to practical problems, and conducted within the practitioner's context.

In that regard, this rapid review was conducted in collaboration with a co-author who is an industry practitioner from a company that develops software tools for developers. The practitioner expressed interest in understanding how software projects manage dependency version migration, particularly the trade-offs among tools and their underlying techniques for automating or semi-automating client code transformations to address breaking changes caused by dependency updates. He participated in the definition of the review protocol and selection criteria, and validated the key methodological decisions throughout the process.

3.2 Scope Definition

For the purposes of this review, we distinguish *dependency updates* — changes to the version number of a declared dependency that require no client-side code adaptation — from *dependency version migration*, which encompasses updates that require developers to adapt client code, configurations, or tests due to breaking changes or altered behaviors. This review focuses on the latter; studies addressing pure version-number updates without an adaptation dimension were excluded under EC4. Automated dependency update tools such as Dependabot, which primarily issue version-bump pull requests without addressing client-side breaking changes, similarly fall outside the scope of this review.

3.3 Research Questions

The following research questions guided the search strategy, selection of primary studies, and data extraction:

- **RQ1: Do developers commonly migrate their dependencies to newer versions?**
  This question investigates whether dependency migration is a frequent practice in real-world software development.
- **RQ2: What motivates developers to migrate dependencies to newer versions?**
  This question explores the common drivers for updating dependencies.
- **RQ3: What are the strategies used to identify dependencies that need to be updated?**
  This question focuses on the strategies and tools used to locate deprecated or outdated dependency usages in a codebase.
- **RQ4: What are the strategies used to identify dependency version migrations that introduce breaking changes?**
  This question examines the mechanisms employed to identify and understand breaking changes, whether through manual inspection or automated tooling.
- **RQ5: What tools have been proposed to support dependency version migration?**
  This question surveys existing tools that aid the migration process, with particular focus on tools that automate or semi-automate client code adaptation when migrating to newer dependency versions.
- **RQ6: What are the underlying techniques used by dependency version migration tools?**
  This question categorizes and analyzes the techniques adopted by migration tools (e.g., AI-driven, rule-based, static analysis, semantic analysis).
- **RQ7: What are the trade-offs of existing tools and techniques?**
  This question examines the limitations, performance concerns, required human effort, and generalization issues reported for current approaches.

By addressing these questions, we aim to synthesize and characterize the current state of research and practice in dependency version migration.

3.4 Search Strategy

We used Scopus[1] as the primary database for our literature search, given its broad coverage of peer-reviewed computer science publications [8]. The choice of a single database is a scope limitation of this rapid review; studies indexed exclusively in ACM Digital Library or IEEE Xplore may not have been captured. We acknowledge this as a threat to completeness and discuss it further in Section 5.

On February 27, 2025, we executed a structured search query targeting papers published from 2005 onwards. The search string was constructed in two parts connected by the Scopus proximity operator `W/0`, which requires both terms to appear within the same field token — effectively enforcing that a dependency-related term and a change-related term co-occur in direct association rather than appearing independently anywhere in the title, abstract, or keywords. This choice was deliberate: a simple `AND` between the two groups would have retrieved a large number of papers that mention, for example, both a library and a refactoring in unrelated contexts. The first term group targets the artifact being migrated (*library*, *API*, *framework*, *dependency*); the second targets the activity (*migration*, *update*, *change*, *deprecation*, *evolution*, *refactoring*, *versioning*). Results were further filtered to the `COMP` subject area to exclude publications in non-computing disciplines that share this vocabulary (e.g., biology and chemistry, where *evolution* and *migration* have entirely different meanings). Gray literature was excluded, as the goal was to complement existing practitioner knowledge

[1] https://www.scopus.com

with evidence from the academic research literature. The query below returned a total of **2,637** candidate papers.

> ***Search String:*** *TITLE-ABS-KEY ((librar* OR api OR framework OR dependenc*) W/0 (migrat* OR updat* OR chang* OR deprecat* OR evol* OR refact* OR version*)) AND PUBYEAR >2004 AND PUBYEAR <2026 AND (LIMIT-TO (PUBSTAGE , "final")) AND (LIMIT-TO (SUBJAREA , "COMP"))*

3.5 Exclusion Criteria

Studies were considered eligible if they were peer-reviewed publications written in English, published between 2005 and 2025, and provided evidence relevant to at least one of the seven research questions on dependency version migration, addressing migration between versions of the same dependency. Papers were then excluded if they:

EC1 were not written in English;
EC2 were not peer-reviewed;
EC3 were published before 2005;
EC4 did not provide relevant insights to any of the seven research questions;
EC5 focused exclusively on migration between different dependencies (e.g., replacing one library with another) rather than version migration of the same dependency.

EC4 and EC5 required the most interpretive judgment during screening and deserve illustration. Under EC4, papers were excluded when they addressed API or library evolution without connecting their findings to any of the seven RQs. For example, studies characterizing how APIs change across versions without addressing detection strategies, migration support, or developer behavior — such as empirical analyses of API change frequency or stability — were excluded because they describe the phenomenon rather than the response to it. Papers proposing general refactoring tools that treat library changes only as incidental examples, and secondary studies reviewing API evolution broadly without synthesizing evidence on migration specifically, were similarly excluded under EC4. Under EC5, papers were excluded when their primary contribution was supporting *cross-library* migration, that is, helping developers replace one library with a functionally equivalent alternative. For example, studies proposing tools to detect that a project should migrate from one logging library to a different logging library, or mining techniques for recommending library replacements at the method level, were excluded because they address library substitution rather than the adaptation of client code to a new version of an existing dependency.

3.6 Screening Process

The exclusion criteria were applied through a three-phase filtering process, summarized in Figure 1. Prior to screening, all four reviewers held an initial calibration meeting to discuss the review scope, align on the interpretation of the exclusion criteria, and work through a representative set of borderline cases together. This calibration was intended to establish a shared understanding of the criteria before independent screening began. After each screening phase, reviewers reconvened — either synchronously or asynchronously — to discuss edge cases, resolve disagreements, and reach consensus on ambiguous decisions before proceeding to the next phase.

In the **first phase**, the four authors independently conducted title-based screening of the 2,637 candidate papers, which were divided into four approximately equal subsets, one per reviewer. Consistent with rapid review guidelines [4], each paper was assessed by a single reviewer. Papers were labeled *In*, *Out*, or *Maybe*, where *Maybe* indicated uncertainty requiring further inspection. Papers marked *Maybe* were retained and carried forward to the next screening phase, where they received a definitive *In* or *Out* decision. This phase reduced the corpus to 438 papers (the union of *In* and *Maybe* decisions). The large reduction from 2,637 to 438 papers after title screening reflects the breadth of the wildcard terms used (e.g., `chang*`, `evol*`, `version*`), which, despite the precision-oriented proximity operator, still matched many papers addressing unrelated software engineering topics such as general software evolution, data migration, or refactoring techniques unrelated to dependency management. Such papers were straightforward exclusions at the title level.

In the **second phase**, the same four reviewers independently screened the abstracts of the 438 remaining papers, again without cross-checking between reviewers, and applying the same three labels. This phase reduced the corpus to 212 papers.

In the **third phase**, the full text of the 212 remaining papers was read to make final inclusion decisions using only *In* or *Out* labels. This phase was conducted by the first author.[2] This phase yielded the final set of **63 primary studies** for data extraction. The spreadsheets documenting all screening decisions are publicly available, as mentioned in the Artifact Availability section.

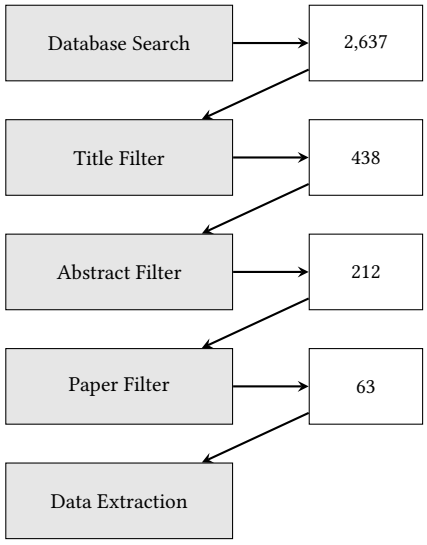

Figure 1: Three-phase paper selection process.

[2] Single-reviewer full-text screening is consistent with the rapid review methodology adopted here [4], but introduces a risk of selection bias that we acknowledge as a threat to validity in Section 5.

3.7 Data Extraction

In parallel with the third screening phase, we designed a data extraction form to collect relevant information from each included study. For each of the 63 selected papers, we recorded the publication year, and mapped each paper to the specific research questions it addressed. This RQ mapping was the primary organizing structure for data extraction: For each RQ, we recorded the evidence provided by each relevant study, including the study design (e.g., tool proposal, empirical analysis, or experience report), the ecosystem or programming language targeted, and the specific findings pertaining to that RQ. A dedicated spreadsheet was used to track the mapping between studies and RQs, facilitating the synthesis of evidence across studies and organizing the results by question.

The mapping of papers to RQs required interpretive judgment, particularly for studies that addressed multiple questions simultaneously or that provided only partial evidence for a given RQ. In such cases, a paper was mapped to an RQ if it contributed at least one concrete finding relevant to that question, even if the RQ was not the paper's primary focus. This decision was made consistently across all 63 studies using the same extraction form.

4 Results

This section presents the results of the data extraction process based on the 63 primary studies selected through the rapid review. For each of the seven research questions, we synthesize the findings across the relevant studies, highlighting patterns, tools, techniques, and trade-offs reported in the literature. The number of primary studies contributing evidence to each RQ varies considerably, reflecting the uneven distribution of research attention across the dimensions of dependency version migration. We note the corpus size for each RQ and qualify findings accordingly.

4.1 RQ1: Do developers commonly migrate their dependencies to newer versions?

A total of 6 studies directly addressed whether developers migrate to newer dependency versions [S29, S30, S35, S57, S58, S61]. The findings consistently indicate that migration to newer versions is not a common practice.

Kula et al. [S29] found that 81.5% of the systems they studied still relied on outdated dependencies. Using the Library Residue metric — which measures the proportion of systems still using an outdated library version after its peak usage — the authors analyzed 2,736 dependencies across 48,495 Java systems on GitHub and reported an average residue ratio of 81.5%.

Salza et al. [S57] observed that only 15.5% of library uses were consistently updated by developers. The authors analyzed 11,626 version histories of Android app libraries and classified update practices using open coding with four researchers. The pattern of *diligent updates* — where developers consistently update libraries to the latest version in each development cycle — was found in only 1,976 library usages (15.5%).

In a complementary study, Salza et al. [S30] examined version histories of 1,126 libraries in 291 Android apps and classified developer behaviors into five categories. The *Not Updating* pattern, where libraries were never updated after their initial introduction, was found in 63.0% of cases.

Jayasuriya et al. [S61] analyzed 18,415 Maven artifacts declaring 142,355 direct dependencies and found that 71.6% were not up-to-date with the latest available version.

A survey by Zaitsev et al. [S35] with 36 developers confirmed that updates occur, but rarely: Three-quarters of respondents updated dependency versions at least twice a year, and fewer than half did so three or more times a year.

Finally, Sawant et al. [S58] asked developers why they did not upgrade their dependencies and identified two primary reasons: the cost of upgrading was often not considered worthwhile, and the version in use was still functioning adequately, reducing the perceived urgency to update.

> ***RQ1 Key Takeaway:*** *Across six studies and multiple ecosystems, the evidence consistently shows that developers rarely update their dependencies proactively. Most projects rely on outdated dependency versions, and migration tends to be deferred unless a concrete need arises.*

4.2 RQ2: What motivates developers to migrate dependencies to newer versions?

Only 2 primary studies directly addressed developer motivations for migrating dependencies [S57, S62]. Given this limited evidence base, the findings reported here should be treated as preliminary indicators rather than definitive conclusions.

Salza et al. [S57] surveyed 73 mobile developers and found that avoiding bug propagation and ensuring compatibility with new Android releases were the primary reasons for updating dependencies.

Yasumatsu et al. [S62] analyzed 21,046 Android apps and found that a targeted security fix campaign was effective in encouraging developers to adopt library version updates, highlighting security concerns as a concrete migration driver.

> ***RQ2 Key Takeaway:*** *The available evidence — drawn from only 2 primary studies, both in the Android ecosystem — suggests that the primary motivations for dependency migration are bug fixes, compatibility requirements, and security vulnerabilities. These findings should be interpreted with caution, as the narrow evidence base and ecosystem-specific context limit their generalizability.*

4.3 RQ3: What are the strategies to identify dependencies that need to be updated?

Only 2 primary studies focused specifically on strategies to identify dependencies that require updating [S01, S06], each proposing a distinct automated approach. As with RQ2, the small number of studies limits the conclusions that can be drawn.

Kumar et al. [S01] proposed VODA (Vulnerable Open-Source Dependency Analyzer), a tool that simultaneously searches thousands of open-source repositories for dependencies with known vulnerabilities, considering both previous and current releases.

Navarro et al. [S06] proposed an automated method to source API deprecation data for popular Python libraries by crawling and parsing release notes from the web. The strategy extracts libraries using Sphinx for documentation, retrieves version histories from

the PyPI API, and parses deprecation notices from the standardized HTML structure of the documentation.

RQ3 Key Takeaway: *The limited evidence available (2 studies) suggests that proposed strategies for identifying dependencies requiring updates are fully automated, targeting either known security vulnerabilities or documented deprecations. Both approaches are ecosystem-specific — one targets open-source vulnerability data broadly, the other targets Python libraries specifically — indicating that no general-purpose identification strategy has yet been characterized in the literature. This represents a clear gap for future research.*

4.4 RQ4: What are the strategies to identify dependency version migrations that introduce breaking changes?

A total of 8 studies addressed strategies for identifying breaking changes introduced by dependency version migrations. Among them, 6 studies [S05, S10, S21, S34, S36, S61] employed automated strategies using or proposing dedicated tools, while 2 studies [S08, S47] relied on manual strategies.

Automated strategies typically involve comparing two versions of a dependency and applying algorithms to identify potential breaking changes.

Du and Ma [S05] introduced AexPy, a tool that detects API breaking changes in Python packages. Their evaluation demonstrated high precision in identifying both documented and undocumented breaking changes across 43 real-world packages.

Brito et al. [S10] developed APIDIFF, which detects API breaking and non-breaking changes by analyzing three API elements: types, methods, and fields.

Reyes et al. [S21] proposed BreakingGood, which identifies breaking changes using three inputs — the source code, the original dependency version, and the new dependency version — and provides an explanation for each detected issue.

Zhang et al. [S34] presented PyCombat, which performs two-phase static analysis: first extracting an API knowledge base, then detecting compatibility issues against it.

Serbout and Pautasso [S36] applied oasdiff, a command-line tool designed to compare OpenAPI specifications and detect changes between versions.

Jayasuriya et al. [S61] used jacimp, which detects breaking changes between two library versions by running static analysis on the differences between two JAR files.

Manual strategies typically involve updating the dependency version and evaluating whether the system continues to function correctly, either by executing it or running the existing test suite.

Jayasuriya et al. [S08] adopted a manual approach in which they updated the dependency version and verified whether the code continued to work as expected; failures were taken as evidence of a breaking change.

Venturini et al. [S47] applied a similar strategy by updating the dependency, installing the new version, and running the project's test suite; test failures indicated the presence of a breaking change.

RQ4 Key Takeaway: *Strategies for identifying breaking changes fall into two categories. Automated strategies use specialized tools to compare dependency versions and detect incompatibilities algorithmically; manual strategies rely on updating the dependency and observing failures through execution or testing. Automated approaches dominate the literature and offer scalability, but manual strategies remain in use, particularly in empirical studies evaluating migration impact in practice.*

4.5 RQ5: What tools have been proposed to support dependency version migration?

Among the 63 papers selected for this review, 48 mention at least one automated or semi-automated tool related to dependency version migration, totaling 45 distinct tools. Table 1 provides a structured overview of the 38 tools for which a name or clear identity could be established, organized by target ecosystem. The remaining tools are either unnamed approaches described within a single paper or instantiations of general frameworks. They are included in the technique analysis of RQ6 but are not individually listed here.

Three observations emerge directly from Table 1. First, the set of tools is heavily ecosystem-specific: Java (13 tools) and Android (7 tools) together account for more than half of the named tools, while JavaScript, Pharo, Swift, and REST/OpenAPI are each served by only one or two tools. This concentration is not simply a reflection of ecosystem popularity — it reflects a structural fragmentation in which tool contributions rarely generalize beyond their original target environment. Second, migration support tools (27 of 38) substantially outnumber detection-only tools (11 of 38), suggesting that the research community has invested more in automating code adaptation than in automating the identification of when adaptation is needed. Third, the technique distribution mirrors the findings of RQ6: Static and Structural Analysis is the most common primary technique (15 tools), followed closely by AI and Example-Driven approaches (14 tools), with Rule and Pattern-Based (6 tools) and Semantic and Contextual techniques (3 tools) less frequent.

Most tools were proposed within the papers themselves; only three studies analyze pre-existing tools [S13, S29, S63]. Several tools appear in more than one paper, reflecting ongoing development and evaluation across publications: AndroEvolve [S08, S09], AppEvolve [S14, S15], and MLCatchUp [S23, S42] each appear in two studies.

A notable line of incremental development exists within the Android ecosystem. AppEvolve was developed first as a solution for Android API evolution. CocciEvolve was subsequently inspired by it. AndroEvolve was later proposed as a further refinement, adapting and extending the techniques of its predecessor. This progression illustrates how research on migration support can accumulate within a single ecosystem, while leaving other ecosystems underserved.

Several tools are designed as IDE plugins, embedding migration assistance directly into developer workflows: HiMa [S03], CatchUp! [S22], and Trident [S49] target Eclipse, while the Python Package Tool [S06] targets PyCharm. This subset represents an attempt to reduce the adoption friction that is often cited as a barrier to tool use in practice, though no study in the corpus evaluates whether IDE integration actually improves adoption rates.

Table 1: Named tools identified across the 63 primary studies, organized by target ecosystem. Technique categories correspond to those in Table 2. Purpose: Det. = detection of outdated dependencies or breaking changes; Mig. = client code migration support.

Tool	Source	Ecosystem	Primary Technique	Purpose
Android (7 tools)				
AppEvolve	[S14, S15]	Android	AI/Example-Driven	Mig.
CocciEvolve	[S17]	Android	AI/Example-Driven	Mig.
AndroEvolve	[S08, S09]	Android	Rule/Pattern-Based	Mig.
APIMigrator	[S12]	Android	AI/Example-Driven	Mig.
LIBBANDAID	[S18]	Android	Static/Structural	Mig.
ACRYL	[S26]	Android	AI/Example-Driven	Det.
NEAT	[S59]	Android	Rule/Pattern-Based	Mig.
Java (13 tools)				
LIBSYNC	[S02]	Java	AI/Example-Driven	Mig.
HiMa	[S03]	Java	AI/Example-Driven	Mig.
APIDIFF	[S10]	Java	Static/Structural	Det.
APIFix	[S11]	Java	AI/Example-Driven	Mig.
CatchUp!	[S22]	Java	Static/Structural	Mig.
ComeBack!	[S24]	Java	Rule/Pattern-Based	Mig.
ReBA	[S47]	Java	Static/Structural	Mig.
Trident	[S49]	Java	AI/Example-Driven	Mig.
RefactoringMiner 2.0	[S51]	Java	Static/Structural	Det.
SemDiff	[S55]	Java	Semantic/Contextual	Det.
RefactoringCrawler	[S16]	Java	Static/Structural	Mig.
BreakingGood	[S21]	Java/Multi	Static/Structural	Det.
jacimp	[S61]	Java	Static/Structural	Det.
Python (7 tools)				
AexPy	[S05]	Python	Static/Structural	Det.
PyCombat	[S34]	Python	Static/Structural	Det.
Python Pkg. Tool	[S06]	Python	AI/Example-Driven	Mig.
MLCatchUp	[S23, S42]	Python	AI/Example-Driven	Mig.
PYEVOLVE	[S46]	Python	Rule/Pattern-Based	Mig.
SOAR	[S56]	Python	AI/Example-Driven	Mig.
Jupyter Repair	[S52]	Python	AI/Example-Driven	Mig.
JavaScript (2 tools)				
JSFIX	[S53]	JavaScript	Static/Structural	Mig.
JS Detector	[S28]	JavaScript	Static/Structural	Det.
Language-specific — other (3 tools)				
DEPREWRITER	[S27]	Pharo	Semantic/Contextual	Mig.
Apodini Migrator	[S48]	Swift	Static/Structural	Mig.
oasdiff	[S36]	REST/OpenAPI	Static/Structural	Det.
Cross-ecosystem / General (6 tools)				
VODA	[S01]	Multi	Static/Structural	Det.
Polyglot Tool	[S04]	Multi	Rule/Pattern-Based	Mig.
Meditor	[S39]	Multi	AI/Example-Driven	Mig.
LLM Migrator	[S19]	Multi	AI/Example-Driven	Mig.
Two-Stage Patch	[S60]	Multi	Rule/Pattern-Based	Mig.
Compiler-Directed	[S25]	Multi	Semantic/Contextual	Mig.

Figure 2 shows the temporal distribution of studies contributing to RQ5, RQ6, and RQ7. Publication activity was sparse before 2019 and has remained at a consistent level of 3–6 papers per year since. The uptick beginning in 2019 coincides with the emergence of learning-based and example-driven migration tools (e.g., AndroEvolve, CocciEvolve), suggesting that renewed interest in this period was driven in part by the growing availability of large-scale mined migration examples as training data, rather than by continued growth in traditional static-analysis approaches. The subsequent plateau may reflect either a maturing research area or a partial shift toward venues not fully captured by our Scopus-based search (see Section 5).

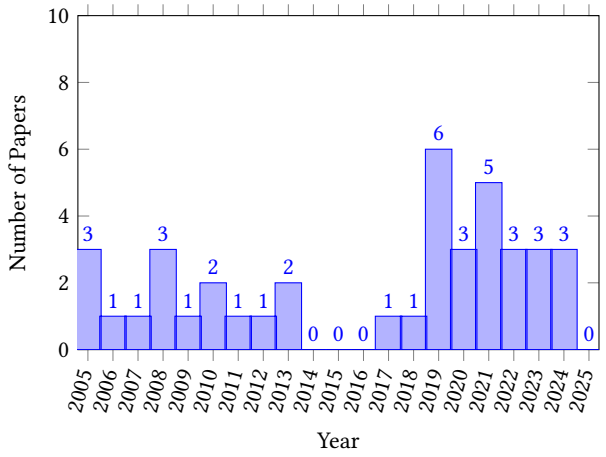

Figure 2: Distribution of papers by publication year for RQ5, RQ6, and RQ7.

> ***RQ5 Key Takeaway:*** *A large and diverse set of tools has been proposed to support dependency version migration, with 45 distinct tools identified across 48 of the 63 primary studies. However, the set of tools is heavily ecosystem-specific: Java and Android together account for more than half of named tools, and no tool demonstrates broad applicability across diverse ecosystems or migration scenarios. Migration support tools substantially outnumber detection tools, indicating a relative gap in research on identifying* when *migration is needed.*

4.6 RQ6: What are the underlying techniques used by these dependency version migration tools?

A total of 48 studies addressed the underlying techniques adopted by dependency version migration tools. After analyzing these papers, we grouped the underlying techniques into four major categories using a lightweight thematic analysis. Techniques were first extracted verbatim from each study's self-description (e.g., static analysis, rule-based transformation, example-based migration, semantic analysis, AI-based methods). We then performed an iterative coding process, analogous to open coding in qualitative research, grouping techniques by their operational characteristics (how the technique functions) and methodological focus (the underlying analytical principle). Through iterative discussion among the authors, four higher-level categories emerged, reported in Table 2. The goal was not to propose a new taxonomy, but to organize a diverse set of techniques into coherent, comparable groups.

AI and Example-Driven Techniques, found in 21 papers, include methods that use examples, learning, and artificial intelligence to automate migrations. These approaches typically extract patterns from real-world migrations, apply machine learning or program synthesis, or rely on large language models (LLMs) to generate transformation logic. They are especially effective in adapting to diverse migration contexts and handling less deterministic cases.

Rule and Pattern-Based Transformations, also represented in 21 papers, involve defining explicit rules or reusable patterns for migration. These rules may be specified manually, inferred from

examples, or synthesized automatically. They offer control and predictability in the transformation process, making them well-suited to structured and well-documented APIs.

Static and Structural Analysis, the most prominent group, appears in 29 papers. This category encompasses techniques that examine the structure and composition of code without requiring its execution. Their popularity reflects general applicability across languages and contexts, as well as scalability to large codebases.

Semantic and Contextual Analysis, identified in 10 studies, refers to techniques that analyze not only syntax but also the meaning and context of code. Although less frequent, these methods are valuable for understanding higher-level migration implications and avoiding behavioral regressions in complex scenarios.

Table 2: Techniques used by dependency version migration tools, grouped by category. Papers may contribute to more than one technique category; totals reflect technique occurrences rather than unique paper counts.

Technique/Approach	Tools (Papers)	Total
AI and Example-Driven Techniques		
	7 techniques, 21 papers	
Artificial Intelligence (AI)	[S06], [S19], [S42], [S52], [S56]	5
Data-Driven	[S26]	1
Example-Based	[S11], [S12], [S13], [S14], [S17], [S31], [S49]	7
History-Based Analysis	[S03]	1
Mining Software Repositories	[S02], [S44]	2
Program Synthesis	[S11]	1
Search-Based	[S26], [S39], [S52], [S56]	4
Rule and Pattern-Based Transformation		
	5 techniques, 21 papers	
Domain-Specific Language (DSL)	[S04]	1
Pattern-Based	[S08], [S28], [S40]	3
Refactoring-Based	[S24]	1
Rule-Based Transformation	[S04], [S11], [S12], [S13], [S20], [S38], [S41], [S43], [S46], [S59]	10
Semantic Patch Language (SmPL)	[S08], [S54], [S60]	3
Static and Structural Analysis		
	7 techniques, 29 papers	
AST-Based (Abstract Syntax Tree)	[S51]	1
Graph-based Analysis	[S02], [S04]	2
Impact Analysis	[S18]	1
Lightweight Static Analysis	[S04], [S20], [S22], [S28], [S49]	5
Static Analysis	[S06], [S11], [S12], [S13], [S16], [S39], [S42], [S47], [S48], [S50], [S53], [S54], [S55], [S59]	14
Transformation-Based	[S33]	1
Version Diff	[S13], [S23], [S51], [S60]	4
Semantic and Contextual Analysis		
	4 techniques, 10 papers	
Document Analysis	[S32]	1
Dynamic Analysis	[S13], [S25], [S27], [S52]	4
Heuristic-Based	[S32], [S55], [S59]	3
Method Deprecation	[S27]	1

RQ6 Key Takeaway: *Static and structural analysis is the most prevalent technique family in dependency version migration tools, appearing in 29 of the 48 relevant studies. AI-driven and example-based methods, and rule-based transformations, are equally represented (21 papers each), while semantic and contextual techniques are less frequent (10 papers). When examined at the individual technique level, static analysis, rule-based transformation, and example-based approaches are the three most commonly adopted.*

4.7 RQ7: What are the trade-offs of existing tools and techniques?

A total of 48 papers discussed trade-offs, benefits, and limitations related to tools or approaches supporting dependency version migration. These trade-offs are closely tied to the type of technique employed, as well as to design decisions, implementation strategies, and tool maturity.

AI and Example-Driven Techniques. Tools in this group — including LIBSYNC, APIFix, PyEvolve, and MLCatchUp — tend to offer high precision and recall, especially when trained or calibrated with concrete usage scenarios. However, they typically incur higher computational cost and depend on the availability of historical migration examples. Their specialization makes them powerful for complex or context-specific changes, but this often limits their generalization to new or underrepresented migration contexts.

Rule and Pattern-Based Transformations. Tools such as TAPIR, JaSCUT, DAAMT, and Coccinelle4j use explicit or inferred rules to automate migration steps. They are generally fast and accurate for well-documented, structured APIs, and their deterministic behavior makes them preferable in production settings where transparency is important. However, they require manual configuration or predefined rules, and their effectiveness diminishes for poorly documented or dynamically structured APIs.

Static and Structural Analysis. Tools in this group — including RefactoringMiner 2.0, SemDiff, ReBa, and RELANCER — are the most prevalent in the dataset. They operate on code structure without requiring execution, offering scalability and consistency across large codebases. Their main limitation is reduced effectiveness for complex, cross-layer structural changes or deeply semantic incompatibilities. Additionally, some tools (e.g., SemDiff) require complete version histories, reducing their flexibility in ad hoc migration scenarios.

Semantic and Contextual Analysis. Tools such as Apodini Migrator and JDiff attempt to capture meaning beyond code structure. While potentially effective for detecting higher-level impacts, they frequently require additional manual intervention and depend on external metadata such as change logs, semantic tags, and contextual annotations. Their adoption remains limited due to the complexity of integrating semantic understanding into automated workflows.

Taken together, the four technique categories present a spectrum of trade-offs. AI-driven and example-based tools offer automation and precision at the cost of computational resources and data requirements. Rule-based tools provide transparency and speed at the cost of flexibility. Static analysis tools are scalable but semantically shallow. Semantic tools are expressive but difficult to automate reliably.

RQ7 Key Takeaway: *Each category of dependency version migration techniques involves distinct trade-offs. No single technique family dominates across all migration scenarios: static analysis excels in scalability, rule-based approaches in transparency, AI-driven methods in precision for well-defined contexts, and semantic techniques in handling higher-level changes. Selecting an appropriate approach requires matching the technique's strengths to the specific characteristics of the migration context.*

5 Discussion

This section interprets the findings from the seven research questions, draws out their implications for research and practice, and analyzes the threats to the validity of the study.

5.1 Interpretation of Findings

Taken together, the results across RQ1–RQ7 paint a consistent picture: dependency version migration is a recognized maintenance need, yet it remains an infrequent, reactive, and largely unsupported activity in practice.

The adoption evidence from RQ1 is striking in its consistency across ecosystems and study designs. Whether measured through repository mining of Java systems [S29], version histories of Android apps [S30, S57], or Maven artifact analysis [S61], the result is the same: Most projects use outdated dependencies, and proactive update practices are rare. The motivations reported in RQ2, although based on limited evidence, are consistent with this pattern: Developers migrate primarily in response to concrete failures (bugs, incompatibilities, and security disclosures), rather than as a routine practice. This reactive posture suggests that the perceived cost of migration consistently outweighs its perceived benefit, a finding with direct implications for how tools and documentation should be designed.

The set of tools revealed by RQ5 is large but fragmented. Forty-five distinct tools appear across the 63 studies, yet no tool demonstrates broad applicability across ecosystems. The Android ecosystem accounts for a disproportionate share of proposals, while other ecosystems are served by isolated, language-specific solutions. This fragmentation is not merely a cataloguing observation: It means that a developer working outside a well-studied ecosystem (Java, Android, Python, JavaScript) is unlikely to find mature automated support for their migration tasks.

The technique taxonomy from RQ6 helps explain why fragmentation persists. Static and structural analysis — the dominant technique family — operates on code structure without execution, which gives it broad applicability and scalability, but limits its ability to handle semantically complex or cross-layer changes. AI-driven and example-based approaches can handle more complex cases, but depend on the availability of migration examples, which are often ecosystem- or tool-specific. The trade-offs reported in RQ7 thus reinforce the fragmentation finding: Each technique family has a domain where it performs well, but none generalizes reliably beyond that domain. The absence of a general-purpose migration technique, and the absence of cross-study benchmarks that would reveal this clearly, are the most important structural gaps identified by this review.

Viewed together, the findings reveal a reinforcing cycle. Developers migrate primarily in response to bugs, compatibility issues, and security concerns, yet the available migration support remains fragmented and ecosystem-specific. The trade-offs inherent in current techniques limit their general applicability, increasing the effort and uncertainty associated with dependency updates. As a result, migration is often postponed until external pressures make it unavoidable, further reinforcing the reactive maintenance behavior observed across ecosystems. A concrete illustration of this dynamic is the Android ecosystem's tooling concentration (RQ5): Android's mandatory API deprecation cycles create explicit, externally imposed migration triggers, which help explain both why tooling accumulated in this ecosystem first and why migration rates remain comparatively low in ecosystems where such triggers are absent.

5.2 Implications

For Researchers. This review identifies three specific directions where future research would have the greatest impact.

Understanding migration avoidance. RQ1 establishes that developers rarely migrate proactively, but the *mechanisms* behind this avoidance are undercharacterized. The cost-benefit reasoning reported by Sawant et al. [S58] points to developer decision-making as a key variable, yet no study in our corpus directly models this decision process across ecosystems or project types. Research combining behavioral studies with repository mining could establish when and why the perceived cost of migration is accurate versus systematically overestimated.

Benchmarking migration techniques comparably. The 45 tools identified in RQ5 have been evaluated in isolation, each using its own benchmarks, ecosystems, and metrics. This makes it impossible to compare their effectiveness directly. A shared benchmark dataset — analogous to those used in automated program repair or code summarization — would allow systematic comparison of static, rule-based, and AI-driven approaches under controlled conditions, and would reveal where each technique family genuinely falls short.

Evaluating tool integration in realistic workflows. RQ7 identifies usability and integration as recurring trade-offs, yet no study in our corpus evaluates migration tools within a realistic CI/CD or IDE-integrated workflow under controlled conditions. Research on developer-facing aspects of migration support — adoption barriers, cognitive load, and workflow fit — is almost entirely absent from the literature and represents a considerable opportunity.

For Practitioners. The findings suggest that developers and teams would benefit from treating dependency version migration as a structured, recurring maintenance activity rather than an emergency response. Reactive migration — triggered by security disclosures or breaking failures — is more costly and higher-risk than incremental, proactive updates. Automated tools can reduce the effort involved in identifying breaking changes and adapting client code, but practitioners should match tool selection to their specific context. Static analysis tools offer scalability for large codebases. Rule-based tools offer transparency and speed for well-documented APIs. AI-driven tools offer precision where training data is available.

Library and framework maintainers have a complementary role to play. Consistent adoption of semantic versioning, explicit documentation of breaking changes, and the provision of migration guides or codemods alongside new releases can substantially reduce the adaptation burden on client projects. The evidence from RQ2 suggests that security fix campaigns can be effective in driving adoption; similar structured communication strategies for breaking changes could reduce migration deferral.

5.3 Threats to Validity

We discuss threats to validity following standard categories for secondary studies.

Search completeness (external validity). The search was conducted exclusively in Scopus, executed in February 2025. Scopus provides broad coverage of peer-reviewed computer science publications [8] but does not index all venues covered by ACM Digital Library or IEEE Xplore, and given the rapid pace of development in LLM-based and agentic approaches to software engineering tasks, tools and techniques published since our search date may not be captured either. We acknowledge both as threats to the completeness of our corpus and to the generalizability of our findings on tool and technique coverage, limitations inherent to the rapid review design, which prioritizes timeliness over exhaustiveness, and to any review of a fast-moving research area.

Study selection bias (internal validity). The three-phase screening process was conducted without cross-reviewer validation at any phase, consistent with the rapid review methodology [4] but introducing a risk of systematic selection bias. In particular, full-text screening and all data extraction were performed by a single reviewer, which may have introduced inconsistency in the application of EC4 and EC5 — the two criteria requiring the most interpretive judgment. No inter-rater reliability statistic was computed. Findings from the full corpus of 63 studies should be interpreted with awareness that individual study inclusion decisions were not independently validated.

Thin evidence base for RQ2 and RQ3 (construct validity). The findings for RQ2 (motivations) and RQ3 (identification strategies) are drawn from only 2 primary studies each. Both RQ2 studies are specific to the Android mobile ecosystem. These findings should be treated as preliminary indicators rather than generalizable conclusions. The scarcity of primary studies directly addressing these dimensions is itself a finding — it indicates that developer motivation and dependency identification strategy are underresearched relative to tool and technique proposals.

Classification subjectivity (construct validity). The grouping of tools and techniques into four categories in RQ6 involved interpretive judgment, particularly for studies that combined multiple technique families. Different classification decisions could alter the relative frequency counts reported in Table 2. We mitigated this risk through a structured data extraction form applied consistently across all 63 studies, but the classification scheme was not externally validated.

Publication bias (external validity). As with any literature review restricted to peer-reviewed publications, there is a risk that negative results, failed tool evaluations, and inconclusive migration studies are underrepresented in the corpus. The exclusion of gray literature (EC2) further limits visibility into practitioner-facing tools and industrial migration experiences that may not appear in academic venues.

6 Conclusion

This paper presents a rapid review of the literature on dependency version migration, synthesizing evidence from 63 primary studies selected from 2,637 candidates retrieved from Scopus. Guided by seven research questions, the review characterized the current state of research across four dimensions: the motivations that drive migration decisions, the strategies used to detect outdated dependencies and breaking changes, the tools proposed to support the migration process, and the trade-offs those tools entail. As discussed in Section 5, the evidence points to a reinforcing cycle in which reactive migration behavior, a fragmented set of available tools, and technique-specific trade-offs sustain one another, and it surfaces concrete priorities for future work.

This review is subject to the limitations discussed in Section 5, most notably its reliance on a single database and the thin evidence base for RQ2 and RQ3. Within these boundaries, it provides a structured map of the dependency version migration research area, identifying where evidence is strong, where it is thin, and where it is absent, offering a foundation for researchers and practitioners seeking to understand or improve how software projects manage the evolution of their dependencies.

Three concrete directions follow: (1) empirical studies of migration decision-making and avoidance behavior; (2) shared benchmark datasets enabling comparable cross-tool evaluation; and (3) evaluation of migration tools under realistic developer workflows, including IDE and CI/CD integration.

Artifact Availability

In accordance with the SBES 2026 Open Science Policy, the artifacts produced by this study are archived on Zenodo under a CC BY 4.0 license: https://doi.org/10.5281/zenodo.22127847. The package contains the protocol, search exports, screening records, and data-extraction forms produced during the review. The spreadsheets documenting all inclusion and exclusion decisions across the three screening phases for all 2,637 candidate papers are available, as well as one structured data extraction file per included study (63 files), mapping extracted evidence to the research questions addressed.

Acknowledgements

This work was partially supported by INES.IA (National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence),[3] CNPq grant 408817/2024-0. Leopoldo is partially supported by CNPq grant 310405/2025-4. We thank the anonymous SBES 2026 reviewers for their constructive feedback.

References

[1] Bashair Althani and Souheil Khaddaj. 2017. Systematic Review of Legacy System Migration. In *2017 16th International Symposium on Distributed Computing and Applications to Business, Engineering and Science (DCABES)*. 154–157. doi:10.1109/DCABES.2017.41

[2] Wesley K. G. Assunção, Luciano Marchezan, Lawrence Arkoh, Alexander Egyed, and Rudolf Ramler. 2025. Contemporary Software Modernization: Strategies,

[3] https://www.ines.org.br

Driving Forces, and Research Opportunities. *ACM Trans. Softw. Eng. Methodol.* 34, 5, Article 142 (May 2025), 35 pages. doi:10.1145/3708527

[3] Bruno Cartaxo, Gustavo Pinto, and Sergio Soares. 2018. The Role of Rapid Reviews in Supporting Decision-Making in Software Engineering Practice. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018* (Christchurch, New Zealand) *(EASE '18)*. Association for Computing Machinery, New York, NY, USA, 24–34. doi:10.1145/3210459.3210462

[4] Bruno Cartaxo, Gustavo Pinto, and Sergio Soares. 2020. *Rapid Reviews in Software Engineering*. Springer International Publishing, Cham, 357–384. doi:10.1007/978-3-030-32489-6_13

[5] Bruno Cartaxo, Gustavo Pinto, Elton Vieira, and Sérgio Soares. 2016. Evidence Briefings: Towards a Medium to Transfer Knowledge from Systematic Reviews to Practitioners *(ESEM '16)*. Association for Computing Machinery, New York, NY, USA, Article 57, 10 pages. doi:10.1145/2961111.2962603

[6] Dhanushka Jayasuriya, Samuel Ou, Saakshi Hegde, Valerio Terragni, Jens Dietrich, and Kelly Blincoe. 2024. An extended study of syntactic breaking changes in the wild. *Empirical Softw. Engg.* 30, 2 (Dec. 2024), 45 pages. doi:10.1007/s10664-024-10563-4

[7] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. 2017. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/thou-shalt-not-depend-me-analysing-use-outdated-javascript-libraries-web/

[8] Erica Mourão, João Felipe Pimentel, Leonardo Murta, Marcos Kalinowski, Emilia Mendes, and Claes Wohlin. 2020. On the performance of hybrid search strategies for systematic literature reviews in software engineering. *Information and Software Technology* 123 (2020), 106294. doi:10.1016/j.infsof.2020.106294

[9] S. Raemaekers, A. van Deursen, and J. Visser. 2017. Semantic versioning and impact of breaking changes in the Maven repository. *J. Syst. Softw.* 129, C (July 2017), 140–158. doi:10.1016/j.jss.2016.04.008

[10] 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 *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 35–45. doi:10.1109/ICSME46990.2020.00014

A. Primary Sources

[S1] Kumar, S.H.B.I.; Sampaio, L.R.; Martin, A.; Brito, A.; Fetzer, C.. 2024. A Comprehensive Study on the Impact of Vulnerable Dependencies on Open-Source Software. *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*.

[S2] Nguyen H.A.; Nguyen T.T.; Wilson Jr. G.; Nguyen A.T.; Kim M.; Nguyen T.N.. 2010. A Graph-based Approach to API Usage Adaptation. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*.

[S3] Meng S.; Wang X.; Zhang L.; Mei H.. 2012. A History-Based Matching Approach to Identification of Framework Evolution. *Proceedings - International Conference on Software Engineering*.

[S4] Ketkar A.; Ramos D.; Clapp L.; Barik R.; Ramanathan M.K.. 2024. A Lightweight Polyglot Code Transformation Language. *Proceedings of the ACM on Programming Languages*.

[S5] Du X.; Ma J.. 2022. AexPy: Detecting API Breaking Changes in Python Packages. *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*.

[S6] Navarro N.; Alamir S.; Babkin P.; Shah S.. 2023. An Automated Code Update Tool for Python Packages. *Proceedings - 2023 IEEE International Conference on Software Maintenance and Evolution, ICSME 2023*.

[S7] Jayasuriya D.; Ou S.; Hegde S.; Terragni V.; Dietrich J.; Blincoe K.. 2025. An extended study of syntactic breaking changes in the wild. *Empirical Software Engineering*.

[S8] Haryono S.A.; Thung F.; Lo D.; Jiang L.; Lawall J.; Kang H.J.; Serrano L.; Muller G.. 2022. AndroEvolve automated Android API update with data flow analysis and variable denormalization. *Empirical Software Engineering*.

[S9] Haryono S.A.; Thung F.; Lo D.; Jiang L.; Lawall J.; Jin Kang H.; Serrano L.; Muller G.. 2021. AndroEvolve Automated Update for Android Deprecated-API Usages. *Proceedings - International Conference on Software Engineering*.

[S10] Brito A.; Xavier L.; Hora A.; Valente M.T.. 2018. APIDiff: Detecting API breaking changes. *25th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2018 - Proceedings*.

[S11] Gao X.; Radhakrishna A.; Soares G.; Shariffdeen R.; Gulwani S.; Roychoudhury A.. 2021. APIfix: Output-oriented program synthesis for combating breaking changes in libraries. *Proceedings of the ACM on Programming Languages*.

[S12] Fazzini M.; Xin Q.; Orso A.. 2020. APIMigrator: An API-usage migration tool for Android apps. *Proceedings - 2020 IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems, MOBILESoft 2020*.

[S13] Robillard M.P.; Bodden E.; Kawrykow D.; Mezini M.; Ratchford T.. 2013. Automated API property inference techniques. *IEEE Transactions on Software Engineering*.

[S14] Fazzini M.; Xin Q.; Orso A.. 2019. Automated API-Usage update for android apps. *ISSTA 2019 - Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[S15] Thung F.; Haryono S.A.; Serrano L.; Muller G.; Lawall J.; Lo D.; Jiang L.. 2020. Automated Deprecated-API Usage Update for Android Apps: How Far are We?. *SANER 2020 - Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution, and Reengineering*.

[S16] Dig D.; Johnson R.. 2006. Automated upgrading of component-based applications. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*.

[S17] Haryono S.A.; Thung F.; Kang H.J.; Serrano L.; Muller G.; Lawall J.; Lo D.; Jiang L.. 2020. Automatic android deprecated-api usage update by learning from single updated example. *IEEE International Conference on Program Comprehension*.

[S18] Duan Y.; Gao L.; Hu J.; Yin H.. 2019. Automatic generation of non-intrusive updates for third-party libraries in Android applications. *RAID 2019 Proceedings - 22nd International Symposium on Research in Attacks, Intrusions and Defenses*.

[S19] Almeida A.; Xavier L.; Valente M.T.. 2024. Automatic Library Migration Using Large Language Models: First Results. *International Symposium on Empirical Software Engineering and Measurement*.

[S20] Perkins J.H.. 2005. Automatically generating refactorings to support API evolution. *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*.

[S21] Reyes F.; Baudry B.; Monperrus M.. 2024. Breaking-Good: Explaining Breaking Dependency Updates with Build Analysis. *Proceedings - 2024 IEEE International Conference on Source Code Analysis and Manipulation, SCAM 2024*.

[S22] Henkel J.; Diwan A.. 2005. CatchUp! capturing and replaying refactorings to support API evolution. *Proceedings - 27th International Conference on Software Engineering, ICSE05*.

[S23] Haryono S.A.; Thung F.; Lo D.; Lawall J.; Jiang L.. 2021. Characterization and Automatic Updates of Deprecated Machine-Learning API Usages. *Proceedings - 2021 IEEE International Conference on Software Maintenance and Evolution, ICSME 2021*.

[S24] Savga L.; Rudolf M.; Götz S.. 2008. ComeBack! A refactoring-based tool for binary-compatible framework upgrade. *Proceedings - International Conference on Software Engineering*.

[S25] Zhong H.; Meng N.. 2024. Compiler-Directed Migrating API Callsite of Client Code. *Proceedings - International Conference on Software Engineering*.

[S26] Scalabrino S.; Bavota G.; Linares-Vasquez M.; Lanza M.; Oliveto R.. 2019. Data-driven solutions to detect API compatibility issues in android: An empirical study. *IEEE International Working Conference on Mining Software Repositories*.

[S27] Ducasse S.; Polito G.; Zaitsev O.; Denker M.; Tesone P.. 2022. Deprewriter: On the fly rewriting method deprecations. *Journal of Object Technology*.

[S28] Møller A.; Nielsen B.B.; Torp M.T.. 2020. Detecting locations in JavaScript programs affected by breaking library changes. *Proceedings of the ACM on Programming Languages*.

[S29] Kula R.G.; German D.M.; Ouni A.; Ishio T.; Inoue K. 2018. Do developers update their library dependencies: An empirical study on the impact of security advisories on library migration. *Empirical Software Engineering*.

[S30] Salza P.; Palomba F.; Di Nucci D.; D'Uva C.; De Lucia A.; Ferrucci F.. 2018. Do developers update third-party libraries in mobile apps. *Proceedings - International Conference on Software Engineering*.

[S31] Leuenberger M.. 2019. Exploring example-driven migration. *ACM International Conference Proceeding Series*.

[S32] Lamothe M.; Shang W.. 2018. Exploring the use of automated API migrating techniques in practice An experience report on Android. *Proceedings - International Conference on Software Engineering*.

[S33] Winter V.L.; Mametjanov A.. 2007. Generative programming techniques for Java library migration. *GPCE'07 - Proceedings of the Sixth International Conference on Generative Programming and Component Engineering*.

[S34] Zhang Z.; Zhu H.; Wen M.; Tao Y.; Liu Y.; Xiong Y.. 2020. How Do Python Framework APIs Evolve? An Exploratory Study. *SANER 2020 - Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution, and Reengineering*.

[S35] Zaitsev O.; Ducasse S.; Anquetil N.; Thiefaine A.. 2022. How Libraries Evolve: A Survey of Two Industrial Companies and an Open-Source Community. *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*.

[S36] Serbout S.; Pautasso C.. 2024. How Many Web APIs Evolve Following Semantic Versioning?. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*.

[S37] Venturini D.; Cogo F.R.; Polato I.; Gerosa M.A.; Wiese I.S.. 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*.

[S38] Narasimhan K.; Reichenbach C.; Lawall J.. 2017. Interactive data representation Migration: Exploiting program dependence to aid program transformation. *PEPM*

2017 - Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, co-located with POPL 2017.

[S39] Xu S.; Dong Z.; Meng N.. 2019. Meditor: Inference and application of API migration edits. *IEEE International Conference on Program Comprehension.*

[S40] Xi Y.; Shen L.; Gui Y.; Zhao W.. 2019. Migrating deprecated API to documented replacement: Patterns and tool. *ACM International Conference Proceeding Series.*

[S41] Štrobl R.; Troníček Z.. 2013. Migration from deprecated API in java. *SPLASH 2013 - Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, and Applications: Software for Humanity.*

[S42] Haryono S.A.; Thung F.; Lo D.; Lawall J.; Jiang L.. 2021. MLCatchUp: Automated Update of Deprecated Machine-Learning APIs in Python. *Proceedings - 2021 IEEE International Conference on Software Maintenance and Evolution, ICSME 2021.*

[S43] Wu W.. 2011. Modeling framework API evolution as a multi-objective optimization problem. *IEEE International Conference on Program Comprehension.*

[S44] Alrubaye H.; Mkaouer M.W.; Ouni A.. 2019. On the use of information retrieval to automate the detection of third-party Java library migration at the method level. *IEEE International Conference on Program Comprehension.*

[S45] Şavga I.; Rudolf M.; Götz S.; Aßmann U.. 2008. Practical refactoring-based framework upgrade. *GPCE'08: Proceedings of the ACM SIGPLAN 7th International Conference on Generative Programming and Component Engineering.*

[S46] Dilhara M.; Dig D.; Ketkar A.. 2023. PYEVOLVE: Automating Frequent Code Changes in Python ML Systems. *Proceedings - International Conference on Software Engineering.*

[S47] Dig D.; Negara S.; Johnson R.; Mohindra V.. 2008. ReBA: Refactoring-aware binary adaptation of evolving libraries. *Proceedings - International Conference on Software Engineering.*

[S48] Schmiedmayer P.; Bauer A.; Bruegge B.. 2023. Reducing the Impact of Breaking Changes to Web Service Clients During Web API Evolution. *Proceedings - 2023 IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems, MOBILESoft 2023.*

[S49] Kapur P.; Cossette B.; Walker R.J.. 2010. Refactoring references for library migration. *ACM SIGPLAN Notices.*

[S50] Balaban I.; Tip F.; Fuhrer R.. 2005. Refactoring support for class library migration. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA.*

[S51] Tsantalis N.; Ketkar A.; Dig D.. 2022. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering.*

[S52] Zhu C.; Saha R.K.; Prasad M.R.; Khurshid S.. 2021. Restoring the Executability of Jupyter Notebooks by Automatic Upgrade of Deprecated APIs. *Proceedings - 2021 36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021.*

[S53] Nielsen B.B.; Torp M.T.; Moller A.. 2021. Semantic Patches for Adaptation of JavaScript Programs to Evolving Libraries. *Proceedings - International Conference on Software Engineering.*

[S54] Kang H.J.; Thung F.; Lawall J.; Muller G.; Jiang L.; Lo D.. 2019. Semantic patches for java program transformation. *Leibniz International Proceedings in Informatics, LIPIcs.*

[S55] Dagenais B.; Robillard M.P.. 2009. SemDiff: Analysis and recommendation support for API evolution. *Proceedings - International Conference on Software Engineerings.*

[S56] Ni A.; Ramos D.; Yang A.Z.H.; Lynce I.; Manquinho V.; Martins R.; Le Goues C.. 2021. SOAR: A synthesis approach for data science API refactoring. *Proceedings - International Conference on Software Engineering.*

[S57] Salza P.; Palomba F.; Di Nucci D.; De Lucia A.; Ferrucci F.. 2020. Third-party libraries in mobile apps: When, how, and why developers update them. *Empirical Software Engineering.*

[S58] Sawant A.A.; Robbes R.; Bacchelli A.. 2019. To react, or not to react: Patterns of reaction to API deprecation. *Empirical Software Engineering.*

[S59] Thung F.; Kang H.J.; Jiang L.; Lo D.. 2019. Towards Generating Transformation Rules without Examples for Android API Replacement. *Proceedings - 2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019.*

[S60] Yamaguchi D.; Iwatsuka T.. 2022. Two-Stage Patch Synthesis for API Migration from Single API Usage Example. *Proceedings - Asia-Pacific Software Engineering Conference, APSEC.*

[S61] Jayasuriya D.; Terragni V.; Dietrich J.; Ou S.; Blincoe K.. 2023. Understanding Breaking Changes in the Wild. *ISSTA 2023 - Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis.*

[S62] Yasumatsu T.; Watanabe T.; Kanei F.; Shioji E.; Akiyama M.; Mori T.. 2019. Understanding the responsiveness of mobile app developers to software library updates. *CODASPY 2019 - Proceedings of the 9th ACM Conference on Data and Application Security and Privacy.*

[S63] Dig D.. 2005. Using refactorings to automatically update component-based applications. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA.*