Software Dependency Version Migration: A Rapid Review on Motivations, Techniques, Tools and Challenges

Arthur V. B. da Silva[a], Leopoldo Teixeira[b], Wesley K. G. Assunção[c], Bruno Cartaxo[a]

[a] *Federal Institute of Pernambuco, Recife, Brazil*
[b] *Federal University of Pernambuco, Recife, Brazil*
[c] *North Carolina State University, Raleigh, United States of America*

Abstract

Context: Modern software development relies heavily on frequent updates to dependencies (e.g. third-party libraries, frameworks and APIs), which introduce the need for consistent and efficient migration practices to avoid breaking changes. As these dependencies evolve, developers face the challenge of adapting their code-bases to maintain compatibility, security, and performance.
Goal: We investigate the current landscape of dependency version migration, focusing on identifying the motivations, techniques, tools, and challenges to conduct such activity. The study also aims to understand the strategies to detect the need for dependency migration and the trade-offs involved in these decisions.
Method: We conducted a rapid review of the literature using the Scopus search engine, which returned 2,637 papers in February 2025. After applying predefined inclusion and exclusion criteria, we ended up with 63 included studies, which we extracted data to answer each of the research questions.
Results: The findings indicate that developers rarely migrate dependencies unless strongly motivated by bug fixes, compatibility, or security fixes. The strategies to identify dependencies that require updates rely on automated tools or methods. Typically, strategies to detect breaking changes are manual inspection or the use of specialized tools. Several tools and strategies have been proposed to support dependency updates, many of which rely on static analysis, rule-based transformations, or example-driven techniques. However, these tools often involve trade-offs between precision, usability, and manual effort.
Conclusions: This review highlights that dependency migration is not a common practice and when it happens it is for reasons of fixes and compatibility. The proposed strategies to identify dependencies that need to be updated rely on automated tools, on the other hand, the strategies to identify braking changes varies between cases using manual and cases using automation. Furthermore, there is no silver bullet solution in the dependency migration techniques, they have different proposals and specificity.

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. However, the evolution of these dependencies, especially those that introduce breaking changes, often forces developers to adapt their systems to ensure continued functionality. In a large-scale study of the Maven repository, Raemaekers et al. [8] found that breaking changes still occur even when semantic versioning is used, suggesting that updating dependencies may lead to unexpected risks.

The prevalence of breaking changes, many systems continue to rely on outdated dependencies, even in critical web applications. Lauinger et al. [6] showed that a significant portion of websites include obsolete and vulnerable JavaScript libraries, indicating that dependency version migration is not a consistently adopted practice in projects.

Based on these findings, dependency version migration has some impediments, but is needed to avoid obsolete and vulnerable features that can result in security issues. According to this information, one needs to understand the migration of dependency versions.

In this paper, we present a Rapid Review of the literature on dependency version migration. The goal is to synthesize existing knowledge and evidence on the motivations, techniques, tools, and challenges to conduct such an activity. The study is guided by seven Research Questions

Email addresses: arthurvini2703@gmail.com (Arthur V. B. da Silva), lmt@cin.ufpe.br (Leopoldo Teixeira), wguezas@ncsu.edu (Wesley K. G. Assunção), brunocartaxo@gmail.com (Bruno Cartaxo)

(RQs), which were answered with findings from 63 out of 2,637 articles selected from a search conducted in Scopus.

In summary, this paper makes the following major contributions:

- Investigates whether developers migrate to new versions and examines the reasons behind their decisions;
- Surveys strategies for identifying dependencies that need to be updated, as well as the strategies for detecting updates that introduce breaking changes;
- Analyzes the underlying techniques employed by dependency version migration tools, highlighting their trade-offs and identifying the main solutions available.

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 based on the seven research questions. Section 5 discusses the implications of these findings. Finally, Section 6 concludes the paper and outlines the directions for future work.

2. Related Work

The migration of dependencies and the evolution of APIs have been extensively explored in recent years, particularly with regard to how developers handle version updates and the challenges posed by breaking changes. Several studies have investigated how often developers update their dependencies in real-world projects.

For instance, Kula et al. [S29] analyzed how security advisories influence library migration decisions, revealing that many developers still rely on outdated versions due to concerns about migration effort and compatibility. Similarly, in the context of mobile development, Morales et al. [S30] observed that developers tend to postpone updates to third-party libraries, often due to perceived migration effort or uncertainty regarding breaking changes introduced in newer versions.

Other studies have focused on the tools and techniques available to support automated or semi-automated migration. Dig et al. [S63] proposed using refactorings to automate component updates, and presented tools capable of transforming client code based on change logs. Hora and Robbes [S13] conducted a comprehensive review of techniques for inferring API changes, highlighting static and dynamic analysis approaches. Moreover, Bavota et al. [S32] provided an experience report on the practical use of automated migration tools in the Android ecosystem, identifying trade-offs and integration challenges.

Several secondary studies have investigated different aspects of software modernization and legacy system migration. Althani and Khaddaj [1] conducted a systematic review focused on the migration of legacy systems, identifying challenges related to data, architecture, and interface transformation. Their findings are particularly relevant in the context of dependency migration, as legacy systems often rely on outdated components, and their migration to modern dependencies involves complex re-engineering of both the code and the architecture.

Similarly, Assunção et al. [2] provided an overview of contemporary software modernization strategies, highlighting the challenges of integrating new dependencies into existing systems and categorizing the driving forces behind modernization initiatives. These studies underscore the broad challenges involved in modernizing and evolving software systems, but often focus on system-level modernization rather than the specific issue of dependency migration.

To accelerate evidence collection in software engineering, recent studies have adopted rapid review methodologies. Cartaxo et al. [3] discussed the role of rapid reviews as an efficient alternative to full systematic reviews, especially when quick decision-making is required in industry. They argue that, when carefully planned, rapid reviews can provide relevant insights without compromising methodological rigor. In a complementary study, the authors proposed the use of "evidence briefings" as a format to improve the transfer of knowledge from reviews to practitioners [5], emphasizing accessibility and clarity of findings. These contributions support the feasibility and value of applying rapid reviews to guide software engineering practices.

While the existing literature provides valuable insights into dependency migration, the majority of studies tend to focus on isolated aspects, such as security vulnerabilities or migration to specific frameworks. However, there remains a lack of consolidated evidence regarding the tools, techniques, and challenges that developers face when updating dependencies in modern software ecosystems, especially those involving complex dependencies and breaking changes. This gap in the literature motivates the rapid review presented in this study, which aims to synthesize the current state of knowledge and provide actionable insights for developers and researchers alike.

3. Research Method

We performed a Rapid Review, following Cartaxo et al.'s guidelines [4, 3], in order to provide timely evidence aiming, ultimately, to support decision-making in a real-world software development project.

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 practitioners context".

In that regard, this rapid review (RR) was conducted in close collaboration with a practitioner, Bruno Cartaxo,

of Zulu, 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.

Bruno Cartaxo validated all activities of this RR process and actively participated in the definition of the protocol and selection criteria.

3.2. Research Questions

Here we present the research questions of this study, which 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?**
  Here, we explore the common drivers for updating dependencies.
- **RQ3: What are the strategies to identify dependencies that need to be updated?**
  This question focuses on the strategies and tools used to locate deprecated or outdated API usages in a codebase.
- **RQ4: What are the strategies to identify dependency's version migration that provoke breaking changes?**
  We examine the mechanisms employed to identify and understand breaking changes, whether manually or using automated tools.
- **RQ5: What tools have been proposed to support dependency version migration?**
  This question surveys existing tools that aid in the migration process. In particular, tools that automates or semi-automates breaking changes in the client code when migrating to newer dependency versions?
- **RQ6: What are the underlying techniques used by these dependency version migration tools?**
  We categorize and analyze the underlying techniques adopted by migration tools (e.g. AI driven, syntactic driven, semantic drive, etc).
- **RQ7: What are the trade-offs of existing tools and techniques?**
  This question discusses the limitations, performance concerns, required human effort, and generalization issues faced by current approaches.

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

3.3. Search Strategy

We used **Scopus**[1] as the exclusive database for our literature search, due to its comprehensive indexing of high-quality, peer-reviewed computer science publications, as already validated by Mourão et al. [7]. On February 27, 2025, we executed a structured search query targeting papers published from 2015 onwards (a timeframe of 10 years) with terms related to libraries, APIs, frameworks, and dependencies in the context of migration, evolution, or versioning. The following query returned a total of **2,637** papers.

The gray literature was excluded from this review. Since the practitioner's company, Zulu, already maintains internal data and analyses related to practical migration experiences (as reported by Bruno Cartaxo), the practitioner sought to complement this industrial perspective with evidence from the state-of-the-art in academic research.

> ***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"))*

3.4. Exclusion Criteria

In order to refine the scope of the review and ensure relevance, we defined a set of exclusion criteria. Papers were 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 defined research questions

EC5 focused exclusively on techniques for migrating between different dependencies (e.g. from one library to another) rather than version migration of the same dependency.

These criteria were applied in a multi-phase filtering process. During the first phase, we conducted a **title-based screening** of the 2,637 initial results. Four reviewers independently evaluated separate subsets of the papers. Thus, each paper was analyzed by a single person, as preconized by RRs' guideline [4] At this phase, papers could be marked as In, Out, or Maybe — with "Maybe" indicating uncertainty or a need for further inspection. After

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

this phase, the number of papers was reduced to 438 (sum of "Ins" and "Maybes").

In the second phase, we reviewed the **abstract** of each of the 438 remaining papers. This was again carried out by the same four reviewers, independently and without pair cross-checking. The same three labels (In, Out, Maybe) were used, and this step reduced the dataset to 212 papers.

The final screening phase involved reading the **full text** of the 212 papers. This step was performed solely by the first author, who decided on inclusion using only the "In" or "Out" labels. At the end of this phase, a final set of 63 papers was selected for inclusion in this RR for data extraction.

This multi-stage filtering process is summarized in the following Figure 1, which illustrates the number of papers retained after each phase of exclusion. The spreadsheets of the selection process are publicly available [2].

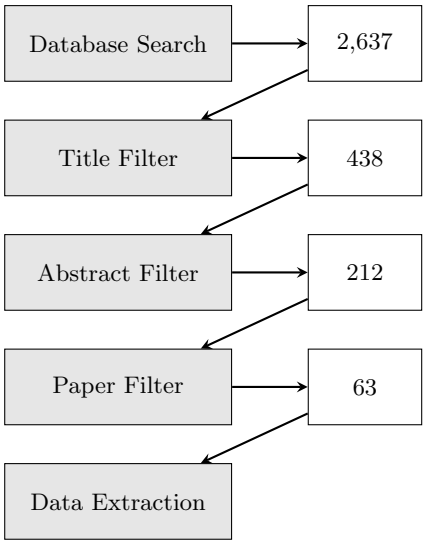

Figure 1: Systematic paper filtering

3.5. Data Extraction

In parallel with the final screening, we designed a form to collect relevant information from each included study. For each of the 63 selected papers, we recorded which of the seven research questions the paper has addressed. A dedicated spreadsheet was used to track the mapping between studies and RQs. This mapping facilitated the synthesis of evidence across studies and helped organize the results by question. The forms and spreadsheets of the data extraction process are publicly available [3].

4. Results

This section presents the results of the data extraction process based on the 63 primary studies selected through the rapid review. The analysis aimed to answer the seven predefined research questions, each addressing a specific aspect of dependency version migration in software systems. For each RQ, we synthesized the findings across the studies, highlighting patterns, tools, techniques, and trade-offs reported in the literature.

The results are organized in the following subsections, one for each research question (RQ1 to RQ7), where we summarize the main insights supported by concrete examples and references to the reviewed studies.

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

A total of 6 studies discussed whether developers actually migrate to newer dependency versions. These works [[S29], [S30], [S35], [S57], [S58], [S61] report 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. This result comes from the Library Residue metric, which measures the proportion of systems still using an outdated version of a library after its peak usage. Analyzing 2,736 dependencies from 48,495 Java systems on GitHub, the average ratio of current usage to peak usage was 81.5%.

Similarly, Salza et al. [S57] observed that only 15.5% of library uses were consistently updated by developers. This result reflects the pattern of 'diligent updates', where developers consistently update libraries to the latest version in each development cycle. The authors analyzed 11,626 version histories of Android app libraries using open coding, with four researchers manually classifying update practices. Only 1,976 library usages (15.5%) followed this pattern.

Furthermore, Salza et al. [S30] noted that in 63.0% of the cases, the authors did not update the libraries' versions after their introduction. The authors 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 use, was found in 63.0% of cases.

In line with this, Jayasyriya et al. [S61] found that 71.6% of dependencies in client systems were not up-to-date with the latest version available for the library. The authors analyzed 18,415 Maven artifacts which declared 142,355 direct dependencies, of which 71.6% were not up-to-date.

A survey conducted by Zaitsev et al. [S35] with 36 developers has shown that they update their dependencies, but rarely. Three-quarters of the client developers update the dependencies version at least twice a year, and less than a half do it three times a year or more often.

In addition, Sawant et al. [S58] asked developers why they would not upgrade the version of the API they were using and identified two major reasons: The cost involved in the upgrade was often not worth it, and since the version in use was still functioning properly, there was no pressing need to upgrade.

[2] `https://drive.google.com/drive/folders/1zIwYSCik4Bsw7I45P4p6Fzm_4TbfHN4j?usp=sharing`

[3] `https://drive.google.com/drive/folders/1jsTJKQHf9omDxEXb21o2HL5uL_Hmot0t?usp=sharing`

Based on the synthesis of these findings, we conclude that:

> ***RQ1 Key Takeaway:*** *Developers rarely update their dependencies to newer versions.*

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

A total of 2 studies focused discussing the developers' motivations to migrate dependencies. These works [S57], [S62] pointed three main motivations.

A survey conducted by Salza et al. [S57] with 73 mobile developers has shown that avoiding bug propagation and making the app compatible with new Android releases are the main reasons why developers update their code.

Furthermore, Yasumatsu et al. [S62] collected 21,046 Android apps and concluded that the security fix campaign was effective in encouraging app developers to adopt library version updates.

Based on the synthesis of these findings, we conclude that:

> ***RQ2 Key Takeaway:*** *Developers' main motivations to migrate to a new version are: bug fixes, compatibility and security fixes.*

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

A total of 2 studies focused on strategies to identify dependencies that need to be update. These works [S01], [S06] proposed two strategies.

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

Furthermore, Navarro et al. [S06] proposed a method to automatically source API deprecation data for popular Python libraries by crawling and parsing their release notes from the web. Their strategy involves extracting libraries that use Sphinx for documentation, retrieving their version history from the PyPI API, and locating their release notes through programmatic Google searches. Deprecations are then parsed from the standardized HTML structure of the documentation.

> ***RQ3 Key Takeaway:*** *The proposed strategies are using a tool called VODA that searches for dependencies with vulnerability and a method to automatically source API deprecation data for popular Python libraries by crawling and parsing their release notes from the web.*

4.4. RQ4: What are the strategies to identify dependency's version migration that provoke breaking changes?

A total of 8 studies focused on strategies to identify dependency' version migrations that provoke breaking changes. Among them, the following works — [S05], [S10], [S21], [S34], [S36], and [S61] — employed automated strategies by using or proposing tools. On the other hand, the studies [S08] and [S47] relied on manual strategies.

Automated Strategies, mainly involve comparing two versions of code and executing algorithms to identify potential breaking changes.

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

Similarly, Brito et al. [S10] developed APIDIFF, a tool that detects API breaking and non-breaking changes based on three API elements: types, methods, and fields.

Expanding on this, Reyes et al. [S21] proposed BreakingGood, which identifies dependency breaking changes using three inputs, the source code dependency version, the new dependency version and the source code. Based on these inputs, the tool determines whether a breaking change has occurred and provides an explanation for the detected issue.

Zhang et al. [S34] presented PyCombat, which performs static analysis to detect breaking API changes. This tool operates in two phases: first, it extracts an API knowledge base; second, it detects issues based on the extracted data.

Other tools include oasdiff, used by Serbout & Pautasso [S36], which is a command line tool and GO package designed to compare OpenAPI specifications.

Additionally, Jayasuriya et al. [S61] used a tool called jacimp, that detects breaking changes between two library versions using two versions of jar files and running a static analysis on the changes between them.

Manual Strategies, mainly involve updating the dependency version and evaluating whether the system continues to work correctly, either by executing the code or running the project's tests.

For example, Jayasuriya et al. [S08] adopted a manual approach where they updated the dependency version and checked whether the code still working as expected. If the update introduced failures, a breaking change has been identified.

Likewise, Venturini et al. [S47] applied a similar manual strategy by updating the dependency version, installing it, and running the project's test suite. If the tests failed, they considered it evidence of a breaking change.

Based on the synthesis of these findings, we conclude that:

RQ4 Key Takeaway: *The strategies can be manual — typically involving updating the dependency version and testing whether the application still works — or automatic, using tools that compare the two versions of the code and run algorithms to detect potential breaking changes.*

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. This finding underscores the interest in tool-supported solutions to help developers cope with the challenges of evolving software dependencies.

Many of the tools identified are proposed within the papers themselves, with only three studies focusing on analyzing pre-existing tools [[S13], [S29], [S63]]. In some cases, a tool appears in more than one paper, such as AndroEvolve [[S08], [S09]], AppEvolve [[S14], [S15]], and ML-CatchUp [[S23], [S42]], indicating ongoing development or evaluation efforts by the research community.

The variety of tools is noteworthy. Some, such as LIBSYNC [S02] and HiMa [S03], use advanced techniques like graph-based and heuristic-driven analysis to recommend code edits. Others focus on the integration with development environments, such as the Paper's PyCharm Plugin [S06] or Our Workbench [S28], aiming to embed migration assistance directly into developers' workflows. Tools like APIFix [S11], APIMigrator [S12], and CocciEvolve [S17] emphasize automated patch generation and migration script synthesis, while others such as RefactoringCrawler [S16] and RefactoringMiner 2.0 [S51] specialize in detecting structural changes between versions.

Several tools are domain-specific or target particular programming languages or ecosystems. For example, DEPREWRITER [S27] is tailored to Pharo, Apodini Migrator [S48] supports semantic migrations in Swift, and JSFIX [S53] handles JavaScript APIs. This diversity reflects the varying needs of developers across languages and platforms.

Diverse papers in the amount focused on tools design specifically for Android ecosystem. These include AndroEvolve [S08], ApiMigrator [S12], AppEvolve [S14], CocciEvolve [S17], LIBBANDAID [S18], ACRYL [S26] and NEAT [S59].

Between these tools focused in the Android ecosystem, the AppEvolve [S14] was developed first as a solution for Android API evolution. Building on this approach the CocciEvolve [S17] was inspired by AppEvolve. Later the AndroEvolve [S08] as a tool compared to CocciEvolve. This forms a clear progression in the research, with each tool refining and adapting the techniques of its predecessor.

A subset of the papers focuses on tools developed as IDE plugins, allowing developers to use them directly within their preferred development environments. For instance, the Paper's PyCharm Plugin [S06] was built for PyCharm, while HiMa [S03], CatchUp! [S22], and Trident [S49] were all designed as plugins for the Eclipse IDE.

Looking at the temporal distribution of the studies in Figure 2, since 2019, there is a growing interest in tools to automate or semi-automate dependency version migrations. This trend may reflect the increasing complexity of software ecosystems and the need for scalable, automated solutions to manage evolving dependencies.

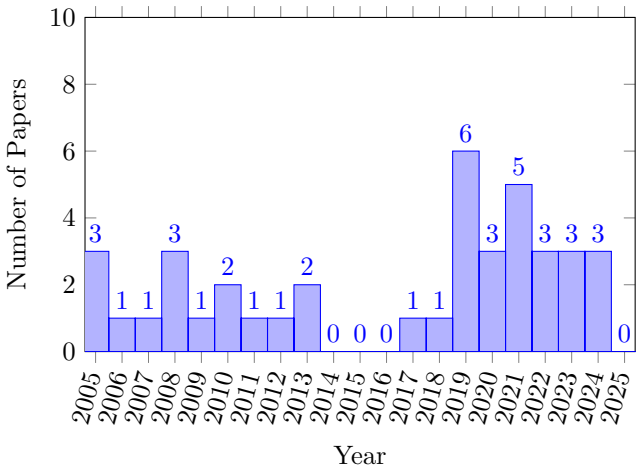

Figure 2: Distribution of papers by publication year for the RQ5, RQ6 and RQ7

Based on the synthesis of these findings, we conclude that:

RQ5 Key Takeaway: *The development of automated tools for dependency migration is an active and growing area of research, with most tools emerging in the last decade and addressing diverse programming environments and challenges.*

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 tools that support dependency version migration. After analyzing these papers, the techniques were grouped into four major categories based on their operational characteristics and methodological focus. A summary of these findings is presented in Table 1, which categorizes the papers and techniques by their respective groups.

AI and Example-Driven Techniques, found in 21 papers, include methods that take advantage of 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 are used to express common migration actions in a structured and repeatable way, offering control and predictability in the transformation process.

Static and Structural Analysis, the most prominent group, is represented in 29 papers. This category encompasses techniques that operate by examining the structure and composition of the code without requiring its execution. These approaches focus on the syntactic and structural elements of software systems to identify potential migration points and assess compatibility with newer versions of dependencies. Their popularity can be attributed to their general applicability across languages and contexts.

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

Based on the synthesis of these findings, we conclude that:

> ***RQ6 Key Takeaway:*** *The most common techniques used by code migration tools are those grounded in static and structural analysis. In addition, AI-driven and example-based methods, along with rule-based transformations, also play a significant role. Semantic and contextual techniques, while less frequent, contribute unique advantages in specialized scenarios.*

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 migration. These trade-offs are closely tied to the type of technique employed, as well as factors such as design decisions, implementation strategies, and tool maturity.

AI and Example-Driven Techniques, this group includes tools that apply learning, historical analysis, or reasoning through examples (e.g., LIBSYNC, APIFix, PyEvolve, MLCatchUp). Tools in this category tend to offer high precision and recall, especially when trained or calibrated with concrete usage scenarios. However, they usually have higher computational cost (e.g., LIBSYNC, HiMa) or rely on datasets of past migrations to perform well. Their specialized behavior makes them powerful in solving complex or deep changes, but this often comes at the expense of generalization and reuse across unrelated contexts.

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 Depreciation	[S27]	1

Table 1: Techniques and Papers' tools Grouped by Category

Rule and Pattern-Based Transformations, these tools (e.g., TAPIR, JaSCUT, DAAMT, Coccinelle4j) leverage explicit or inferred rules to automate migration steps. While often fast and relatively accurate for well-documented or structured APIs, they still require some degree of manual configuration or predefinition of rules (as seen in TAPIR and DAAMT). Moreover, their dependency on documentation quality or pattern availability limits their application in less predictable migration scenarios. Despite their limitations, they are frequently preferred in production settings due to their deterministic behavior and transparency.

Static and Structural Analysis, tools in this group (e.g., RefactoringMiner 2.0, SemDiff, ReBa, RELANCER) are the most frequent in the dataset and tend to focus on the structural behavior of code without requiring execution. These tools are precise in identifying syntactic and structural differences and are often favored for their scalability. However, their performance may degrade with very complex or deeply integrated changes (e.g., structural transformations across multiple layers). Additionally, some tools (e.g., SemDiff) require a complete version history, making them less flexible in ad hoc migration scenarios.

Semantic and Contextual Analysis, this group includes tools like Apodini Migrator and JDiff, which attempt to capture meaning beyond code structure. While potentially powerful in detecting higher-level impacts, they are often limited in scope or require additional manual intervention. These tools struggle with deep semantic changes, and their coverage depends heavily on external metadata (e.g., change logs, semantic tags, contextual cues). As a result, while insightful, their adoption is still limited due to the complexity of integrating semantic understanding into automated workflows.

In summary, different categories of techniques bring distinct trade-offs to migration tools. While AI-driven and rule-based tools offer automation and precision, they are often limited by computational cost or the need for predefined knowledge. Static analysis approaches are scalable and robust but may lack semantic depth. Meanwhile, semantic techniques show promise for complex changes, yet still face practical limitations in terms of tooling support and automation.

Based on the synthesis of these findings, we conclude that:

> ***RQ7 Key Takeaway:*** *Dependency migration tools present distinct trade-offs based on the techniques they employ. While some favor precision and automation, others prioritize scalability or semantic depth. Choosing the right approach depends on the migration context and project needs.*

5. Discussion

This section presents a comprehensive discussion of the findings derived from the research questions. The goal is to interpret the results, highlight their significance, and reflect on their implications for both research and practice. In addition, this section addresses potential threats to the validity of the study.

The discussion is structured into three parts. First, we summarize and interpret the main findings, organized by research question. Next, we explore the implications of these findings, distinguishing between research-oriented and practice-oriented impacts. Finally, we analyze threats to the validity of our review.

5.1. Findings

This subsection discusses the key findings obtained from the analysis of the selected studies per research question.

RQ1: Do developers commonly migrate their dependencies to newer versions? The review shows that dependency migration is not yet a widespread or systematic practice. While some developers perform updates when necessary, most tend to postpone migrations due to fear of breaking changes.

RQ2: What motivates developers to migrate dependencies to newer versions? Bug fixes, compatibility, and security fixes are the main reasons that drive developers to update dependencies. These findings reflect practical and technical priorities in software projects, where stability and risk mitigation often outweigh innovation or performance gains.

RQ3: What are the strategies to identify dependencies that need to be updated? The proposed strategies aim in automatic tools or methods to identify these dependencies that need to be updated.

RQ4: What are the strategies to identify dependency's version migration that provoke breaking changes? The strategies found commonly identify either by using specialized automatic tools or manually testing the new version.

RQ5: What tools have been proposed to support dependency version migration? A wide variety of tools have been proposed to support code migration, with different scopes, ecosystems, and levels of automation. Most tools aim to minimize developer effort and increase migration safety.

RQ6: What are the underlying techniques used by these dependency version migration tools? Each class of techniques bring together a range of approaches, highlighting that there is no silver bullet solution. Tool design often involves trade-offs between automation, flexibility, and accuracy.

The AI-based and example-driven category includes techniques that rely on examples, learning, or automated reasoning, such as Artificial Intelligence, example-based approaches, and search-based methods.

The rule and pattern-based transformation category encompasses techniques that operate using explicit or inferred rules, including rule-based transformations, pattern-based techniques, and Semantic Patch Language.

The static and structural analysis category covers methods that examine code without executing it, such as graph-based analysis, static analysis, and version differencing.

Lastly, the semantic and contextual category includes techniques that go beyond the syntactic structure of code, considering broader aspects like heuristic-based approaches, documentation analysis, and dynamic analysis.

When examining the techniques individually, rather than by broader category, the most frequently adopted are Static Analysis, Rule-Based Transformation, and Example-Based methods.

RQ7: What are the trade-offs of existing tools and techniques? Each class of techniques involves trade-offs that vary according to the underlying strategy employed.

The AI-based and example-driven tools often achieve high precision and effectiveness, especially in well defined domains with available training data. However, they tend to require more computational resources and are particularly limited in cases where historical examples are scarce or highly domain-specific.

The rule and pattern-based transformation approaches are effective when migrations follow consistent, well documented structures. These techniques are relatively easy to implement but struggle with flexibility, especially in dynamic or poorly documented environments.

The static and structural analysis methods are scalable and suitable for large code-bases, offering consistent results without needing runtime data. They perform well in projects with strong structure and clear typing but may miss deeper semantic issues.

Finally, semantic and contextual analysis techniques are useful for capturing abstract or high-level changes, yet they often require manual input and exhibit limited scalability. These methods can be less reliable in large-scale or highly variable code-bases.

5.2. Implications

This subsection presents the implications of the findings from both a research and a practical perspective.

Research. The findings highlight several opportunities for future research in the area of code migration. First, further investigation is needed to understand why developers often avoid applying updates. Second, when code migration is primarily driven by fixes concerns, is this a positive indication, or does it point to deeper systemic issue?

Finally, future research could explore how well current tools integrate with modern development workflows, such as CI/CD pipelines and integrated development environments (IDEs), and whether such integration enhances the adoption and effectiveness of code migration practices.

Practice. The findings suggest that developers could benefit from improved tooling and documentation practices to support safer and more frequent code migrations. In particular, automated tools can substantially reduce the manual effort involved in identifying breaking changes and adapting code, thereby making the migration process more efficient and less error prone.

However, to fully leverage the benefits of such tools, developers must be aware of the trade-offs associated with different techniques. Factors such as flexibility, accuracy, performance, and required developer input vary across tools and approaches. A clear understanding of these differences is essential to select the most appropriate solution for a given project or organizational context.

The findings also suggest that libraries and frameworks maintainers, should adopt clear semantic versioning practices, effectively communicate "breaking changes," and provide documentation and tools that facilitate the transition to new versions. This helps developers adopt updates more quickly and securely.

5.3. Limitations and Threads to Validity

This subsection discusses the potential threats to the validity of this study.

In an effort to ensure greater rigor in our research, we defined clear research questions and employed a structured data extraction process to ensure consistency in identifying and categorizing relevant information. Nonetheless, the interpretation of the data may be influenced by subjective judgment, particularly in the classification of tools and techniques across studies.

Although predefined criteria guided the screening process, the inclusion and exclusion of studies was carried out without a peer review step. Each author was responsible for independently evaluating a subset of the retrieved papers according to the inclusion and exclusion criteria to ensure coverage and reduce selection bias. However, in the final stage, only the first author performed full text evaluation and data extraction, which may introduce risks of systematic bias or oversight during the classification and interpretation of the studies.

The scope of this review was limited to studies indexed in the Scopus database, which excludes potentially relevant work published only in other sources not covered by Scopus, or gray literature. Consequently, some tools, techniques, or insights may not have been captured.

Furthermore, several exclusion criteria were applied to ensure relevance and quality, including language restrictions (EC1), peer-review status (EC2), publication date (EC3), alignment with research questions (EC4), and focus on migration between versions of the same library, language or framework, excluding studies that addressed inter-library or cross-framework migration (EC5). While these measures reinforced rigor, they may have limited the diversity of perspectives and the generalization of the findings

Considering the qualitative nature of this rapid review and its sample of 63 papers, the conclusions should be interpreted with caution. Potential researcher bias and the rapid review process may limit the depth and robustness of the findings. Therefore, while the results highlight important insights and trends, further research with larger and more comprehensive studies is necessary to validate and expand these conclusions.

Despite these limitations, we believe that this study provides valuable insights into the code migration process and lays a solid foundation for future research and practical improvements.

6. Conclusion

This study presented a Rapid Review on the topic of code migration, with the aim of synthesizing current knowledge on the motivations, techniques, tools, and trade-offs involved in this process. A total of 63 primary studies were analyzed to answer seven research questions that span both theoretical and practical aspects of code migration.

The findings reveal that developers often tend not to engage in migration unless motivated by specific needs such as bug fixes, compatibility, or security fixes. Migration detection varies, with many approaches relying on manual inspections and others on automated tools.

Furthermore, various techniques are used in migration tools, including static analysis, rule-based transformations, and AI-driven approaches, each presenting distinct trade-offs in terms of precision, usability, and computational cost.

These results provide a structured overview of current migration practices and highlight some research gaps. Future research should investigate these aspects in more depth, investigating developer perceptions, organizational constraints, and the effectiveness of support tools. Future research will be able to provide further support for this article, as well as fill in gaps and resolve some limitations and threads to validity.

Although this review followed a conscientious methodology, certain limitations remain. It was restricted to the Scopus database, peer-reviewed publications in English, and migration cases within the same library, language, or framework. In addition, the rapid nature of the review process may have limited the depth of the analysis.

Overall, the study contributes a consolidated foundation for understanding code migration and points to promising directions for future research and tool development.

Acknowledgments

The authors would like to thank the academic advisors involved in this work for their valuable guidance, feedback, and support throughout the development of this study.

This study was conducted as part of the first author's undergraduate thesis project at Instituto Federal de Pernambuco.

References

[1] Althani, B., Khaddaj, S., 2018. Systematic review of legacy system migration. International Journal of Advanced Computer Science and Applications 9 (5), 375–381.

[2] Assunção, W. K., Marchezan, L., Arkoh, L., Egyed, A., Ramler, R., 2020. Contemporary software modernization: Strategies, driving forces, and research opportunities. IEEE Access 8, 186572–186603.

[3] Cartaxo, B., Pinto, G., Soares, S., 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. EASE '18. Association for Computing Machinery, New York, NY, USA, p. 24–34.
URL https://doi.org/10.1145/3210459.3210462

[4] Cartaxo, B., Pinto, G., Soares, S., 2020. Rapid Reviews in Software Engineering. Springer International Publishing, Cham, pp. 357–384.
URL https://doi.org/10.1007/978-3-030-32489-6_13

[5] Cartaxo, B., Pinto, G., Vieira, E., Soares, S., 2016. Evidence briefings: Towards a medium to transfer knowledge from systematic reviews to practitioners. ESEM '16. Association for Computing Machinery, New York, NY, USA.
URL https://doi.org/10.1145/2961111.2962603

[6] Lauinger, T., Chaudhry, A., Arshad, S., Robertson, W., Wilson, D., Wills, C., Robertson, W., Kirda, E., 2017. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. In: Proceedings of the 2017 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE, pp. 389–404.

[7] Mourão, E., Pimentel, J. F., Murta, L., Kalinowski, M., Mendes, E., Wohlin, C., 2020. On the performance of hybrid search strategies for systematic literature reviews in software engineering. Information and Software Technology 123, 106294.
URL https://www.sciencedirect.com/science/article/pii/S0950584920300446

[8] Raemaekers, S., van Deursen, A., Visser, J., 2017. Semantic versioning and impact of breaking changes in the maven repository. Journal of Systems and Software 129, 140–158.

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*.