# DepUpgrade: Automating Dependency Upgrade through State-Path Exploration

Xiangxi Ma*
Beihang University
Beijing, China
maxiangxi@buaa.edu.cn

Yifan An*
Beihang University
Beijing, China
anyifan@buaa.edu.cn

Wentong Tian
Beihang University
Beijing, China
tianwentong2000@buaa.edu.cn

Xuanqi Wang
Beihang University
Beijing, China
wangxuanqi@buaa.edu.cn

Qingao Dong
Beihang University
Beijing, China
dongqa930@buaa.edu.cn

Xiang Gao†
Beihang University
Beijing, China
xiang_gao@buaa.edu.cn

Hailong Sun
Beihang University
Beijing, China
sunhl@buaa.edu.cn

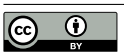
## Abstract

Third-party dependencies frequently introduce API breaking changes, and upgrading them causes compilation or runtime failures in client applications. Existing approaches have explored automating dependency upgrades, but struggle with fixing complex issues. This is mainly because traditional methods lack semantic adaptability, while recent prompt-only LLM-based migration approaches often suffer from hallucinations and fail to navigate multi-step, cascading repairs. To solve this problem, in this paper, we propose DepUpgrade, a framework that models dependency upgrade as a state-path exploration problem. A state is a snapshot of the client codebase at a particular point during migration. Transitions between states are validated via compiler feedback and rolled back if regressions occur. Instead of performing a single-pass generation, DepUpgrade systematically explores a search space of intermediate states with the goal of reaching a successful state. DepUpgrade achieves this goal in three steps: API migration knowledge extraction, repair task formulation and state-path exploration. We evaluate DepUpgrade on a benchmark of 123 migration tasks and 18 real-world projects. Experimental results demonstrate that our framework achieves an unprecedented 96.7% full repair rate on the benchmark, outperforming other baselines. On the real-world projects, DepUpgrade fully repairs 15 of the 18 projects and leaves 31 of the 1,085 initial compilation errors unresolved.

## CCS Concepts

• **Software and its engineering → Software maintenance tools**; *Software evolution*; *Automatic programming*.

*Both authors contributed equally to this work.
†Corresponding author.




*ASE '26, Munich, Germany*

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837442

## Keywords

Code Migration, Static Analysis, Knowledge Retrieval, Large Language Model

**ACM Reference Format:**
Xiangxi Ma, Yifan An, Wentong Tian, Xuanqi Wang, Qingao Dong, Xiang Gao, and Hailong Sun. 2026. DepUpgrade: Automating Dependency Upgrade through State-Path Exploration. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3837442

## 1 Introduction

Modern software systems heavily rely on third-party libraries to accelerate development. To facilitate reuse, these libraries expose Application Programming Interfaces (APIs). Client applications must invoke these APIs correctly to interact with the libraries. However, libraries evolve rapidly to introduce new features and fix bugs/vulnerabilities. While updates are essential, they may modify API contracts, potentially breaking existing client applications. Such changes, referred to as **breaking changes** [20, 36], account for approximately 15% of all API modifications [42]. As a result, developers frequently encounter compilation errors or runtime failures after dependency upgrades. Manually repairing these issues is time-consuming and error-prone, particularly in large-scale systems [26]. This indicates the necessity and importance of automatically upgrading dependencies under API-breaking changes.

Researchers have proposed various approaches to automate this process. *API-mapping-based methods* [4, 5, 7] mine correspondences between deprecated and updated APIs. *Edit-script-based methods* [10, 26, 45] infer rewrite rules from historical migration commits. *Compilation error-repairing strategies* [47] utilize predefined transformation rules driven by compiler diagnostics. More recently, *Prompt-only LLM-based approaches* [1, 19] leverage semantic reasoning to migrate code without relying on predefined rules.

Although existing approaches have achieved promising results, they face two main limitations:

**1) Traditional methods lack the semantic adaptability required for complex structural changes.** Non-LLM approaches heavily rely on static knowledge. Methods based on API mappings or edit scripts suffer from data scarcity and fail to generalize. Even compilation-error-repairing strategies (e.g., LibCatch [47]) remain constrained by rigid templates. Consequently, they struggle with unseen semantic modifications, such as altered return types or complex inheritance constraints.

**2) Prompt-only LLM-based migration methods struggle with cascading multi-step repair.** While recent prompt-only LLM-based migration approaches demonstrate strong semantic comprehension, they typically formulate API migration as a *"stateless" transformation process*. However, in reality, dependency upgrade usually involves multi-step stateful transformations from the initially broken build (a client using a new dependency version involving breaking changes) to the successful upgraded project. For instance, resolving an initial compilation error caused by API return type change often alters the program's type environment, triggering a chain reaction of cascading errors across dependent files. Because such prompt-only pipelines lack continuous environmental feedback and state persistence capabilities, they cannot systematically determine whether a given transformation is genuine progress toward a successful upgrade or an incorrect fix that needs to rollback. Forced to either guess in the dark with limited snippets or process overwhelmingly large codebases at once, these models frequently hallucinate, accumulate invalid patches, and fail to converge to a globally successful state.

To address these limitations, we propose an automated API migration framework **DepUpgrade**, which models API migration as a *state-path exploration* problem. A state is a snapshot of the client codebase at a particular point during migration. Transitions between states are validated via compiler feedback and rolled back if regressions occur. Instead of performing a single-pass generation, DepUpgrade systematically explores a search space of intermediate states. Crucially, DepUpgrade evaluates the causal relationship between resolved errors and newly emerged errors. If new errors are structurally derived from the resolved ones, it recognizes the state as an advancement. This controlled exploration ensures steady convergence toward a successfully upgraded project.

DepUpgrade operationalizes this through three core mechanisms. First, it extracts migration knowledge directly from dependencies' source code before and after the upgrade, eliminating the need for manual migration examples. Second, it formulates bounded atomic repair tasks integrating local code structures, API facts, and compiler diagnostics, providing the LLM with precise context while preventing context explosion. Third, it incorporates a causality-guided evaluation loop to dynamically assess each state. By determining whether new compilation errors are causally linked to recent edits, DepUpgrade continuously validates structural progress and executes strategic rollbacks upon regressions, progressively resolving cascading errors.

We extensively evaluated DepUpgrade on public benchmarks and real-world projects. Experimental results demonstrate that DepUpgrade significantly outperforms state-of-the-art baselines. On a benchmark of 123 migration tasks, it achieves a full repair rate of 96.7%, compared to 49.6% for the best traditional baseline, LibCatch. On the real-world projects, DepUpgrade fully repairs 15 of the 18 projects, where existing tools mostly fail, while maintaining high semantic correctness.

Our main contributions are summarized as follows:

(1) We conceptualize API migration as a *state-path exploration* problem and present an LLM-based framework DepUpgrade to address it.
(2) We introduce state-aware causal feedback to systematically evaluate intermediate code states and control the LLM's exploration trajectory, overcoming cascading error challenges.
(3) Evaluations show that DepUpgrade outperforms existing SOTA baselines, achieving strong compilation-repair effectiveness across 18 real-world projects and maintaining high semantic correctness.

## 2 Motivating Example

We illustrate the challenges of library upgrades using a representative case from the project `greplin-lucene-utils`. The project depends on `lucene-core`. When upgrading this dependency from version `3.5.0` to `3.6.0`, a structural breaking change occurs: the `termPositions(Term)` method (with a parameter) in the base `IndexReader` class is marked as `final`, meaning it can no longer be overridden by subclasses.

As shown in Listing 1, `greplin-lucene-utils` contains an abstract class `FilteredIndexReader.java` that extends `IndexReader` and redeclares `termPositions(Term)` as `abstract`. Consequently, two concrete child classes inherit this class and override the parameterized method.

Following the upgrade, the compiler throws a "Cannot override the final method" error. To resolve this, the developer must systematically delete the invalid parameterized `termPositions(Term)` method across the abstract parent and both child classes. Instead, they must override the permitted parameterless `termPositions()` method to restore the original semantics.

Existing automated repair paradigms fail to resolve this cascading migration case for the following reasons:

- API-mapping approaches assume migrations are one-to-one symbol substitutions. They fail because they cannot find mapping rules for function modifier changes.
- Edit-script-based methods depend heavily on historical migration commits for specific version pairs. They fail because it is hard to find the corresponding migration commits or manual examples for this specific scenario.
- Compilation-error-repair methods rely on predefined repair templates. They fail because their predefined action spaces do not contain the corresponding operation required to resolve this specific `final` modifier conflict.
- Prompt-only LLM-based migration methods may fail because fixing the parent problem first reveals two more errors in child, so the intermediate state appears worse before it can become correct. This causes them to stop exploring the very repair path that leads to the correct solution.

```
1 // 1. FilteredIndexReader.java (Parent, extends IndexReader)
2 @Override
3 public abstract TermPositions termPositions(final Term term)
```

```
    throws IOException;
// 2. FilteredMultiReader.java (Child A)
@Override
public TermPositions termPositions(final Term term) throws
      IOException {
  TermPositions result = termPositions();
  result.seek(term);
  return result;
}
// 3. FilteredSegmentReader.java (Child B)
@Override
public TermPositions termPositions(final Term term) throws
      IOException {
  return new FilteredTermPositions(getUnderlyingReader().
      termPositions(term));
}
```

**Listing 1: Code Snippet Before Upgrade.**

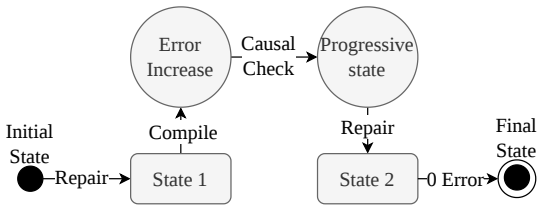



**Figure 1: DepUpgrade Repairing for motivating example.**

As illustrated in Figure 1, DepUpgrade successfully solves this complex migration. First, DepUpgrade retrieves the specific evolution evidence: the `termPositions` method has been marked as `final`. It begins by addressing the root of the hierarchy, rewriting the method in the parent class. Interestingly, after this initial parent patch, the total number of compilation errors *increases* because the fix exposes new, previously hidden errors in the child classes. While naive repair tools would view this temporary error spike as a failure and stop migration, DepUpgrade evaluates the causal relationship between the resolved error and the newly emerged errors. It analyzes the error signatures and recognizes that the initial patch successfully unlocked the next layer of hierarchical errors. By correctly recognizing this intermediate state as a progressive state, DepUpgrade retains the correct parent patch and proceeds to iteratively migrate the child classes along the optimal repair trajectory until the entire hierarchy compiles successfully.

## 3 Approach

This section presents the architecture and methodology of DepUpgrade. As illustrated in Figure 2, DepUpgrade decomposes the code migration process into three phases:

(1) **Phase I: Knowledge Extraction.** Extracts and categorizes raw API discrepancies from the target library versions. These extracted descriptors are projected into a dual-index architecture: a Vector Index to capture semantic intent and an Inverted Index to preserve exact lexical identities.
(2) **Phase II: Bounded Task Formulation.** Constructs a valid dependency graph and executes an initial compilation. The resulting global API breaking changes are then partitioned into independent file-level repair tasks, enriched with precise semantic context.
(3) **Phase III: State-Path Exploration.** Guided by hybrid retrieval from the knowledge base, an LLM generates schema-constrained patches. After applying the patch and recompiling, a causality-guided evaluation loop controls the state-path exploration. It accepts patches that are positive, while rolling back degraded patches and injecting negative feedback warnings into the next iteration.

### 3.1 Problem Formulation

*Inputs and outputs.* Let $P$ denote a client project under a baseline dependency version $v_{old}$. We formally define the upgrade specification as $\Delta : v_{old} \rightarrow v_{new}$, representing the transition of the target third-party library to a newer version $v_{new}$. Applying dependency upgrade exposes the client code to API breaking changes introduced by the new version. Consequently, the compilation process fails and yields a set of compilation errors, denoted as $E$. If $P$ already contains compilation errors under $v_{old}$, these pre-existing diagnostics are merged into the initial error set $E$ and processed within the same repair loop. Each individual compilation error $e \in E$ carries essential structural information: a file path, a precise location span, an error category (e.g., missing symbol, incompatible type, overridden `final` method), and an error message. The ultimate goal of DepUpgrade is to autonomously navigate the code state space to find an optimal repair trajectory $\Pi = \langle \pi_1, \pi_2, \ldots, \pi_k \rangle$ over the client code such that the upgraded project compiles successfully under $v_{new}$.

*Repair as a state-path exploration problem.* Conceptually, we treat the code migration process as a systematic state-path exploration that searches for an optimal sequence of patches to achieve successful dependency upgrade. Rather than naively minimizing the raw count of compilation errors—which fluctuates when hidden errors are exposed—we mathematically model this as finding a valid repair trajectory:

$$\text{Find} \quad \Pi = \langle \pi_1, \ldots, \pi_k \rangle \quad \text{s.t.} \quad \begin{cases} E(S_k) = \emptyset & \\ \Phi(S_{i-1}, S_i) \geq 0, & \forall i \in [1, k] \\ \pi_i \subseteq \mathcal{A}_i, & \forall i \in [1, k] \end{cases} \tag{1}$$

where $S_0 = P(v_{new})$ denotes the initial codebase state compiled under the new dependency version, and $S_i = S_{i-1} \oplus \pi_i$ represents the new codebase state after applying a code patch $\pi_i$ to the previous state. Here, $E(S_k) = \emptyset$ represents the ultimate goal of achieving a zero-error compilation state. The function $\Phi(S_{i-1}, S_i)$ serves as our causality-guided evaluation, which yields a non-negative value if the transition either reduces errors or structurally unlocks deeper ones (i.e., a structurally progressive state). Finally, $\mathcal{A}_i$ defines the *actionable semantic boundary* at step $i$. Instead of a rigid file whitelist, $\mathcal{A}_i$ represents the specific structural regions (e.g., enclosing method bodies or logically coupled class declarations) that surround the targeted errors. The constraint $\pi_i \subseteq \mathcal{A}_i$ ensures that the spatial footprint of each generated patch $\pi_i$ is strictly confined, preventing uncontrolled project-wide refactoring and guaranteeing that every repair step is causally traceable throughout the entire exploration trajectory.

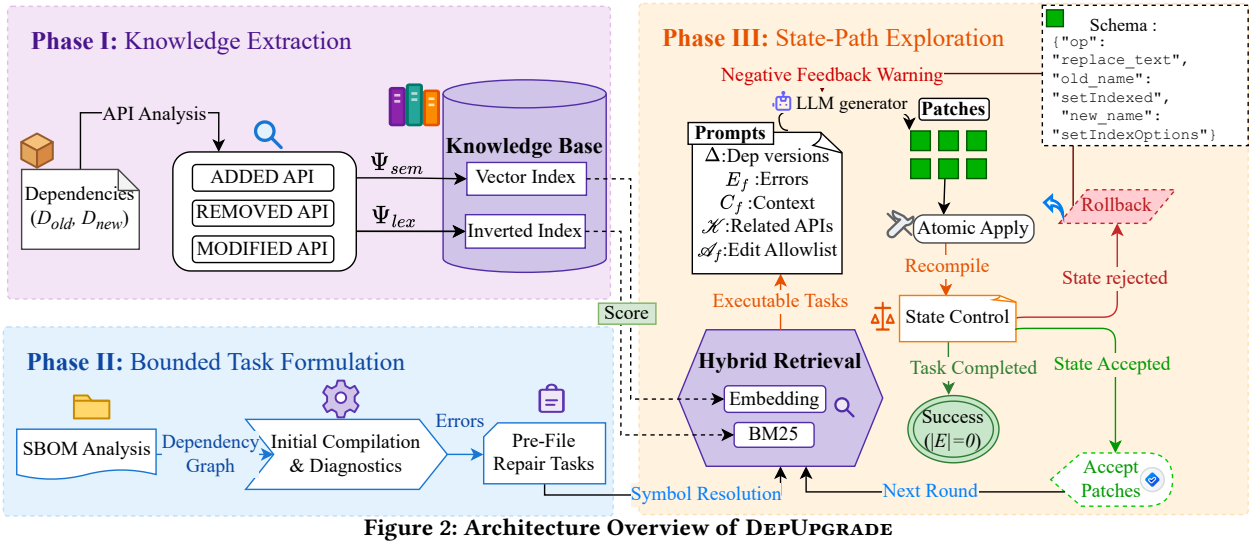



**Figure 2: Architecture Overview of DepUpgrade**

## 3.2 Knowledge Extraction

To provide the LLM with accurate information about how APIs have evolved, DepUpgrade establishes a Migration Knowledge Base, denoted as $\mathcal{K}$. Instead of relying on historical commits or manual documentation, we automatically construct this knowledge base by comparing the old and new versions of the third-party dependencies.

*Extracting API Differences.* Given the baseline dependencies $D_{old}$ and the target dependencies $D_{new}$, DepUpgrade first identifies the precise structural differences between them. We define a differential operator $\mathcal{T}_\Delta$ that compares the compiled files (such as `.jar` files) of both versions. It examines class signatures, method parameters, and access modifiers to detect what has been modified. The output of $\mathcal{T}_\Delta$ is a comprehensive set of API differences. Based on the type of modification, we partition these APIs into three distinct subsets:

$$\mathcal{T}_\Delta(D_{old}, D_{new}) = API_{add} \cup API_{rem} \cup API_{mod} \tag{2}$$

Here, $API_{add}$ contains newly introduced APIs, $API_{rem}$ contains removed APIs, and $API_{mod}$ contains APIs whose signatures or modifiers have been modified.

*Building the Dual-Index Knowledge Base.* A key insight of DepUpgrade is that different types of API modifications require different retrieval strategies. When client code fails to compile, the error message typically contains the name of an *old* API that was removed or modified. Because the exact name is present in the error, these APIs can be efficiently found using text matching. Conversely, the developer needs to know what *new* APIs to use as replacements. Since the new API names are unknown to the broken client code, finding them requires searching for semantic similarities.

To implement this insight, we group the removed and modified APIs together as $API_{old} = API_{rem} \cup API_{mod}$, and split the knowledge base $\mathcal{K}$ into two specialized indices:

$$\begin{aligned} \mathcal{K}_{lex} &= \{\Psi_{lex}(api) \mid api \in API_{old}\}, \\ \mathcal{K}_{sem} &= \{\Psi_{sem}(api) \mid api \in API_{add}\} \end{aligned} \tag{3}$$

This formulation defines a dual-index architecture:

- **Inverted Index ($\mathcal{K}_{lex}$):** The projection function $\Psi_{lex}$ maps the old APIs into a sparse keyword representation. This index is optimized for lexical keyword matching.
- **Vector Index ($\mathcal{K}_{sem}$):** The projection function $\Psi_{sem}$ maps the newly added APIs into a high-dimensional vector space. This index captures the abstract semantic intent of the new APIs.

*Two-Stream Context-Aware Retrieval.* Once $\mathcal{K}$ is built, it serves as the reference for repairing compilation errors. When generating the migration patch for a given compilation error $e$ at a specific location, we construct a textual query $q$ containing the error message and the broken code snippet. We also extract the local import statements $\mathcal{I}_{loc}$ enclosing the error to understand the code's context.

To retrieve the most relevant migration knowledge, DepUpgrade executes a two-stream retrieval process, selecting the top-$K$ candidates from both indices independently:

$$\mathcal{R}_{lex} = \underset{k \in \mathcal{K}_{lex}}{\text{Top-K}} \left[\mathbb{I}(k, \mathcal{I}_{loc}) \cdot \text{BM25}(q, k)\right] \tag{4}$$

$$\mathcal{R}_{sem} = \underset{k \in \mathcal{K}_{sem}}{\text{Top-K}} \left[\mathbb{I}(k, \mathcal{I}_{loc}) \cdot \text{sim}_{\cos}(q, k)\right] \tag{5}$$

where $\text{BM25}(q, k)$ calculates the exact keyword overlap and $\text{sim}_{\cos}(q, k)$ computes the cosine similarity in the vector space. $\mathbb{I}(k, \mathcal{I}_{loc}) \in \{0, 1\}$ is a strict indicator function preventing identically named classes in unrelated dependencies from causing confusion. This filter evaluates to 1 only if two contextual conditions are met: (1) the candidate $k$ comes from a dependency that is actually present in the project's valid dependency graph, and (2) the candidate's package namespace aligns with the client's local imports ($\mathcal{I}_{loc}$). If either condition fails, the indicator outputs 0, instantly nullifying the candidate's score and dropping it from the search results.

Finally, the retrieved evidence sets are combined:

$$\mathcal{K}_{top} = \mathcal{R}_{lex} \cup \mathcal{R}_{sem} \tag{6}$$

$\mathcal{R}_{lex}$ effectively pinpointing the old, broken APIs, while $\mathcal{R}_{sem}$ discovering newly added APIs that share semantic similarities with the query's intent. The combined set $\mathcal{K}_{top}$ provides the LLM with

a complete picture: why the old code broke and what new APIs are available to fix it.

*Connection to the motivating example.* In the `lucene-core` upgrade, the differential operator $\mathcal{T}_\Delta$ compares dependency binaries to detect the new `final` modifier on `termPositions(Term)`, indexing this change into $\mathcal{K}_{lex}$. When `FilteredIndexReader` fails due to an invalid override, DepUpgrade formulates a query $q$.The two-stream retrieval then leverages $\mathcal{R}_{lex}$ to match the exact method signature and $\mathcal{R}_{sem}$ to identify alternatives. This provides the LLM with the factual evidence of the `final` constraint, enabling it to synthesize a valid delegated implementation.

## 3.3 Bounded Task Formulation

Given an upgrade task $\Delta$, DepUpgrade compiles the client project under the new dependency version $v_{new}$. As expected, the API breaking changes cause this initial compilation to fail, yielding a global set of compilation errors, denoted as $E$.

Rather than feeding all errors to the LLM at once or fixing them strictly one by one, we partition the global error set by file:

$$E = \bigcup_{f \in \mathcal{F}} E_f, \quad E_f = \{e_{f,1}, e_{f,2}, \ldots, e_{f,n_f}\} \tag{7}$$

where $\mathcal{F}$ denotes the set of all affected source files in the project, $E_f$ denotes the subset of compilation errors associated with a specific file $f \in \mathcal{F}$, and $n_f$ denotes the number of compilation errors in $f$. Each affected file $f$ is therefore treated as a candidate repair unit.

To accurately repair the errors in $E_f$, the generative engine requires structural awareness. For each file $f$, we extract a bounded context slice $C_f$ to serve as the reasoning environment for the LLM. Specifically, $C_f$ includes: (i) necessary header information, (ii) the enclosing declarations directly surrounding each error location, and (iii) a syntactically expanded window of neighboring statements. For small files, $C_f$ may represent the entire file content; for larger files, it is dynamically bounded to prevent context explosion.

*Task definition.* Leveraging the information generated above, DepUpgrade synthesizes these components into a unified, atomic migration task $R_f$ for the target file:

$$R_f = \langle \Delta, E_f, C_f, \mathcal{K}, \mathcal{A}_f \rangle \tag{8}$$

Here, $\mathcal{K}$ denotes the dual-index migration knowledge base introduced in Section 3.2, which is not consumed directly as prompt content but serves as the retrieval source for constructing error-specific evidence during patch generation. $\mathcal{A}_f$ is the *actionable semantic boundary* introduced in Section 3.1. $A_f$ defines an actionable semantic boundary centered on $f$ and may include structurally coupled regions in other files when required. This atomic task formulation is the cornerstone of DepUpgrade's stability: it ensures a bounded reasoning context ($C_f$), provides hallucination-free factual guidance ($\mathcal{K}$), prevents unrelated project-wide modifications ($\mathcal{A}_f$), and guarantees that every generated patch can be causally attributed to a concrete subset of compiler diagnostics ($E_f$).

*Connection to the motivating example.* After the initial compilation, the global error set $E$ contains multiple failures across the inheritance chain due to the newly introduced `final` restriction on the `termPositions(Term)` method. DepUpgrade parses these diagnostics and partitions them, isolating the specific subset of errors $E_f$ occurring in the parent class file, `FilteredIndexReader.java`. It then extracts the context slice $C_f$, capturing the class header alongside the invalid `@Override` method declaration. The initial boundary is centered on the parent class and is expanded to structurally coupled child classes as the downstream errors are exposed. By encapsulating these elements alongside the retrieved evidence from $\mathcal{K}$ into a cohesive atomic task $R_f$, DepUpgrade provides the LLM with a highly focused reasoning environment, setting the stage to safely resolve the parent-level override issue before systematically progressing to the child classes.

## 3.4 State-Path Exploration

We model the migration process as a *state-path exploration* problem. Recall that a state is defined as a snapshot of the client codebase at a particular point in the API migration process. Formally, we define a transition function $G$ that maps a current state $S_t$ and a candidate patch $\pi_t$ to a next state $S_{t+1}$, contingent on a successful state-aware causal feedback check $\Phi$:

$$S_{t+1} = G(S_t, \pi_t) = \begin{cases} S_t \oplus \pi_t & \text{if } \Phi(S_t, S_t \oplus \pi_t) \geq 0 \\ S_t & \text{otherwise} \end{cases}$$

It ensures that migration is a controlled state-path exploration process: invalid patches are rejected and rolled back, while valid, causally-linked patches drive the migration forward. The following section will introduce how DepUpgrade generates the migration patches and how $G$ controls the state-path exploration.

*3.4.1 LLM-Driven Schema-Constrained Patch Synthesis.* DepUpgrade employs LLM as its reasoning engine for migration patch generation. For each compilation error $e \in E_f$ contained in the task $R_f$, DepUpgrade formulates a retrieval query $q$ and retrieves the top-ranked evidence $\mathcal{K}_{top}$ from the knowledge base $\mathcal{K}$. The LLM is then prompted with $\mathcal{K}_{top}$ together with the other information in $R_f$ to generate a migration patch, which may contain multiple migration operations targeting different code locations. Although each task is centered on a primary file, a generated patch may contain multiple edit operations across all structurally coupled regions included in $A_f$. To prevent the generation of unparsable code or hallucinations, we constrain the LLM's output space to a strict JSON schema, which defines the structure and allowed types of all localized edit operations, shown in Table 1.

Formally, the migration patch $\pi$ is generated as a structured set of operations, where each element $op_i \in \pi$ must satisfy the schema constraint and target only allowlisted regions:

$$\pi = \langle op_1, op_2, \ldots, op_n \rangle \text{ s.t. } \forall i,\ op_i \sim \mathcal{O} \wedge \text{Target}(op_i) \in \mathcal{A}_f \tag{9}$$

$\mathcal{O}$ denotes the operations defined in JSON schema, and $\mathcal{A}_f$ is the actionable semantic boundary. Specifically, the constraint $op_i \sim \mathcal{O}$ enforces structural validity, guaranteeing that every operation adheres to the predefined JSON schema, while $\text{Target}(op_i) \in \mathcal{A}_f$ restricts modifications strictly to the allowlist regions.

To mitigate issues arising from excessive prompt length, the LLM prompt is engineered with a strict minimality bias: it is instructed to generate the smallest set of edits that satisfy the API breaking changes in $E_f$ based only on the provided evidence. By strictly

**Table 1: Atomic Edit Operations under Schema Constraints.**

| Operation | Target | Description | JSON Schema Example |
|---|---|---|---|
| `replace text` | Code Pattern | Replaces an exact string pattern with a new replacement snippet. | `{"op": "replace_text", "pattern": "setIndexed(true)", "replacement": "..."}` |
| `delete text` | Code Pattern | Safely removes a specific obsolete code fragment matching the given pattern. | `{"op": "delete_text", "pattern": "import com.old.A;"}` |
| `insert_*` | Anchor Point | Inserts new code immediately before or after a specified anchor string. | `{"op": "insert_before", "anchor": "doc.add(f);", "text": "FieldType ft = ..."}` |
| `replace insert delete` | Line Range | Performs exact line-based edits (used only when line numbers and fragments are highly confident). | `{"op": "replace", "start_line": 10, "end_line": 12, "text": "newMethod()"}` |
| `import` | File Header | Handles the addition or modification of import statements to resolve dependencies. | `{"op": "import", "text": "import org.apache.lucene.document.TextField;"}` |

bounding both the input context and the output edit schema, the framework constrains the LLM's solution search space, thereby significantly reducing the likelihood of generating code that is syntactically correct but semantically flawed.

*3.4.2 Causal Error Analysis and Strategic Rollback.* After applying patch $\pi_t$ to transition from $S_t$ to $S_{t+1}$, DepUpgrade recompiles the project and compares the new diagnostics against the previous state $S_t$. It divides the compile errors into two sets: the resolved set $E_{disappeared}$ and the newly introduced set $E_{appeared}$. The causality-guided evaluation function $\Phi(S_t, S_{t+1})$ yields a non-negative value (i.e., the state is directly or tentatively accepted) under either of the following conditions:

- **Error Reduction:** The total number of compiler errors strictly decreases.
- **Structural Advancement:** The total error count does not decrease, but some errors are replaced and the newly introduced errors in $E_{appeared}$ are structurally or semantically derived from the resolved ones in $E_{disappeared}$.Specifically, the new and resolved errors must involve same failing API (method, class, or field), share same retrieved knowledge-base evidence, or belong to same file family through inheritance or polymorphism.

Such structural advancement reflects the cascading nature of API migration: one breaking change may induce a chain of dependent errors, and resolving the root constraint can expose previously hidden downstream errors. The structural relations indicate that the repair may have advanced along a correct migration path even when the total error count does not immediately decrease. Otherwise, the transition is regarded as a regression and rolled back.

The overall migration process of DepUpgrade is shown in Algorithm 1. If the transition is accepted, DepUpgrade structurally advances the workspace to the new state $S_{t+1}$. A patch that directly reduces the error count is committed to the global patch set $\Pi$. In contrast, structural advancement is treated as conditional unlocking and accepted only tentatively. Within the next three rounds, the exploration path must produce a net reduction in compilation errors; otherwise, the path is rolled back and logged as negative feedback. After that, the system clears any negative feedback history from previous attempts and proceeds to the next iteration. Conversely, if the transition is rejected or a conditionally unlocked path fails to reduce the error count within three rounds, DepUpgrade performs an atomic rollback to $S_t$ to preserve code integrity. To avoid blind trial-and-error loops, it synthesizes a structured negative feedback prompt. This prompt explicitly details the specific compiler diagnostics introduced or left unresolved by the rejected patch. This semantic feedback is then injected into the LLM's context window to heavily guide the reasoning engine toward a refined, corrected patch in the subsequent attempt. This state-driven iteration continues until one of two termination criteria is met: (1) global success, where the total compilation error count drops to zero, signifying that the project compiles flawlessly under $v_{new}$; or (2) budget exhaustion, where the system reaches a predefined maximum iteration limit $N$ and returns the best-effort valid state found so far. Ultimately, this mechanism establishes a robust state-path exploration loop, transforming the unpredictable nature of LLM generation into a controlled trajectory toward a valid build state.

**Algorithm 1** Patch Generation and State-Path Exploration Loop

**Input:** Project $P$, upgrade $\Delta$, knowledge base $\mathcal{K}$, budget $N$
**Output:** Accepted patch set $\Pi$; remaining diagnostics $E^\star$

$S \leftarrow$ Materialize$(P, \Delta)$; $E \leftarrow$ Compile$(S)$
$\Pi \leftarrow \emptyset$; $\mathcal{H} \leftarrow \emptyset$ ▷ history for regression measure
**for** $t \leftarrow 1$ **to** $N$ **do**
  **if** $|E| = 0$ **then**
    **break**
  **end if**
  Choose a file-task $R_f$ induced by current $E$
  Extract local imports $\mathcal{I}_{loc}$ and formulate query $q$ from $R_f$
  $\mathcal{K}_{top} \leftarrow$ TwoStreamRetrieve$(\mathcal{K}, q, \mathcal{I}_{loc})$
  $\pi \leftarrow$ LLM_ProposePatch$(R_f, \mathcal{K}_{top}$, Feedback$)$ ▷ schema-constrained
  $S' \leftarrow$ ApplyAtomic$(S, \pi)$; $E' \leftarrow$ Compile$(S')$
  Compute structural diff $(E_{disappeared}, E_{appeared})$
  **if** IsImproved$(E, E')$ **or** UnlockingLeadsToReduction$(S', E', 3)$ **then**
    $S \leftarrow S'$; $E \leftarrow E'$
    $\Pi \leftarrow \Pi \cup \{\pi\}$; $\mathcal{H} \leftarrow \mathcal{H} \cup$ Sig$(E)$
    Feedback $\leftarrow \emptyset$ ▷ Clear feedback on success
  **else**
    Rollback$(S)$
    Feedback $\leftarrow$ SynthesisFeedback$(\pi, E')$
  **end if**
**end for**
$E^\star \leftarrow E$; **return** $(\Pi, E^\star)$

*End-to-end resolution of the motivating example.* By querying $\mathcal{K}$, DepUpgrade retrieves the relevant API change and guides the LLM to synthesize a schema-constrained patch. The patch removes the invalid override using the `replace_text` operation and safely delegates to the newly permitted `termPositions()` method. Although removing the invalid method triggers new errors in downstream child classes, DepUpgrade recognizes this as a *Progressive* state: the root structural mismatch is resolved despite the increased error count. The system then accepts the state, clears the feedback history, and iteratively resolves the newly exposed child-class tasks to achieve global convergence.

# 4 Evaluation

We conduct evaluations to answer the research questions:

- RQ1: How does DepUpgrade compare to other baselines, and does it preserve semantic correctness?
- RQ2: How do individual components and hyperparameters influence the effectiveness of DepUpgrade?
- RQ3: How effective is DepUpgrade in real-world API migrations?

## 4.1 Experimental Settings

*4.1.1 Benchmark.* To evaluate DepUpgrade, we utilize the datasets provided by LibCatch [47]. For RQ1 and RQ2, the benchmark dataset comprises 123 migration version tasks from five popular libraries. Each task represents an end-to-end project upgrade for a specific source–target dependency version pair. The benchmark covers 376 library versions, 827 class APIs, 1,189 field APIs, and 1,570 method APIs, providing diverse and complex migration scenarios for evaluating DepUpgrade. For RQ3, which focuses on real-world applicability, we evaluate 18 real-world projects, including 10 originally released alongside LibCatch and eight additional projects collected from GitHub.

*4.1.2 Baselines.* In order to make fair comparison, we choose these tools as our baselines:

- SemDiff [7]: a widely used API mapping approach.
- Meditor [45]: an edit-script-based tool.
- LibCatch [47]: a compilation-error repair tool that remains state of the art in this area among non-LLM approaches.
- First Results (FR) [1]: an LLM-based approach that relies on prompt engineering for migration. Since FR requires migration examples for its one-shot settings, which are unavailable in our zero-shot evaluation context, we follow the methodology of prior work [19] and use its Chain-of-Thought (CoT) prompt template for evaluation.
- Codex: a modern coding-agent. For each task, we provide Codex with the task description and the corresponding project repository, and allow it to autonomously inspect, edit, compile, and iterate using its default tools and permissions without manual intervention.

For RQ1, we exclude SemDiff and Meditor from the benchmark comparison. Specifically, the source code and artifacts of SemDiff have been removed from its official website and public repositories, making the tool inaccessible for reproduction. Meditor fundamentally relies on the availability of historical migration commits or scripts. Since such historical data is absent for the migration tasks in our benchmark, Meditor is not applicable in our evaluation setting. Consequently, for this research question, we restrict the comparative analysis to LibCatch , FR, and Codex.

*4.1.3 Metrics.* Following these established practices [9, 10, 29, 47, 48], we use compilation error rate as the primary metric for comparing DepUpgrade with baseline tools, and semantic equivalence as a complementary metric for assessing correctness beyond compilation. However, because most baseline tools are closed-source and do not support semantic validation, a unified comparison over both metrics is not feasible. We therefore adopt the following evaluation strategy across our research questions:

- *Compilation errors*: This metric is evaluated across all three RQs.
- *Semantic equivalence*: In RQ1 and RQ3, we cross-review migration patches for semantic correctness, adhering to best practices [10, 47].

*4.1.4 Implementation and Hyperparameter Settings.* The structural differential operator ($\mathcal{T}_\Delta$) is built on Eclipse JDT AST parser [37], while the compilation oracle is implemented with the Maven API.

We evaluate four LLMs as the reasoning engine of DepUpgrade: `gemini-3.1-pro-preview`, `deepseek-v3.2`, `deepseek-v4-flash`, and `gpt-5.4`, with `gpt-5.4` as the default model. We use the same decoding settings for all models, with *temperature* set to 0 and *top-p* set to 0.95.

For migration knowledge retrieval, we construct a hybrid index consisting of a vector index built with `text-embedding-v4(qwen)` and an inverted index based on TF-IDF/BM25. During retrieval, the system selects the top-$K$ candidates from both the lexical and semantic streams, with $K = 4$ for each stream. For state-path exploration, the maximum search budget is set to $N = 5$ iterations. Each iteration consists of patch generation, patch application, project compilation, and state evaluation, and the budget is applied independently to each dependency-upgrade task. All experiments are conducted in a Linux environment on a machine with 8 GB RAM.

## 4.2 RQ1. Comparison to Baselines

In this section, we evaluate DepUpgrade on the latest open-source benchmark against several baseline tools.

*4.2.1 Experimental Results.* The baseline comparison results are presented in Table 2. In the table, the number in parentheses after each library name denotes the total number of tasks in that family. The symbols ↓ and ✓ denote the numbers of tasks whose compilation errors are reduced and eliminated, respectively, while % denotes the proportion of tasks with all compilation errors eliminated. DepUpgrade and the LLM-based baselines FR and Codex are configured to utilize `gpt-5.4`. Since LibCatch is not open-source, we use the official results reported in their original paper [47].

**Table 2: Migration results compared with baseline tools (using GPT-5.4 as the backbone LLM)**

| Tool | Accumulo (24) | Cassandra (72) | Karaf (3) | Lucene (13) | POI (11) | Total (123) | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | ↓ | ✓ | % |
| LibCatch | ↓ 23/✓ 11 | ↓ 71/✓ 43 | ↓ 2/✓ 1 | ↓ 9/✓ 2 | ↓ 9/✓ 4 | 114 | 61 | 49.59% |
| FR | ↓ 20/✓ 15 | ↓ 12/✓ 0 | ↓ 2/✓ 2 | ↓ 8/✓ 4 | ↓ 11/✓ 2 | 53 | 23 | 18.70% |
| Codex | ↓ 22/✓ 17 | ↓ 10/✓ 0 | ↓ 1/✓ 1 | ↓ 8/✓ 7 | ↓ 11/✓ 5 | 52 | 30 | 24.39% |
| DepUpgrade | ↓ 24/✓ 23 | ↓ 72/✓ 72 | ↓ 3/✓ 3 | ↓ 13/✓ 12 | ↓ 11/✓ 9 | **123** | **119** | **96.75%** |

As shown in the table, DepUpgrade successfully eliminates all compilation errors in 119 out of 123 migration tasks, achieving an unprecedented full repair rate of 96.7%. Moreover, Codex fully repairs 30 tasks and FR repairs 23, both trailing LibCatch's 61 and remaining substantially below DepUpgrade's 119. Under the same `gpt-5.4` backbone, Codex's iterative agentic workflow outperforms FR's prompt-only repair by 7 tasks. However, neither Codex nor FR explicitly analyzes API evolution to determine the required code changes. Moreover, Codex tends to address successive compilation errors by layering new edits over previous states rather

than reverting ineffective patches. As ineffective or even regressive modifications accumulate, the repair process may eventually enter repetitive cycles without resolving the underlying incompatibility. The superiority of DepUpgrade is particularly evident in structurally complex projects. For instance, in `cassandra`, which contains the largest number of migration tasks (72), DepUpgrade achieved a flawless 100% success rate, compared to LibCatch's 59.7%. Similarly, in `lucene`, DepUpgrade boosted the full repair rate from a mere 15.4% to an impressive 92.3%.

Beyond completely resolving errors, DepUpgrade exhibits stability during the migration process. It successfully reduced the number of compilation errors in 100% of the tasks. In contrast, LibCatch failed to reduce errors in 9 tasks, FR failed to reduce errors in 70 tasks, and Codex failed to reduce errors in 71 tasks. This finding confirms that even in the rare instances where DepUpgrade cannot completely eliminate every single error within the budget limit, it consistently produces a final state with fewer compilation errors than the initial upgraded project.

**Table 3: Effectiveness of DepUpgrade with different LLMs.**

| Model | Accumulo (24) | Cassandra (72) | Karaf (3) | Lucene (13) | POI (11) | Total (123) | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | ↓ | ✓ | % |
| GPT-5.4 | ↓ 24/✓23 | ↓ 72/✓72 | ↓ 3/✓3 | ↓ 13/✓12 | ↓ 11/✓9 | **123** | 119 | 96.75% |
| Gemini-3.1 | ↓ 24/✓24 | ↓ 71/✓70 | ↓ 3/✓3 | ↓ 13/✓13 | ↓ 11/✓11 | 122 | **121** | **98.37%** |
| DeepSeek-v3.2 | ↓ 7/✓5 | ↓ 70/✓66 | ↓ 2/✓2 | ↓ 10/✓7 | ↓ 9/✓6 | 98 | 86 | 69.92% |
| DeepSeek-v4 | ↓ 24/✓23 | ↓ 70/✓66 | ↓ 3/✓3 | ↓ 12/✓10 | ↓ 10/✓9 | 119 | 111 | 90.24% |

As DepUpgrade utilizes an LLM to generate migration patches, we evaluate its performance across various models. Table 3 shows that the effectiveness of DepUpgrade varies across backbone LLMs. Gemini-3.1-pro-preview achieves the highest full repair rate, eliminating all compilation errors in 121 tasks, followed by GPT-5.4 with 119 tasks, DeepSeek-v4-flash with 111 tasks, and DeepSeek-v3.2 with 86 tasks. Nevertheless, all four models reduce compilation errors in at least 98 tasks and completely eliminate them in at least 86 tasks, demonstrating that DepUpgrade maintains stable effectiveness across different LLMs while benefiting from stronger reasoning models.

We further report the average cost per migration task, following the methodology of prior work [1]. We conduct this comparison on all 123 migration tasks in the entire benchmark. Table 4 illustrates the token usage and cost differences among the evaluated models and tools. Costs are estimated based on the official API pricing as of March 2026 [8, 11, 33]. Among the four backbone models used by DepUpgrade, Gemini-3.1-pro-preview incurs the highest average cost at $0.1646 per task, followed by GPT-5.4 at $0.1015, DeepSeek-v3.2 at $0.0211, and DeepSeek-v4-flash at $0.0104. Under the same `gpt-5.4` backend, DepUpgrade consumes 3,406,576 LLM tokens at an average cost of $0.1015 per task, whereas Codex consumes 27,114,694 tokens and costs $0.3558 per task. Thus, DepUpgrade uses substantially fewer tokens and incurs a lower cost than Codex. FR consumes 727,845 tokens and costs $0.0366 per task. Although its absolute cost per attempted task is lower because FR performs only one repair round, it amounts to 36.1% of DepUpgrade's per-task cost despite using only one fifth of the maximum iteration budget. When normalized by the number of fully repaired tasks, the average costs are approximately $0.105 for DepUpgrade, $0.196 for FR, and $1.459 for Codex. Both FR and Codex incur a higher cost per fully repaired task than DepUpgrade.

**Table 4: Token usage over all 123 migration tasks.**

| Model | Tool | Embedding | LLM Tok. | Cost(USD)/task |
|---|---|---|---|---|
| GPT-5.4 | DepUpgrade | 39,959,408 | 3,406,576 | $0.1015 |
| | Codex | - | 27,114,694 | $0.3558 |
| | FR | - | 727,845 | $0.0366 |
| DeepSeek-v4 | DepUpgrade | 18,870,415 | 4,695,681 | $0.0104 |
| Gemini-3.1 | DepUpgrade | 35,917,765 | 2,872,697 | $0.1646 |
| DeepSeek-v3.2 | DepUpgrade | 55,905,899 | 5,072,468 | $0.0211 |

*4.2.2 Preservation of Semantic Correctness.* We conducted an expert review to evaluate the semantic equivalence of the patches generated by DepUpgrade with the default `gpt-5.4` backend on the 119 successfully compiled migration tasks. Two reviewers independently inspected each patch set and resolved disagreements through discussion. Based on cross-reviews, as summarized in Table 5, we classify the results into three categories:

**Table 5: Semantic correctness of 119 compilable patches.**

| Category | Tasks | Percentage |
|---|---|---|
| Semantically Equivalent | 62 | 52.10% |
| Compilable but Suboptimal | 40 | 33.61% |
| Semantically Incorrect | 17 | 14.29% |
| **Total** | **119** | **100.00%** |

- **Semantically Equivalent (62 tasks):** In over half of the cases, DepUpgrade successfully migrated the code while perfectly preserving the original semantics. The generated patches in this category are highly idiomatic and consistent with the solutions recommended by domain experts.
- **Compilable but Suboptimal (40 tasks):** In these tasks, DepUpgrade successfully eliminated all compilation errors, but the resulting patches may preserve the intended behavior only under specific usage conditions and introduce maintainability, robustness, or boundary-condition risks. For example, in `SearchFiles.java` (migrating `lucene` from `2.9.4` to `3.0.0`), DepUpgrade dynamically passes `Version.LUCENE_CURRENT` as a parameter, while human experts specify an exact static version. The patch generated by DepUpgrade is semantically equivalent with human patch under the observed version. However, we cannot make sure it is still equivalent when `SearchFiles.java` evolves in the future, e.g., changing `Version.LUCENE_CURRENT`'s value. In another instance within the same file (migrating from `6.6.4` to `7.0.0`), DepUpgrade explicitly casts the `long` return value of `search(query, numTotalHits)` to an `Integer`, which may introduce boundary-related issues in numerical operations. While this compiles and is correct under our observed scenarios, it poses a risk of integer overflow if the returned hit count exceeds `Integer.MAX_VALUE`.
- **Semantically Incorrect (17 tasks):** In a minority of cases, DepUpgrade generates patches that pass the compiler but violate the original program semantics. For example, when upgrading `lucene` from `3.6.2` to `4.0.0` in `CollationKeyAnalyzer.java`, DepUpgrade replaces a deprecated `KeywordTokenizer` with a rudimentary anonymous `Tokenizer` implementation. Although

this resolves the compilation errors, it drops the core tokenization logic, rendering the analyzer behaviorally incorrect.

In summary, among the 119 compile-passing tasks, 62 are semantically equivalent. Another 40 may preserve the intended behavior under certain usage scenarios but cannot be regarded as fully equivalent because of maintainability, robustness, or boundary-condition risks, while the remaining 17 are semantically incorrect. These results show that compilation success alone is insufficient evidence of semantic equivalence.

**Answer to RQ1.** The experimental results demonstrate that DepUpgrade significantly outperforms both traditional and LLM-based baselines, while exhibiting a promising capability to preserve semantic correctness.

## 4.3 RQ2. Ablation Study and Hyperparameter Analysis

To deeply understand the contribution of individual components (Knowledge Retrieval $\mathcal{K}$ and State-Path Exploration $\mathcal{SPE}$) and the influence of the iteration budget ($N$), we conduct detailed analyses. All experiments in this section are conducted on the complete benchmark using the default `gpt-5.4` backend. We evaluate $N \in \{1, 3, 5, 7, 9\}$. The w/o $\mathcal{K}$ variant disables migration-knowledge retrieval, while the w/o $\mathcal{SPE}$ variant disables state-path exploration. The w/o Both variant retains iterative patch generation and compilation feedback but disables both components, thereby serving as the naive-iteration baseline under the same model and iteration budget as the complete configuration.

**Table 6: Ablation study and iteration budget ($N$) analysis.**

| $N$ | DepUpgrade | | | w/o $\mathcal{K}$ | | | w/o $\mathcal{SPE}$ | | | Naive Iter. (w/o Both) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ↓ | ✓ | % | ↓ | ✓ | % | ↓ | ✓ | % | ↓ | ✓ | % |
| $N=1$ | 94 | 37 | 30.1% | 74 | 34 | 27.6% | 94 | 37 | 30.1% | 74 | 34 | 27.6% |
| $N=3$ | 123 | 104 | 84.6% | 121 | 94 | 76.4% | 115 | 85 | 69.1% | 113 | 75 | 61.0% |
| $N=5$ | 123 | 119 | 96.7% | 123 | 103 | 83.7% | 121 | 96 | 78.0% | 119 | 94 | 76.4% |
| $N=7$ | 123 | 119 | 96.7% | 121 | 104 | 84.6% | 121 | 96 | 78.0% | 119 | 94 | 76.4% |
| $N=9$ | 123 | 120 | 97.6% | 123 | 104 | 84.6% | 121 | 96 | 78.0% | 119 | 94 | 76.4% |

*4.3.1 Impact of Iteration Budget ($N$).* Table 6 shows that repair effectiveness improves rapidly as N increases to 5 and then largely saturates. At N=5, the complete configuration repairs 119 tasks, compared with 94 for naive iteration, demonstrating the combined contribution of $\mathcal{K}$ and $\mathcal{SPE}$. Increasing N from 5 to 9 yields only one additional repair; therefore, we use N=5 as the default.

*4.3.2 Ablation of Core Components.*

*Ablation on $\mathcal{K}$.* Removing $\mathcal{K}$ reduces the number of fully repaired cases from 119/123 (96.7%) to 103/123 (83.7%), demonstrating the importance of explicit API-evolution evidence. Although the numerical drop might appear modest, this is because a large portion of API breaking changes involve relatively straightforward updates (e.g., simple method renaming) that an advanced LLM can often repair correctly based on its pre-trained parametric memory. However, for complex and uncommon API structural changes, lacking the deterministic evolutionary evidence provided by $\mathcal{K}$ forces the LLM into severe hallucinations, fabricating non-existent API endpoints or incorrect parameters.

**Table 7: Conditionally accepted unlocking transitions.**

| Model | Accepted | Reduced | Rolled Back | Misleading |
|---|---|---|---|---|
| GPT-5.4 | 168 | 139 | 6 | 23 |
| Gemini-3.1 Pro Preview | 29 | 22 | 0 | 7 |
| DeepSeek-V3.2 | 205 | 126 | 34 | 45 |
| DeepSeek-V4 Flash | 155 | 125 | 7 | 23 |
| **Total** | **557** | **412** | **47** | **98** |

Accepted denotes conditionally accepted unlocking transitions; Reduced denotes transitions followed by a net error reduction within three rounds. The three columns are mutually exclusive.

For example, when upgrading Cassandra from 2.2.12 to 3.0.0-alpha1, the removed `decorateKey` call in `InvertedIndex.java` must be migrated:

```
// Original
DecoratedKey key =
    metadata.decorateKey(cell.value());

// Without K
DecoratedKey key =
    SinglePartitionNamesCommand
        .fullPartitionRead(metadata,
            System.currentTimeMillis(),
            cell.value())
        .partitionKey();

// With K
DecoratedKey key = update.partitionKey();
```

Without $\mathcal{K}$, the inferred replacement incorrectly reuses `cell.value()` and introduces a type incompatibility. With $\mathcal{K}$, DepUpgrade retrieves the corresponding evolution evidence and uses `update.partitionKey()`, preserving the intended semantics.

*Ablation on $\mathcal{SPE}$.* Disabling State-Path Exploration leads to a more substantial performance degradation, suggesting that $\mathcal{SPE}$ is more critical to DepUpgrade's effectiveness, as expected. The configuration without $\mathcal{SPE}$ struggles with complex, cascading errors. Without $\mathcal{SPE}$, it accumulates faulty code edits, eventually failing to reach a globally valid build state. This demonstrates that while most tasks may only require shallow reasoning, the robust state-path exploration is critical for resolving the most intricate, deeply nested structural mismatches in real-world projects.

Table 7 quantifies conditional unlocking across four complete runs of the 123-task benchmark. Among the 557 conditionally accepted transitions, 412 (74.0%) lead to a net reduction in compilation errors within three rounds, while 47 (8.4%) fail to make progress and are rolled back. The remaining 98 transitions (17.6%) are later identified as misleading because, although they satisfy the structural unlocking rule, they do not contribute to a valid error-reduction trajectory. These results show that structural relations provide useful guidance in most cases, while the bounded verification and rollback mechanism remains necessary to contain inaccurate decisions.

To illustrate the importance of $\mathcal{SPE}$, we examine a specific migration scenario from the benchmark. While Section 2 presents a scenario where newly appeared errors are structurally linked to the resolved errors, this case study demonstrates the opposite: how $\mathcal{SPE}$ prevents the system from getting trapped in a cycle of erroneous edits when an incorrect patch introduces new bugs.

In the `cassandra` benchmark file `WordCount.java`, there is a class declaration that should remain unaffected by the API upgrade:

```
public class HadoopWordCount extends Configured implements Tool
```

**Table 8: Repair effectiveness on real-world projects.**

| Project | #Det. | Ver. | LC | SD | Med. | FR | Codex | NI | ours |
|---|---|---|---|---|---|---|---|---|---|
| explicit-semantic-analysis | 23 | 4.8.1→5.0.0 | 0 | 23 | 23 | 16 | 9 | 0 | 0 |
| flea-db | 6 | 4.10.3→5.0.0 | 0 | 4 | 6 | 4 | 6 | 0 | 0 |
| IKAnalyzer | 7 | 4.10.0→5.0.0 | 0 | 7 | 7 | 5 | 2 | 0 | 0 |
| stemmer-sk | 2 | 5.3.1→6.3.0 | 0 | 2 | 2 | 2 | 2 | 1 | 0 |
| lucene-generic-highlighter | 3 | 3.5.0→4.0.0 | 0 | 3 | 3 | 3 | 1 | 0 | 0 |
| lqt | 2 | 6.1.0→7.0.0 | 1 | 5 | 5 | 1 | 3 | 0 | 0 |
| greplin-lucene-utils | 17 | 3.5.0→3.6.0 | 1 | 17 | 17 | 14 | 14 | 2 | 0 |
| LuceneQueryExpansion | 18 | 4.8.1→5.0.0 | 3 | 18 | 18 | 15 | 1 | 0 | 0 |
| lucene-interval-fields | 35 | 3.5.0→4.0.0 | 9 | 35 | 35 | 29 | 0 | 0 | 0 |
| marple | 37 | 6.5.1→7.0.0 | 21 | 37 | 37 | 26 | 14 | 0 | 0 |
| Hetty | 208 | - | - | - | - | 132 | 141 | 80 | 27 |
| kafka-sink-graphdb | 4 | - | - | - | - | 0 | 0 | 0 | 0 |
| mesa-desktop | 47 | - | - | - | - | 6 | 0 | 14 | 2 |
| pg-jsonb | 200 | - | - | - | - | 200 | 56 | 3 | 0 |
| slack-api | 92 | - | - | - | - | 6 | 6 | 0 | 0 |
| st-js-server (server) | 238 | - | - | - | - | 7 | 3 | 0 | 0 |
| stripe-api-java | 85 | - | - | - | - | 5 | 6 | 6 | 0 |
| zenvia-sdk-java | 61 | - | - | - | - | 19 | 11 | 4 | 2 |
| **Total** | 1085 | - | - | - | - | 490 | 275 | 110 | 31 |

#Det.: initial errors; Ver.: Lucene version upgrade; LC/SD/Med./NI: LibCatch/SemDiff/Meditor/Naive Iteration; Values denote remaining errors.

When attempting to fix other API errors during the upgrade from `2.2.12` to `3.0.0-alpha1`, an LLM hallucination incorrectly alters this class name to `WordCount`. This reckless change immediately triggers three new `already defined` compilation errors elsewhere in the project.

Without $\mathcal{SPE}$, DepUpgrade attempts to "fix" the new errors by making further absurd modifications, such as randomly deleting the `public` access modifier. When that predictably fails, it blindly adds `public` back, entering a repetitive cycle of meaningless trial-and-error edits until the iteration budget is entirely exhausted. In contrast, with $\mathcal{SPE}$ enabled, DepUpgrade instantly detects the sharp increase in compilation errors and identifies that the recent patch did not unlock a valid transitional state, but rather introduced regressions. Guided by causal feedback, DepUpgrade rejects the faulty path and rolls back, inferring that the correct action is to revert the erroneous class name change. This strictly prevents error accumulation and keeps the search trajectory on the correct path.

**Answer to RQ2.** The experimental results demonstrate that with Knowledge Retrieval and State-Path Exploration, DepUpgrade performs significantly better and successfully repairs complex tasks. Furthermore, with a suitable setting of repair rounds, DepUpgrade can achieve highly effective performance.

## 4.4 RQ3. Performance in Real-World Migrations

In this section, we evaluate DepUpgrade by migrating a set of real-world projects.

*4.4.1 Experimental Results.* Table 8 presents the performance of DepUpgrade on real-world migration tasks compared to other baselines across the ten Lucene projects. Moreover, we further perform experiment on eight projects randomly sampled from our collected Maven projects, covering Netty, HttpComponents, Hibernate, and Jackson upgrades. Together, the two project sets contain 18 projects and 1,085 initial JDT compilation errors. For SemDiff, Meditor, and LibCatch, we use the results provided by LibCatch [47], as our evaluation on Lucene follows the exact same projects and target upgrade versions. FR, Codex, naive iteration and DepUpgrade are evaluated on all 18 projects using `gpt-5.4` with $N = 5(N_{\text{FR}} = 1)$.

On the ten Lucene projects, DepUpgrade eliminate all 150 initial compilation errors. Among the baselines, naive iteration fully repairs eight projects, LibCatch repairs five, and Codex repairs one, whereas FR, SemDiff, and Meditor do not fully repair any project. Across all 18 projects, DepUpgrade leaves the 31 remaining errors and fully repairs 15 projects, compared with 110 for naive iteration, 275 for Codex, and 490 for FR.

Specifically, DepUpgrade excels at resolving complex structural mismatches that traditional methods abandon. To illustrate this, we examine a case from the `greplin-lucene-utils` project, which LibCatch documented as an unresolved challenge in their paper. The project contains the following code extending `IndexReader`:

```
@Override
public ... getFieldNames(final FieldInfos fldOption) {
    return this.delegate.getFieldNames(fldOption);
}
```

When upgrading `lucene` from `3.5.0` to `3.6.0`, the `getFieldNames` method is entirely removed from the parent `IndexReader` class. Traditional tools like LibCatch rely on rigid compilation-error matching templates. In this case, LibCatch's template detects that the `@Override` annotation causes an error and removes the annotation. However, the method body still attempts to call the now-deleted `delegate.getFieldNames(fldOption)`, leaving a compilation error that LibCatch cannot resolve because it lacks the capacity to synthesize new API mappings for the method body.

In contrast, DepUpgrade resolves this issue effortlessly. Through knowledge retrieval, it identifies that the method has been removed in the new `lucene` architecture. Rather than blindly stripping the annotations, DepUpgrade directly deletes the entire method, which is consistent with the manually validated repair reported in the LibCatch paper.

Similarly, in another failure case documented by LibCatch from the `LuceneQueryExpansion` project:

```
Directory dir = DirectoryReader.open(index);
```

Upgrading from `4.8.1` to `5.0.0` changes the input type and return behavior of `open()`. LibCatch cannot compose the required transformations, whereas DepUpgrade retrieves the relevant evolution knowledge and generates:

```
Directory dir = DirectoryReader.open(new SimpleFSDirectory(
      index.toPath())).directory();
```

These cases show that combining explicit evolution knowledge with generative reasoning enables DepUpgrade to handle structural migrations beyond rigid heuristics and edit scripts.

*4.4.2 Semantic Correctness.* Three LLM-based evaluators—`GPT-5.6 Sol`, `Claude Opus 4.8`, and `DeepSeek-V4-Pro`—independently classify all 1,054 successful error-repair patches produced by DepUpgrade across the 18 projects. Disagreements are adjudicated by `Claude Fable 5` using the patch and the three assessments. Overall, 816 patches (77.4%) are semantically equivalent, 57 (5.4%) are compilable but suboptimal with minor behavioral changes, and 181 (17.2%) are semantically incorrect. Thus, most successful repairs preserve intended behavior, but compilation success does not guarantee semantic correctness.

**Answer to RQ3.** DepUpgrade achieves strong compilation-repair effectiveness on real-world upgrades, outperforming the evaluated baselines through knowledge retrieval and state-path exploration.

## 5 Threats to Validity

The primary threat to the external validity of our evaluation lies in the scope of the programming languages tested. Currently, our experiments and benchmarks are exclusively conducted on Java projects. Consequently, the performance of DepUpgrade on dynamically typed languages or other statically typed languages remains empirically unverified. However, the core design of DepUpgrade is fundamentally language-agnostic. As long as a target ecosystem provides determinable compilation errors and access to package evolution history, the framework can be seamlessly adapted. Future work will extend DepUpgrade to support a broader range of programming languages and package managers to fully demonstrate its ability. The benchmark may partially overlap with the training data of the evaluated LLMs, since some projects and dependency versions predate their knowledge cutoffs. Although all LLM-based approaches are evaluated under comparable settings, potential data leakage cannot be completely excluded. Finally, compilation success does not necessarily imply semantic correctness. We therefore assess compile-passing patches, but such assessment may remain subjective, particularly when behavioral differences arise only under unobserved inputs or boundary conditions.

## 6 Related Work

Researchers have proposed various approaches to migrate client code in order to repair API-breaking changes. These works can be broadly categorized into four main methods.

**API-Mapping-Based Methods.** These methods focus on mining correspondences between deprecated and updated APIs by analyzing source code changes and method invocation patterns across library versions [4, 5, 7, 16, 22, 27, 41, 43, 44]. Besides inferring mappings in the same language, researchers also mine API mappings across different programming languages [29–31, 48]. While effective for simple syntactic replacements such as renames, these techniques inherently struggle with semantic and structural modifications.

**Edit-Script-Based Methods.** These approaches infer transformation rules or edit scripts from upgrade examples found in migration commits or documentation [2, 6, 9, 10, 12–15, 17, 24–26, 28, 32, 34, 35, 39, 45]. However, their applicability is severely limited by data scarcity. Upgrade examples are often unavailable, and commits in code repositories rarely provide precise version-to-version mappings. Even in large-scale code repositories, migration commits matching specific version pairs are rare [18, 32]. For instance, Xu et al. [45] extracted 3,674 migration commits from 465 projects, yet only 63 of them corresponded to the most frequent Android version upgrade. As a result, these approaches generalize poorly to libraries lacking rich documentation or sufficient historical migration examples.

**Compilation Error-Repairing Strategies.** This research line leverages compiler feedback to detect breaking changes and apply predefined operators. For example, LibCatch [47] systematically utilizes compilation errors to locate API mismatches and migrate code via a fixed set of repair operators. They struggle with complex, multi-step structural repairs that fall outside their templates, such as cascading errors introduced by altered inheritance constraints or complex method refactoring.

**Prompt-Only LLM-Based Migration Approaches.** Recent progress in LLMs has motivated their use for automated code migration [1, 3, 19, 21, 23]. These works primarily focus on designing prompts to guide LLMs in generating migration patches. To provide LLMs with external API knowledge, some works [46, 50] suggested using Retrieval-Augmented Generation (RAG) to access relevant API documentation. However, researchers [38, 40, 49] argue that ambiguous version-related queries complicate the retrieval process, as similar embeddings of version strings hinder the system's ability to distinguish specific version features accurately.

## 7 Conclusion

In this paper, we present DepUpgrade, a novel automated framework that models API migration as a *state-path exploration* problem to resolve complex, cascading build failures induced by dependency upgrades. By extracting a dual-index migration knowledge base directly from library source code, DepUpgrade grounds LLM generation to mitigate structural hallucinations. Furthermore, its causality-guided evaluation loop continuously assesses compiler feedback to successfully navigate cascading errors without manual intervention. Extensive evaluations demonstrate that DepUpgrade achieves a 96.7% full repair rate on the benchmark and strong compilation-repair effectiveness across 18 real-world projects. The semantic audit shows that most successful error-repair patches preserve the intended behavior. Future work will extend DepUpgrade to support a broader range of programming languages.

## Acknowledgments

This work was supported by the National Key R&D Program of China No 2024YFB4506200 and the Fundamental Research Funds for the Central Universities.

## Data Availability Statement

A permanent archive of the source code and experimental datasets is also available on Zenodo with DOI https://doi.org/10.5281/zenodo.20397314.

## References

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

[2] Jesper Andersen and Julia L Lawall. 2010. Generic patch inference. *Automated software engineering* 17, 2 (2010), 119–148. doi:10.1007/s10515-010-0062-z

[3] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, Balasubramanyan Ashok, and Shashank Shet. 2024. Codeplan: Repository-level coding using llms and planning. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 675–698. doi:10.1145/3643757

[4] Ittai Balaban, Frank Tip, and Robert Fuhrer. 2005. Refactoring support for class library migration. *ACM SIGPLAN Notices* 40, 10 (2005), 265–279. doi:10.1145/1103845.1094832

[5] Chunyang Chen, Zhenchang Xing, Yang Liu, and Kent Ong Long Xiong. 2021. Mining likely analogical apis across third-party libraries via large-scale unsupervised api semantics embedding. *IEEE Transactions on Software Engineering* 47, 3 (2021), 432–447. doi:10.1109/TSE.2019.2896123

[6] Kingsum Chow and David Notkin. 1996. Semi-automatic update of applications in response to library changes. In *1996 Proceedings of International Conference on Software Maintenance*. IEEE, 359–368. doi:10.1109/ICSM.1996.565039

[7] Barthélémy Dagenais and Martin P Robillard. 2011. Recommending adaptive changes for framework evolution. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 20, 4 (2011), Article 19, 19:1–19:35. doi:10.1145/2000799.2000805

[8] DeepSeek. 2026. DeepSeek API Pricing. https://api-docs.deepseek.com/quick_start/pricing. Accessed: 2026-03-26.

[9] Mattia Fazzini, Qi Xin, and Alessandro Orso. 2019. Automated API-usage update for Android apps. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*. 204–215. doi:10.1145/3293882.3330571

[10] Xiang Gao, Arjun Radhakrishna, Gustavo Soares, Ridwan Shariffdeen, Sumit Gulwani, and Abhik Roychoudhury. 2021. APIfix: output-oriented program synthesis for combating breaking changes in libraries. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–27. doi:10.1145/3485538

[11] Google DeepMind. 2026. Gemini API Pricing. https://ai.google.dev/pricing. Accessed: 2026-03-26.

[12] Stefanus A Haryono, Ferdian Thung, Hong Jin Kang, Lucas Serrano, Gilles Muller, Julia Lawall, David Lo, and Lingxiao Jiang. 2020. Automatic Android deprecated-API usage update by learning from single updated example. In *Proceedings of the 28th international conference on program comprehension*. 401–405. doi:10.1145/3387904.3389285

[13] Stefanus A Haryono, Ferdian Thung, David Lo, Julia Lawall, and Lingxiao Jiang. 2021. MLCatchUp: Automated update of deprecated machine-learning APIs in Python. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 584–588. doi:10.1109/ICSME52107.2021.00061

[14] Johannes Henkel and Amer Diwan. 2005. Catchup! capturing and replaying refactorings to support api evolution. In *Proceedings of the 27th international conference on Software engineering*. 274–283. doi:10.1145/1062455.1062512

[15] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. 2019. Inferring program transformations from singular examples via big code. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 255–266. doi:10.1109/ASE.2019.00033

[16] Sukrit Kalra, Ayush Goel, Dhriti Khanna, Mohan Dhawan, Subodh Sharma, and Rahul Purandare. 2016. POLLUX: safely upgrading dependent application libraries. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 290–300. doi:10.1145/2950290.2950345

[17] Ameya Ketkar, Oleg Smirnov, Nikolaos Tsantalis, Danny Dig, and Timofey Bryksin. 2022. Inferring and applying type changes. In *Proceedings of the 44th International Conference on Software Engineering*. 1206–1218. doi:10.1145/3510003.3510115

[18] Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. 2020. Understanding type changes in java. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 629–641. doi:10.1145/3368089.3409725

[19] Li Kuang, Qi Xie, HaiYang Yang, Yang Yang, Xiang Wei, HaoYue Kang, and YingJie Xia. 2025. APIMig: a project-level cross-multi-version API migration framework based on evolution knowledge graph. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*. 7455–7463. doi:10.24963/ijcai.2025/829

[20] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? An empirical study on the impact of security advisories on library migration. *Empirical Software Engineering* 23, 1 (2018), 384–417. doi:10.1007/s10664-017-9521-5

[21] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024). doi:10.48550/arXiv.2412.19437

[22] Mingwei Liu, Yanjun Yang, Yiling Lou, Xin Peng, Zhong Zhou, Xueying Du, and Tianyong Yang. 2023. Recommending analogical apis via knowledge graph embedding. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1496–1508. doi:10.1145/3611643.3616305

[23] Runlin Liu, Yuhang Lin, Yunge Hu, Zhe Zhang, and Xiang Gao. 2024. LLM-Based Java Concurrent Program to ArkTS Converter. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery, 2403–2406. doi:10.1145/3691620.3695362

[24] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 727–739. doi:10.1145/3106237.3106253

[25] Na Meng, Miryung Kim, and Kathryn S McKinley. 2011. Systematic editing: generating program transformations from an example. *ACM Sigplan Notices* 46, 6 (2011), 329–342. doi:10.1145/1993498.1993537

[26] Na Meng, Miryung Kim, and Kathryn S McKinley. 2013. LASE: locating and applying systematic edits by learning from examples. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 502–511. doi:10.1109/ICSE.2013.6606596

[27] Sichen Meng, Xiaoyin Wang, Lu Zhang, and Hong Mei. 2012. A history-based matching approach to identification of framework evolution. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 353–363. doi:10.1109/ICSE.2012.6227179

[28] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. Deepdelta: learning to repair compilation errors. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 925–936. doi:10.1145/3338906.3340455

[29] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2014. Statistical learning approach for mining API usage mappings for code migration. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 457–468. doi:10.1145/2642937.2643010

[30] Anh Tuan Nguyen and Tien N Nguyen. 2015. Graph-based statistical language model for code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 858–868. doi:10.1109/ICSE.2015.336

[31] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2013. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*. 651–654. doi:10.1145/2491411.2494584

[32] Hoan Anh Nguyen, Tien N Nguyen, Danny Dig, Son Nguyen, Hieu Tran, and Michael Hilton. 2019. Graph-based mining of in-the-wild, fine-grained, semantic code change patterns. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 819–830. doi:10.1109/ICSE.2019.00089

[33] OpenAI. 2026. OpenAI API Pricing. https://openai.com/api/pricing/. Accessed: 2026-03-26.

[34] Rick Ossendrijver, Stephan Schroevers, and Clemens Grelck. 2022. Towards automated library migrations with error prone and refaster. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*. 1598–1606. doi:10.1145/3477314.3507153

[35] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 404–415. doi:10.1109/ICSE.2017.44

[36] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. 2016. On the reaction to deprecation of 25,357 clients of 4+ 1 popular Java APIs. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 400–410. doi:10.1109/ICSME.2016.64

[37] The Eclipse Foundation. [n. d.]. *Eclipse Java Development Tools (JDT)*. http://www.eclipse.org/jdt/

[38] 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

[39] Louis Wasserman. 2013. Scalable, example-based refactorings with refaster. In *Proceedings of the 2013 acm workshop on workshop on refactoring tools*. 25–28. doi:10.1145/2541348.2541355

[40] Tongtong Wu, Weigang Wu, Xingyu Wang, Kang Xu, Suyu Ma, Bo Jiang, Ping Yang, Zhenchang Xing, Yuan-Fang Li, and Gholamreza Haffari. 2024. Versicode: Towards version-controllable code generation. *arXiv preprint arXiv:2406.07411* (2024). doi:10.48550/arXiv.2406.07411

[41] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. 2010. Aura: a hybrid approach to identify framework evolution. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 325–334. doi:10.1145/1806799.1806848

[42] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. 2017. Historical and impact analysis of API breaking changes: A large-scale study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 138–147. doi:10.1109/SANER.2017.7884616

[43] Zhenchang Xing and Eleni Stroulia. 2007. API-evolution support with Diff-CatchUp. *IEEE Transactions on Software Engineering* 33, 12 (2007), 818–836. doi:10.1109/TSE.2007.70747

[44] Zhenchang Xing and Eleni Stroulia. 2007. Differencing logical UML models. *Automated Software Engineering* 14, 2 (2007), 215–259. doi:10.1007/s10515-007-0007-3

[45] Shengzhe Xu, Ziqi Dong, and Na Meng. 2019. Meditor: inference and application of API migration edits. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 335–346. doi:10.1109/ICPC.2019.00052

[46] Daoguang Zan, Bei Chen, Zeqi Lin, Bei Guan, Wang Yongji, and Jian-Guang Lou. 2022. When language model meets private library. In *Findings of the Association for Computational Linguistics: EMNLP 2022*. 277–288. doi:10.18653/v1/2022.findings-emnlp.21

[47] Hao Zhong and Na Meng. 2024. Compiler-directed migrating api callsite of client code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12. doi:10.1145/3597503.3639084

[48] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. 2010. Mining API mapping for language migration. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 195–204. doi:10.1145/1806799.1806831

[49] Bingzhe Zhou, Xinying Wang, Shengbin Xu, Yuan Yao, Minxue Pan, Feng Xu, and Xiaoxing Ma. 2023. Hybrid api migration: A marriage of small api mapping models and large language models. In *Proceedings of the 14th Asia-Pacific Symposium on Internetware*. 12–21. doi:10.1145/3609437.3609466

[50] Shuyan Zhou, Uri Alon, Frank F Xu, Zhengbao Jiang, and Graham Neubig. 2022. Docprompting: Generating code by retrieving the docs. In *The Eleventh International Conference on Learning Representations*.

Received 2026-03-26; accepted 2026-06-18