

A Hybrid Automated API Migration Framework for an Emerging Programming Language: A Case Study on Cangjie

Sicheng Lu^{†‡}, Lingyu Zhang^{†‡}, Huiyan Wang^{*†§}, and Chang Xu^{†‡}

[†]State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

[‡]School of Computer Science, Nanjing University, Nanjing, China

[§]Software Institute, Nanjing University, Nanjing, China

{502024330035, zly}@smail.nju.edu.cn, {why, changxu}@nju.edu.cn

Abstract—The rapid evolution of emerging programming languages like Cangjie introduces frequent backward-incompatible API changes, hindering ecosystem stability. Traditional data-driven migration tools mine historical repositories, but suffer from a cold-start problem in new language ecosystems. While Large Language Models (LLMs) offer a promising alternative, directly applying them to project-level migration exposes hallucinations due to training data scarcity, non-determinism, and prohibitive inference cost. To tackle these problems, we propose HAMEL, a hybrid API migration framework for emerging languages. By constructing a structured knowledge base offline from official documentation and changelogs to address the cold-start problem, HAMEL deploys a two-stage online engine: deterministic rule-based migration followed by iterative compiler-feedback LLM repair. Our evaluation shows that HAMEL achieves up to 96% and 80% compilation success on official examples and real-world projects, respectively, with functional correctness demonstrated on oracle-bearing subjects, while outperforming prompting- and agentic-based LLM baselines at a fraction of the agentic baseline’s inference cost. Through an in-depth case study on Cangjie, we derive detailed insights into the design choices of this emerging language, and leave broader generalizations for future investigation.

Index Terms—API Migration, Emerging Programming Language, Program Repair

I. INTRODUCTION

While the continuous evolution of programming languages is essential, rapid iteration in emerging ecosystems such as Cangjie [1], [2] creates a conflict between language advancement and ecosystem stability. Prior to its 1.0.0 LTS release, Cangjie issued internal releases every one to three months, frequently introducing backward-incompatible API changes. This rapid evolution continues post-1.0.0, with Short-Term Support (STS) versions planned for release within 6-month cycles [3]. In our collection, 58% (122/210) of real-world version updates failed to compile on the newer version due to API obsolescence (§IV-B). Moreover, these breaking changes often alter the structural usage patterns of high-frequency APIs, requiring developers to manually adapt all affected call sites [4], [5]. Traditional data-driven API migration approaches rely on mining migration rules from historical repositories or

prior commits [6], [7], but suffer from a cold-start problem in emerging ecosystems like Cangjie, where historical migration data is largely absent [8].

Large Language Models (LLMs) offer a promising alternative [9], [10], inferring semantic transformations without task-specific migration data. Despite this promise, directly applying LLMs to code migration in emerging language ecosystems such as Cangjie exposes critical limitations. First, LLMs lack domain-specific grounding and established usage patterns in emerging ecosystems, where APIs are still evolving and documentation is sparse; they frequently produce hallucinations and overgeneralization [11], [12]. They may come up with nonexistent APIs and struggle with precise API resolution, potentially applying migration strategies from other languages to similarly named but distinct entities, leading to destructive modifications. Second, the inherent instability of emerging language ecosystems exacerbates LLMs’ lack of execution determinism. LLMs may generate inconsistent transformations for identical call sites or silently omit instances, violating the strict consistency required for reliable software evolution [13].

To mitigate these limitations, recent work employs autonomous LLM agents that combine LLM-driven code generation [9], [11], [14] with iterative compiler feedback [15], [16]. While effective for isolated bug fixing [13], this agentic paradigm struggles with the large-scale, repetitive edits of project-level API migration. Agents spend most iterations resolving syntactic errors before addressing the semantic ones [17], [18], and frequent LLM queries required across scattered call sites incur prohibitive inference costs [19], as illustrated by the agent traps in our motivating example (§II).

To address these limitations, we identified three key insights that guide our design. **First**, the cold-start problem and lack of domain-specific grounding can be mitigated by using official API documentation and changelogs, which are typically available in emerging ecosystems despite the absence of historical migration data [20], [21]. **Second**, to achieve consistent entity resolution and deterministic execution, the migration process should shift from purely LLM-driven textual generation to structural code transformation. In real-world projects, the same obsolete API is frequently invoked across

*Corresponding author.

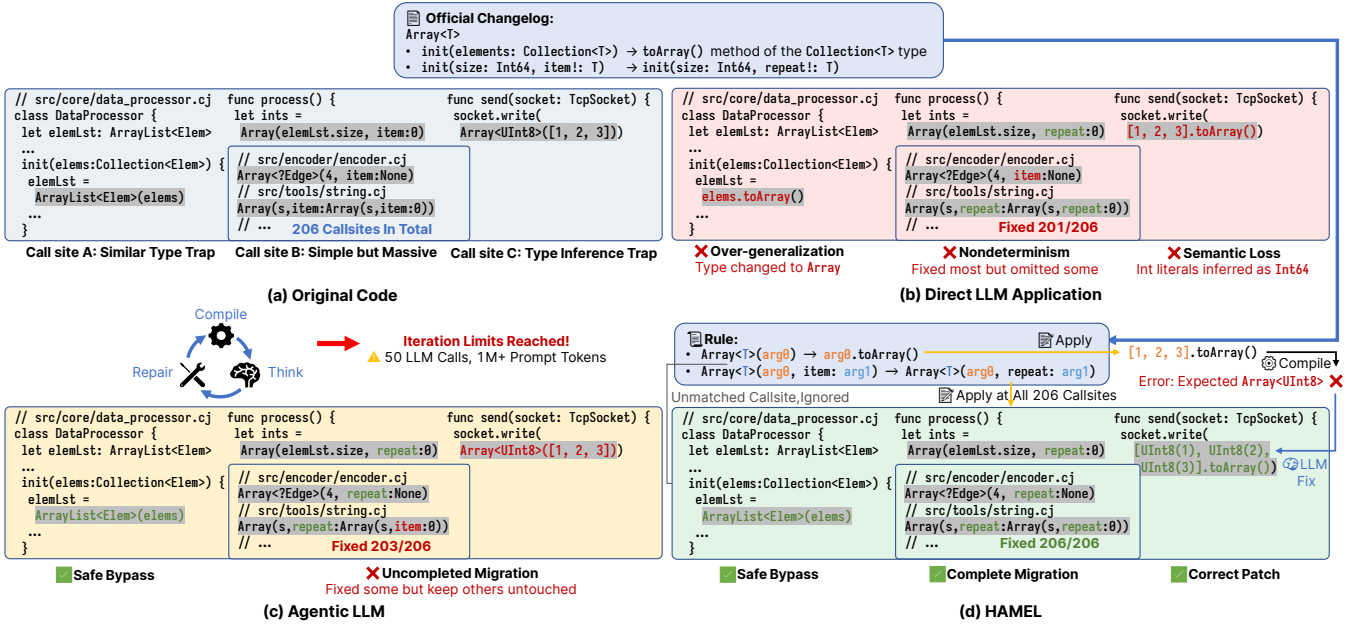


Fig. 1. A composite motivating example comparing Original Code, Direct LLM Application, Agentic LLM, and HAMEL.

multiple locations, and the required AST-level transformations are consistent [22]. **Third**, to address the cost inefficiency of pure LLM approaches, AST rules reduce inference costs by bypassing LLMs for repetitive changes. However, AST rules may struggle with complex edge cases involving deep context dependencies or logic restructuring. For such cases, LLMs are still needed for context-aware reasoning [23]. We thus constrain LLM reasoning with domain knowledge and iterative compiler feedback [24], [25], reducing hallucinations while preserving the LLM’s semantic reasoning.

Based on these insights, we propose HAMEL, a Hybrid API Migration framework for an Emerging Language. In the offline phase, we use LLMs to process official documentation and changelogs, constructing a structured knowledge base of migration rules that addresses the cold-start problem in emerging ecosystems. Then, given any project for migration, the online phase applies such rules deterministically across all affected call sites, with compiler-feedback LLM repair handling only the residual cases that fail to compile. This design confines LLM uncertainty to a bounded rule set, avoids per-call-site inference for repetitive edits, and yields a reusable, inspectable asset that can bootstrap migration even when community examples are scarce. We implement HAMEL on the Cangjie ecosystem and evaluate its performance on both official examples and real-world projects. Results show that HAMEL achieves up to 96% compilation success on examples and 80% on real-world projects, with functional correctness confirmed on oracle-bearing subjects, outperforming both prompting- and agentic-based LLM baselines in this representative emerging-language setting.

The core contributions of this paper are summarized as follows:

- **A rule-first migration engine** that performs most edits with

deterministic, type-resolved AST rules, invoking LLMs only for residual semantic cases.

- **A documentation-led, compiler-validated rule asset** built by offline synthesis from changelogs and API references (optionally enriched by mined community examples), and validated against the target version to ensure reliability in cold-start scenarios.
- **An implementation and evaluation on the Cangjie ecosystem**, reaching up to 96% compilation success on official examples and 80% on real-world projects, with functional correctness confirmed on oracle-bearing subjects, outperforming prompting- and agentic-based LLM baselines.

The rest of this paper is organized as follows. Section II presents an illustrative example. Section III elaborates on our HAMEL. Section IV evaluates HAMEL’s performance and Section V provides further discussions. Finally, Section VI reviews related work, and Section VII concludes the paper.

II. MOTIVATING EXAMPLE

To illustrate the challenges of API migration in emerging ecosystems like Cangjie, we present a composite motivating example in Figure 1. The changelog documents two API changes in Cangjie’s standard library from version 0.53.18 (v_s) to its next internal version (v_t): (1) the constructor `init(elements: Collection<T>)` of `Array<T>` is removed and replaced by the new method `toArray()`, and (2) the parameter `item` in the `init(size: Int64, item!: T)` constructor is renamed to `repeat`.

Figure 1(a) shows a project using the older version of the Cangjie standard library, which contains several *call sites* that correspond to API invocations potentially affected by this library evolution, each presenting potential migration

traps. Specifically, call site A involves an `ArrayList` usage that resembles `Array` but is semantically unrelated to the affected `Array` API, posing a risk of misidentification during migration. Call site B corresponds to the second API change (renaming parameter `item` to `repeat` in the `init(size, item)` constructor). Although the required migration is straightforward, it must be applied consistently across hundreds of occurrences, demanding high automation reliability. Call site C is directly related to the first API change (the removal of `init(elements)` in favor of `toArray()`), requiring challenging semantic preservation.

Directly applying LLMs to this task (Figure 1(b)) exposes typical limitations.

(1) Over-generalization and non-determinism. Without rigorous structural type constraints, the LLM may destructively over-generalize `toArray()` to a syntactically similar but distinct `ArrayList` initialization (call site A). And without execution determinism, it fixes only some of the 206 scattered `item` invocations across the project (call site B), silently omitting others.

(2) Semantic loss. Since LLMs usually lack deep type-inference awareness of emerging languages such as Cangjie, while generating a patch that superficially aligns with the changelog’s intent, the LLM can translate `Array<UInt8>([1, 2, 3])` into `[1, 2, 3].toArray()`, which meets the changelog syntactically but causes semantic loss. In the original code, the explicit `Array<UInt8>` instantiation forced the integer literals to be treated as `UInt8`. Once the generic wrapper is stripped, the final literals will be treated as `Array<Int64>` by default. Since the `socket.write` indeed expects an `Array<UInt8>`, this semantic loss results in an unexpected compilation error (mismatched types).

Although deploying LLMs in a more agentic manner, such as leveraging iterative compiler feedback, can successfully resolve some migration issues, such approaches still face limitations in terms of high automation cost and semantic loss, particularly encountering a severe scalability bottleneck in complex project migrations (Figure 1(c)). For example, call site B, which corresponds to 206 scattered invocations across a multi-module project, exemplifies this challenge. Even with iterative repair, the agent cannot observe all 206 errors simultaneously and may exhaust its iteration budget before completing the migration. Moreover, without sufficient support for the emerging language, the repair may prove ineffective or even lead to repeated attempts.

HAMEL combines deterministic rule-based migration with intelligent iterative repair. Rules resolve the `ArrayList` mismatch (Call site A) and the repeated `item` migration (Call site B), and then compiler diagnostics (e.g., Expected `Array<UInt8>`) and context-sensitive knowledge resolve Call site C.

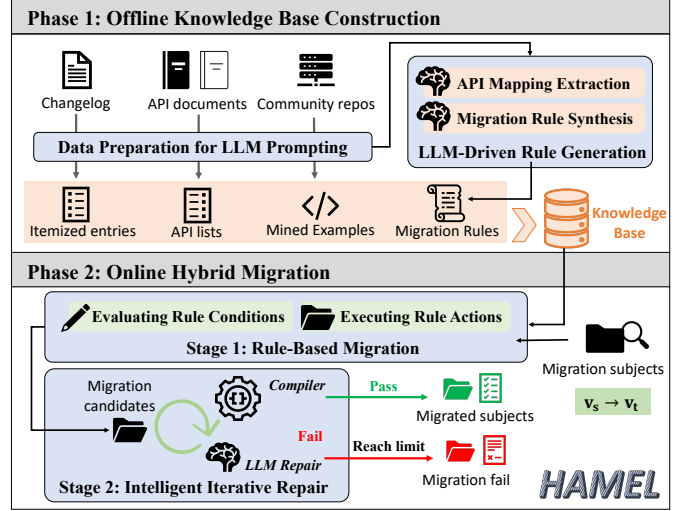


Fig. 2. The HAMEL overview.

III. METHODOLOGY

A. Framework Overview

Figure 2 overviews HAMEL, which migrates a project’s use of the standard library from a source version v_s to a target version v_t in two phases. The *offline* phase builds a knowledge base of executable AST rules from documentation and changelogs, bootstrapping migration in cold-start ecosystems. The *online* engine applies them deterministically and confines compiler-feedback LLM repair to residual edge cases.

B. Offline Knowledge Base Construction

1) *Data Preparation for LLM Prompting*: In order to support LLMs for migration rule generation in an emerging ecosystem where historical API migration data are scarce [6], we first build a structured index and extract data from three sources as follows.

a) *Changelog Itemization*: A parsing script segments monolithic Markdown changelogs into granular entries while preserving structural metadata (e.g., section headers, parent list items) per migration task.

b) *API Documentation Structuring*: We parse official API documentation for each version into structured lists of package names, declaring types, API names, signatures, and descriptions. To handle complexities such as function overloading, APIs are uniquely identified by a quadruple of (version_id, package, declaring_type, signature). Cross-referencing these lists across two versions yields a precise diff of altered APIs.

c) *Contextual Example Mining*: We mine code examples from sparse community repositories to supplement isolated API signatures with usage context. Combining tree-sitter [26] for AST parsing with Cangjie LSP [27] textDocument/hover responses, we resolve example call nodes to fully qualified API signatures. By tracking dependency upgrades in `cjpm.toml` across commits, we partition examples into two assets: *Usage Examples* (raw snippets of an API usage at its version) and *Migration Examples* (developer

diffs migrating an API across source and target versions). They are fetched on demand and feed both offline rule generation (§III-B2) and online iterative repair (§III-C2).

2) **LLM-Driven Rule Generation:** With the structured API lists and contextual examples in hand, the framework synthesizes executable migration rules. For each migration task ($v_s \rightarrow v_t$), a rule is formalized as a composition of fine-grained, deterministic AST manipulation operators, ensuring syntactic correctness while mitigating LLM execution non-determinism. We focus on adjacent-version migration (where the corresponding changelog is naturally available). Non-adjacent migration ($|v_t - v_s| > 1$) is simulated as continuous migration by iteratively applying adjacent-version migrations along $v_s \rightarrow \dots \rightarrow v_t$ and composing the per-step rules.

a) **API Mapping Extraction:** The first step is to align the itemized natural language changelog entries with the structured API lists. For each migration task for the adjacent version pair v_s and v_t , we use an LLM and a fuzzy API search tool to process the corresponding changelog entries. For each changelog entry, the LLM reads the natural-language description and uses the fuzzy search tool to align it with entries in the API lists of v_s and v_t , producing an old-to-new mapping. The resulting mappings fall into three categories: one-to-one (API replacement), one-to-many (API functionality splitting), and one-to-null (API deprecation). We focus on these mappings with a single-source API, as cases involving multiple source APIs are challenging to match precisely in the legacy project using a rule-based treatment and are thus deferred to the intelligent iterative repair stage. Additionally, new APIs introduced in the target version are excluded from these mappings, since they correspond to functionality absent from the legacy project and require no migration. We call the source-version APIs in these mappings the *affected* APIs of the migration task. The per-version signature lists are the indispensable anchor: comparing them across versions already shows which APIs changed, and the changelog text, per-API descriptions, and mined migration examples sharpen each match where present. As changelog clues and examples thin, the LLM relies more on the queryable API documentation. Unresolved mappings result in compile errors handled by the repair stage (§III-C2).

b) **Migration Rule Synthesis:** Drawing inspiration from established practices in automated API evolution [6], we associate each migration rule with a specific affected API and formalize the corresponding migration rule r as a Condition-Action pair, denoted as $r = \langle C, A \rangle$. The Condition (C) defines the strict prerequisite to trigger the migration, typically relying on precise semantic symbol resolution rather than rigid syntactic matching. The Action (A) dictates the deterministic AST manipulations applied once triggered. Under this abstraction, we consider two categories of migration rules as follows:

- 1) **EditEnv:** Migrating statements declaring the environment (i.e., `import`).
- **Condition (C):** The detection of either an import declaration or any call site of an affected API in migration.

- **Action (A):** The deletion of the corresponding import declaration (if present) and the injection of the designated new target dependency at each call site’s file.

- 2) **EditAPI:** Migrating functional call sites to perform code rewriting.

- **Condition (C):** The detection of an AST node representing a specific call site of an affected API.
- **Action (A):** The rewriting of the matched call site based on a predefined template pair (`expr_before`, `expr_after`). Both templates are Cangjie expressions with *placeholder variables*, i.e., abstract symbols (e.g., receiver `obj`, generic `T`, argument `arg0`) for the syntactic sub-components of a valid invocation. `expr_before` describes the affected API’s pre-migration usage. Then, at the trigger time the engine binds the matched call site’s sub-nodes to these placeholders and injects them into `expr_after` to construct the new AST topology.

Based on the extracted API mappings, the framework aims to synthesize these two categories of migration rules. Generally, according to the different synthesizing complexities behind each migration rule, HAMEL leverages two synthesis treatments as follows:

Strict Isomorphisms (Low Complexity, Bypassing LLM): When an API modification preserves the original AST topology, the synthesis complexity is minimal, and the framework mechanically generates the rules without invoking the LLM. This deterministic synthesis covers (1) *type-level renames or relocations* (of classes, structs, enums, or interfaces) and (2) *signature-preserving relocations*, where an API is moved to a new package but its enclosing type (or global scope) and complete signature remain identical. In these cases, an `EditEnv` rule updates dependencies. If the API is also renamed, `EditAPI` performs a parameter-free identifier substitution, otherwise it is omitted.

Structural Shifts (High Complexity, LLM-Driven): For modifications that exceed strict isomorphism (e.g., signature alteration or AST topology changes), the framework first deterministically synthesizes the `expr_before` template for Action (A) by analyzing the abstract syntax structure of the affected API’s signature (e.g., `Array<T>(arg0)` and `Array<T>(arg0, item: arg1)` as illustrated in Section II). Synthesizing the corresponding semantic `expr_after` template, which requires deeper reasoning, is delegated to the LLM. HAMEL constrains the LLM with a four-part prompt that grounds rewriting in target-version evidence: (1) signatures and descriptions of the affected and target APIs; (2) the corresponding changelog entry; (3) the deterministically generated `expr_before` template; and (4) up to five usage examples and five migration examples from community repositories (§III-B1). The LLM also retains the fuzzy search tool to discover undocumented dependencies and configure the corresponding `EditEnv` rules.

Eventually, HAMEL constructs a minimal compilable Cangjie snippet in the v_s environment, filling essential place-

holder symbols for compiler checks only. Candidate rules are applied to the snippet and compiled at v_t . Only rules that pass this check are persisted to the knowledge base, while failed candidates are discarded and regenerated (up to three retries). This ensures deterministic efficiency for subsequent online migration. In our Cangjie construction, most of these LLM-synthesized rewrite candidates compiled on the first attempt. Around one in seven still failed the check after three retries and were discarded rather than persisted, so that every rule in the knowledge base is compile-validated against the target version.

C. Online Hybrid Migration Engine

Once the knowledge base is built, HAMEL migrates a given target project from v_s to v_t with a two-stage online engine: first deterministic rule-based migration, then intelligent iterative repair.

1) *Deterministic Rule-Based Migration*: Stage 1 applies pattern-based AST rules across the codebase with deterministic consistency, eliminating partial or omitted migrations inherent in real-time LLM generation. The engine processes the project file by file, applying instantiated $\langle C, A \rangle$ rules through two coupled steps.

a) *Evaluating Rule Conditions*: To trigger the appropriate Action, the engine evaluates each rule’s Condition (C) by locating call sites of the obsolete API. To avoid matching same-named but unrelated symbols, it parses the target project into ASTs and queries the Cangjie LSP [27] for type information. For `EditEnv` rules, it additionally scans the file’s AST in the same pass for matching obsolete import declarations.

b) *Executing Rule Actions*: For `EditEnv` rules, the engine operates at the file level, cleanly deleting the obsolete import declarations (if present) and injecting the designated new target dependencies.

For `EditAPI` rules, the engine binds the template’s placeholders (abstract symbols such as `obj/T/arg0` in §III-B2) to the matched call site’s concrete sub-nodes, defaulting to original default or empty values when arguments or generic parameters are omitted, and replaces the subtree with `expr_after`. For strict isomorphisms that require only parameter-free identifier substitution, this reduces to direct in-place text replacement.

After processing all files by rule-based migration, the resulting codebase is fed to the Cangjie compiler at v_t . If compilation succeeds, the migration terminates. Otherwise, the framework proceeds to intelligent iterative repair.

2) *Intelligent Compiler-Feedback-Driven Iterative Repair*: Static AST rewrites cannot resolve context-dependent semantic issues such as the type inference at call site C (§II); pure LLM repair likewise lacks execution-level diagnostics [25], [28]. When Stage 1 outputs fail Cangjie compilation, HAMEL performs a localized, diagnostics-driven repair loop.

To ground repair in target-version evidence and reduce hallucination in pure LLM code generation [29], [30], HAMEL constructs a four-part prompt for each erroneous line: (1) the compiler diagnostic (e.g., `Expected Array<UInt8>`,

`got Array<Int64>`) to pinpoint the violation; (2) a code snippet consisting of the erroneous line and up to 10 lines above and 10 lines below; (3) the pre-migration source, capturing the developer’s original intent in case Stage 1’s rewrite is flawed; and (4) relevant changelog entries (if any) and up to five historical migration examples retrieved from the offline knowledge base, as evidence of target-version usage. Following tool-augmented practice [31], the LLM may also invoke the fuzzy search tool (§III-B2) for symbols not covered by the retrieved changelog entries (e.g., “method not found”).

HAMEL restricts the LLM to return suitable patches exclusively for the targeted erroneous line. After that, the framework invokes the corresponding compiler for the targeted version. If the patch resolves the targeted error without introducing new issues, the iterative loop proceeds to the next erroneous line until all are resolved. However, if the patch introduces another compilation error on the same line, the framework classifies it as a destructive modification and triggers a rollback. To prevent infinite loops, each erroneous line is allocated at most one repair attempt. The overall process terminates when either the project compiles successfully or all repair attempts are exhausted.

HAMEL mitigates the effects of imperfect inputs through two mechanisms. First, every LLM-synthesized migration rule is compile-checked against the target-version compiler before admission to the knowledge base, and only rules that pass are retained (§III-B2). Second, cases not resolved at the rule layer become compile errors and are forwarded to the repair stage, which grounds its patches in compiler feedback, source-code context, and target-version documentation. As changelog quality drops, more of the migration workload shifts to this repair stage. Behavioral equivalence beyond what the compiler can statically check falls outside these safeguards and is discussed separately as a limitation (§V-C).

IV. EXPERIMENTAL EVALUATION

In this section, we evaluate HAMEL’s performance and compare it with representative LLM-based baselines.

A. Research Questions

We aim to answer the following three questions:

- **RQ1 (Effectiveness)**: How effective is HAMEL in API migration for Cangjie, as compared to representative LLM-based baselines?
- **RQ2 (Efficiency and Cost)**: How does HAMEL compare to representative LLM-based baselines with respect to efficiency and cost?
- **RQ3 (Stage Contribution)**: How do HAMEL’s specific migration stages contribute to its performance?

B. Experimental Design and Setup

Migration scenarios. We evaluate HAMEL on the Cangjie ecosystem along the evolutionary timeline from an early milestone version 0.53.18 to the first official long-term support (LTS) version 1.0.0. Between the two milestones, there are four internal rapid-iteration releases without public version

TABLE I
STATISTICS OF MIGRATION SUBJECTS

Subjects	Metric	M_S	M_C
S_{ex}	Initial Examples	457	91
	Filtered Examples	67	57
	LoC: Mean (Median)	26 (22.5)	24 (21)
	w/ Expected Output	51	42
S_{pj}	Initial Projects	210	40
	Filtered Projects	122	39
	Files: Mean (Median)	48 (28)	31 (23)
	LoC: Mean (Median)	9,411 (2,761)	7,531 (2,623)
	w/ Test Cases	39	20

strings (referred to as v_1 to v_4), each accompanied by changelogs for the adjacent versions. We thus design two migration scenarios across the six consecutive versions: (1) *single-step migration* (M_S), involving five tasks, each an individual version update (e.g., M_{S_1} : $0.53.18 \rightarrow v_1$, M_{S_5} : $v_4 \rightarrow 1.0.0$); and (2) *continuous migration* (M_C), simulating cross-version migration from 0.53.18 to 1.0.0 by sequentially performing the five tasks, where any task’s failure causes the entire process to fail. M_C reflects practical software maintenance: in the Cangjie community, developers often skip multiple intermediate releases and upgrade directly to a new stable version.

Migration subjects. To cover both atomic API usage and complex real-world logic, we use two types of subjects: (1) *official examples* (S_{ex}), single-file snippets from the official Cangjie documentation that illustrate typical API usage patterns with expected outputs under given versions; and (2) *real-world projects* (S_{pj}), public third-party projects from the primary Cangjie open-source communities, Cangjie-TPC [32] and Cangjie-SIG [33], ranging from small utility scripts to large frameworks, with approximately one-third bundling complete test suites. For each migration task, we retain only subjects that compile in the source version but fail in the target version, and exclude them from HAMEL’s offline knowledge base (Section III-B) to prevent data leakage. Table I summarizes the two subject types per scenario. Under M_S , an initial pool of 457/210 official examples/real-world projects is filtered to 67/122. Under M_C , the corresponding initial pool of 91/40 is filtered to 57/39. M_S counts aggregate (subject, task) pairs across the five sub-tasks. The two types differ markedly in scale. S_{ex} averages 24–26 LoC, while S_{pj} comprises multi-file projects averaging 7,531–9,411 LoC, enabling evaluation of both per-call-site correctness and project-level scalability.

Comparisons. We compare HAMEL with two LLM-based baselines: (1) *Prompt-Engineering (PE) baseline* [8], [9], [34] prompts the LLM with retrieved knowledge (changelog entries, API docs, mined examples) and requests code patches. Inputs are then split at structural boundaries into 4,000-byte chunks (adapted from SWE-agent [15]). (2) *Agentic baseline* [25], [35], coupling an LLM with a closed-loop execution framework. To avoid a hand-tuned agent, we instantiate it with mini-swe-agent, a minimal open-source agent scaffold from the SWE-agent project [15], equipped with the same

changelog and API docs as HAMEL, plus shell and compiler-invocation tools. Each step issues one LLM query plus one tool action, with a 50-step budget, and success requires the submission to compile at v_t . This 50-step budget is generous relative to the typical length of successful agent trajectories on comparable tasks [15], giving a deterministic, cross-model-comparable stopping rule.

RQ1 process. We compare HAMEL with the two baselines across all subjects in both M_S and M_C , using *compilation success rate* (CSR) and *functional correctness rate* (FCR). CSR is the percentage of subjects that compile in the target version after migration. FCR captures end-to-end migration success: over all subjects with a validation oracle (expected outputs for S_{ex} , test suites for S_{pj}), a subject counts only if the migrated code both compiles and satisfies its oracle, so compilation is a necessary gate and non-compiling subjects count against FCR. To probe the effect of the deployed LLM, we evaluate three backends: Llama-3.1-8B (lightweight local) [36], DeepSeek-V3.2 (top-tier open-weight) [37], and Gemini-3.1-Pro (cutting-edge proprietary) [38].

RQ2 process. On S_{pj} under M_S with Gemini-3.1-Pro (RQ1’s strongest backbone), we measure migration efficiency and cost via average per-subject *total execution time*, *number of LLM calls*, and *token usage* (prompt + completion).

RQ3 process. Under the same Gemini-3.1-Pro backbone, we investigate HAMEL’s two online stages (rule-based migration and intelligent iterative repair). We partition subjects by outcome into *rule-resolved* (compile after stage 1), *LLM-resolved* (compile only after stage 2), and *unresolved* (fail both). We also track *modified lines of code* (ΔLoC) per stage and *compilation-error counts* at pipeline checkpoints (initial / post-stage-1 / post-stage-2).

Configuration and hardware. Experiments ran on a commodity PC (Intel Core i5-13500, 16 GB RAM, Ubuntu 22.04, Python 3.12). Llama-3.1-8B was served from a separate Ubuntu 24.04 server (Intel Xeon Silver 4510, 128 GB RAM, NVIDIA RTX 5090 32 GB). All LLMs were set to 4,096 max output tokens and temperature 0.1, balancing execution determinism and variance for iterative code repair.

C. Experimental Results

RQ1 (Effectiveness). We compared HAMEL with two baselines and evaluated their migration effectiveness, as shown in Table II.

We make three observations below. (1) HAMEL achieves the *strictly best* effectiveness in 21 of the 24 settings. All three exceptions are near-ceiling CSR on S_{ex} with stronger backbones (e.g., 89% vs. 95% under M_C /DeepSeek-V3.2). Yet HAMEL still leads on FCR in all three, and those extra compilations do not become more correct migrations. On S_{pj} the CSR margin is widest: 80% vs. at most 68% with the strongest backbone, with the highest FCR (67% vs. 59%). (2) HAMEL outperforms the baselines by substantial margins. Over the PE baseline, peak CSR/FCR gains reach 34/33 (Llama-3.1-8B), 46/45 (DeepSeek-V3.2), and 26/26 (Gemini-3.1-Pro) percentage points. Over the agentic baseline, they

TABLE II
EFFECTIVENESS COMPARISON FOR HAMEL AND BASELINES

Subjects	Scenario	Llama-3.1-8B			DeepSeek-V3.2			Gemini-3.1-Pro		
		PE baseline	Agentic baseline	HAMEL	PE baseline	Agentic baseline	HAMEL	PE baseline	Agentic baseline	HAMEL
Compilation Success Rate (CSR)										
S_{ex}	M_S	18/67 (27%)	2/67 (3%)	41/67 (61%)	48/67 (72%)	62/67 (93%)	61/67 (91%)	48/67 (72%)	60/67 (90%)	64/67 (96%)
	M_C	17/57 (30%)	2/57 (4%)	33/57 (58%)	38/57 (67%)	54/57 (95%)	51/57 (89%)	38/57 (67%)	55/57 (96%)	53/57 (93%)
S_{pj}	M_S	6/122 (5%)	1/122 (1%)	48/122 (39%)	39/122 (32%)	65/122 (53%)	88/122 (72%)	66/122 (54%)	83/122 (68%)	97/122 (80%)
	M_C	1/39 (3%)	0/39 (0%)	6/39 (15%)	5/39 (13%)	11/39 (28%)	23/39 (59%)	13/39 (33%)	21/39 (54%)	23/39 (59%)
Functional Correctness Rate (FCR)										
S_{ex}	M_S	11/51 (22%)	0/51 (0%)	28/51 (55%)	35/51 (69%)	43/51 (84%)	46/51 (90%)	38/51 (75%)	44/51 (86%)	48/51 (94%)
	M_C	11/42 (26%)	2/42 (5%)	22/42 (52%)	28/42 (67%)	36/42 (86%)	38/42 (90%)	30/42 (71%)	38/42 (90%)	39/42 (93%)
S_{pj}	M_S	1/39 (3%)	1/39 (3%)	13/39 (33%)	11/39 (28%)	21/39 (54%)	24/39 (62%)	16/39 (41%)	23/39 (59%)	26/39 (67%)
	M_C	0/20 (0%)	0/20 (0%)	2/20 (10%)	1/20 (5%)	5/20 (25%)	10/20 (50%)	4/20 (20%)	8/20 (40%)	9/20 (45%)

Values. Bold and highlighted cells denote the best approach per LLM (Llama, DeepSeek, Gemini).

Metrics. The denominator of FCR is the total number of subjects with available validation oracles (i.e., expected outputs or test suites).

reach 58/55, 31/25, and 12/8 (Table II). (3) HAMEL depends less on backbone strength. On Llama-3.1-8B it reaches 15–61% CSR (vs. the agentic baseline’s $\leq 4\%$), and on DeepSeek-V3.2 it already beats PE-on-Gemini-3.1-Pro across all settings (Table II).

We further decompose HAMEL’s S_{pj}/M_S result with Gemini-3.1-Pro. Of the 39 projects, 30 compile with a runnable oracle (77%), and 26 of those pass their oracle (87%), indicating that the residual gap lies primarily at the compilation gate, the very dimension where HAMEL excels (Table II). Notably, once code compiles, all approaches preserve behavioral semantics at comparable rates. A manual code review of 40 randomly sampled oracle-less S_{pj} projects, in which the authors visually compared each migrated call site to its pre-migration counterpart, found 36 (90%) to preserve the original behavior. Among the 16 oracle-less S_{ex} examples, the 14 that compiled under the same backbone were also manually reviewed, and all 14 preserved behavior. These manual rates are broadly consistent with the 87% oracle-based rate. We discuss the few observed divergences in §V-C.

Therefore, we answer RQ1 as follows: HAMEL achieves up to 96% CSR and 94% FCR on official examples, and 80% CSR and 67% FCR on real-world projects, matching or outperforming the PE and agentic baselines.

RQ2 (Efficiency and Cost). We focus on the online phase since the offline knowledge base is a one-time investment per version pair (10^5 – 10^6 prompt tokens, 10^4 completion tokens, 10–20 min), amortized to near-zero.

Table III profiles the online phase against baselines on S_{pj} with Gemini-3.1-Pro. HAMEL requires an average of only 4.6 and 9.0 calls, producing only 207 and 404 completion tokens per project for the M_S and M_C scenarios, respectively. In contrast, PE issues 2–3× more LLM calls (10.5–21.0) and 7–9× more completion tokens than HAMEL, with prompt-token volume in a comparable range. PE’s limited success is largely confined to simple migration cases, with its effectiveness dropping sharply as task complexity increases. Although more effective than the PE baseline using this Gemini backbone, the agentic approach proves highly inefficient and costly, as its conversational window expands dramatically. It consumed

TABLE III
EFFICIENCY AND COST COMPARISON AGAINST BASELINES ON S_{pj}
(GEMINI-3.1-PRO BACKEND)

Metric (Average per Project)	PE Baseline	Agentic Baseline	HAMEL
M_S Scenario			
Total Execution Time (s)	101.6	326.0	95.9
- Compilation Time (s)	5.5	35.0	21.6
- Migration Time (s)	96.1	290.9	74.3
Number of LLM Calls	10.5	30.5	4.6
Prompt Tokens	20,228	930,322	21,672
Completion Tokens	1,536	5,253	207
Cost (USD)	0.059	2.699	0.046
M_C Scenario			
Total Execution Time (s)	149.6	690.6	142.0
- Compilation Time (s)	19.3	63.6	37.7
- Migration Time (s)	130.3	627.0	104.3
Number of LLM Calls	21.0	61.3	9.0
Prompt Tokens	41,082	1,472,543	56,584
Completion Tokens	3,519	11,062	404
Cost (USD)	0.124	3.568	0.118

Cost is the token list price, billed per LLM call at the tier set by that call’s prompt size: \$2/\$12 per million input/output tokens at or below 200k, and \$4/\$18 above.

roughly 0.93M and 1.47M prompt tokens per project for M_S and M_C , respectively. This translates to nearly 59× and 30× the cost of HAMEL (\$2.7/\$3.6 vs. \$0.046/\$0.118, as shown in Table III), rendering it prohibitively expensive for large-scale deployment.

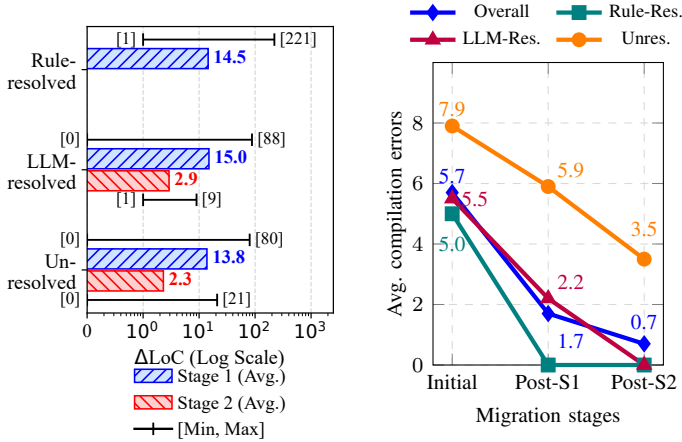
Beyond cost efficiency, our framework also achieves the lowest average total execution times of 95.9s and 142.0s per project for M_S and M_C . For the ~half that compile without repair, the average drops to 37.5s with no additional LLM calls or token consumption.

Therefore, we answer RQ2 as follows: HAMEL achieves relatively low execution time and cost while retaining its effectiveness, as compared to direct PE and Agentic baselines.

RQ3 (Stage Contribution). Table IV presents the classification of subjects according to their compilation results after each stage. (1) The rule-based migration independently migrates 38%–60% of subjects (over 50% in three out of four settings), serving as the primary driver for fundamental migration tasks. (2) The intelligent iterative repair stage contributes 21%–40% additional successfully migrated subjects, bringing CSR to 93%–96% on S_{ex} and 59%–80% on S_{pj} .

TABLE IV
COMPARISON OF STAGED COMPILATION RESULTS ACROSS MIGRATION
SCENARIOS AND SUBJECTS (GEMINI-3.1-PRO BACKEND)

Subjects	Scenario	Rule-resolved	LLM-resolved	Unresolved
S_{ex}	M_S	40/67 (60%)	24/67 (36%)	3/67 (4%)
	M_C	30/57 (53%)	23/57 (40%)	4/57 (7%)
S_{pj}	M_S	71/122 (58%)	26/122 (21%)	25/122 (20%)
	M_C	15/39 (38%)	8/39 (21%)	16/39 (41%)



(a) Modified LoC distribution (b) Error reduction trajectory
Fig. 3. Comparison of code modification and compilation error reduction for M_S scenario in S_{pj} (Gemini-3.1-Pro).

We further analyzed the code modifications (ΔLoC) and the reduction of compilation errors across the two stages. As shown in Figure 3, the rule-based migration performs the vast majority of code modifications, averaging 13.8–15.0 lines per project and reaching up to 221 lines for extensive subjects. This stage resolves most structural and syntax-level barriers, leading to a 70.18% reduction in compilation errors. In contrast, the intelligent iterative repair focuses on the remaining complex problems. While resolving them often requires fewer code modifications (e.g., an average of 2.3–2.9 ΔLoC , ranging from 0 to 21), these modifications typically target deeply hidden issues, thus reducing compilation errors by 17.54%.

Therefore, we answer RQ3 as follows: HAMEL’s rule-based stage migrates 38%–60% of subjects successfully, and the intelligent iterative repair adds a further 21%–40% of subjects, together bringing CSR to 93%–96% on S_{ex} and 59%–80% on S_{pj} .

V. DISCUSSION AND LIMITATIONS

A. Threats to Validity

Internal validity. Only around 1/3 of projects under M_S (and around 1/2 under M_C) carry test suites, so FCR is measured on this oracle-bearing subset. For the rest we rely on the RQ1 manual inspection (largely consistent with expectations) and defer fully automated oracle-based validation to future work. A second threat concerns baseline implementation. We instantiate the agentic baseline with mini-swe-agent [15], a

minimal scaffold aimed at general issue resolution rather than migration. We use it unmodified with the same changelog and API documentation as HAMEL. The PE baseline is implemented by us, as no off-the-shelf API-migration tool exists for Cangjie. Both adopt literature-suggested configurations (4,000-byte chunk for PE; 50-step agent budget). The consistent margins HAMEL maintains across three backbones and two scenarios indicate our conclusions do not hinge on these choices.

External validity. The first threat concerns the evolutionary timeline (0.53.18 to 1.0.0). We chose this window because it marks the run-up to Cangjie’s first LTS release, spanning four rapid-iteration intermediate versions and a rich and diverse set of API changes. The second threat lies in our evaluation subjects: we collected all publicly available projects from the primary Cangjie open-source communities and plan to extend to industrial-level subjects in future work. A further external-validity concern is that our study currently covers a single language. We discuss HAMEL’s generalizability beyond Cangjie in Section V-B.

Construct validity. Our headline metric, CSR, is a necessary but not sufficient proxy for migration correctness. The gap is quantified by FCR on the oracle-bearing subset and probed by manual code review of oracle-less subjects (RQ1), with residual silent-shift cases enumerated in §V-C.

B. Generalizability to Other Languages

Although our study evaluates HAMEL on Cangjie alone, we regard Cangjie as a representative emerging language. It shares with other emerging ecosystems two key characteristics: rapid, backward-incompatible evolution toward a first long-term support (LTS) release, and a cold-start scarcity of historical migration data. Moreover, our evaluation window spans a rich and diverse set of such breaking changes. Since HAMEL’s effectiveness stems from these shared characteristics, we expect its benefits to generalize to other emerging languages that exhibit similar traits. We leave broader cross-language validation to future work.

The inputs assumed by HAMEL, including an AST parser, a type resolver, compiler diagnostics, and per-version documentation with changelogs, are standard for any maintained language ecosystem. When a source is thin, HAMEL continues to operate on the remaining sources rather than failing outright. Per-version signatures anchor the mapping, other sources fill in where available, and unresolved cases result in compile errors handled by the repair stage (§III-C2). Producing such inputs where they are missing, e.g., generating changelog-style documentation [39]–[41] or API descriptions from code [42], [43], is another active research line, and orthogonal to HAMEL.

C. System Limitations

While our hybrid framework successfully resolves the vast majority of structural API changes, our qualitative analysis of unsuccessful migrations still exposes potential limitations of the current approach in the following aspects.

a) Paradigm-Level Architectural Shifts: Deterministic AST manipulations fail when APIs undergo holistic logic restructuring. For instance, early Cangjie versions defined the `Directory` class as an instantiable object, but later versions transformed it into a purely static utility class. Migrating this requires rewriting state management across multiple functions, which is beyond localized structural replacement.

b) Compiler-Bypassing Silent Semantic Shifts: Our correctness safeguards rely on closed-loop compiler validation, which can only statically verify type safety and syntax. This leaves the framework open to syntactically flawless patches that subtly alter runtime logic. For example, the source’s `CreateOrTruncate` option always truncates the file and can be opened with or without read access. The target’s `OpenMode`, however, ties the two together: its `Write` value truncates but forbids reading, whereas `ReadWrite` permits reading but does not truncate, thus no value offers both. For the read-enabled case, the synthesized mapping settled on `ReadWrite`, preserving the read/write intent but silently dropping the truncation the original guaranteed. Such a shift is invisible to the compiler and surfaces only when the migrated code runs. Guaranteeing strict behavioral equivalence against such silent shifts remains an unresolved challenge in automated code migration [28], [44].

c) Cross-Statement Dependencies and Incremental Repair: The current implementation of HAMEL uses a simple compiler-feedback loop to repair errors remaining after rule-based migration. Its local context window and line-restricted patches can hinder repairs requiring changes to upstream definitions. Consider an illustrative variant of call site C (§II), where a variable initialized with `Array<UInt8>([1, 2, 3])` is passed to `socket.write` dozens of lines later. Rewriting the initialization as `[1, 2, 3].toArray()` can change the inferred type to `Array<Int64>`, with the mismatch reported at the downstream call. The declaration may lie outside the local window, while the patch cannot directly modify it. The rollback policy can also reject intermediate patches that resolve one error but expose another on the same line. Our results characterize HAMEL with this implementation, without comparing alternative repair policies. Richer context retrieval and incremental patch refinement are promising extensions. Adapting and evaluating alternative repair strategies within HAMEL remains future work.

VI. RELATED WORK

Traditional Data-Driven API Migration. Traditional automated migration alleviates the manual burden of adapting backward-incompatible breaking changes [45]–[47] by mining transformation rules from historical repositories, prior commits, or code examples [6], [7], [48], [49]. However, these methodologies suffer from a “cold-start” problem in emerging ecosystems where historical migration data is scarce. Since breaking changes are typically documented in official release notes [20], [21], HAMEL’s offline phase uses LLMs to synthesize deterministic AST rules primarily from official documentation and changelogs.

Large Language Models in Code Migration. LLMs offer a promising alternative for zero-shot code translation and end-to-end API mapping [9], [34]. Despite this promise, directly applying LLMs to project-level migration exposes critical limitations. Recent evaluations report frequent factual hallucinations in low-resource settings [11], [12]. While techniques like Retrieval-Augmented Generation (RAG) can partially mitigate these issues [50], pure LLM approaches inherently lack the execution determinism required for strict structural consistency [13]. Furthermore, relying on continuous LLM inference for massive, repetitive updates incurs prohibitive computational costs [51]. HAMEL addresses these by delegating predictable migrations to deterministic AST rules.

Agentic Systems and Hybrid Architectures. To address the unguided nature of pure LLMs, recent advancements have shifted towards tool-augmented agentic systems and iterative compiler feedback [15], [25]. While effective for isolated bug fixing, fully autonomous agents face severe scalability bottlenecks during project-level migrations. Empirical studies show that agents handle information in the middle of long contexts less effectively [17] and often exhaust their step budget on scattered edits [18]. To bridge rigid rule-based migration and inefficient agentic generation, Li et al. [10] argue for hybrid architectures. Building upon this, HAMEL deploys a hybrid engine in which deterministic structural rules carry the bulk of the repetitive edits and the LLM’s iterative repair is confined to localized residual semantic errors. The closest prior effort using an LLM with compiler feedback for Cangjie is Wang et al. [24], who translate isolated Java functions into Cangjie. Their approach also combines rules, an LLM, and compiler feedback, but folds all three into training, i.e., hand-written rules and documentation-retrieval LLM repair clean the back-translated data that fine-tunes their model. HAMEL synthesizes API migration transformations offline from cross-version documentation and replays them deterministically across call sites at migration time, adding a compiler-pre-validated rule asset that resolves most edits before any LLM repair.

VII. CONCLUSION

In this paper, we addressed automated API migration for emerging languages such as Cangjie, where the cold-start problem renders data-driven approaches ineffective and standalone LLMs unreliable and costly. We proposed HAMEL, a hybrid framework that pairs an offline knowledge base synthesized from official documentation and changelogs, with an online two-stage engine combining deterministic rule-based migration and compiler-feedback LLM repair. Extensive evaluations on the Cangjie ecosystem show that HAMEL achieves up to 96% compilation success on official examples and 80% on real-world projects, with functional correctness demonstrated on oracle-bearing subjects, outperforming prompting- and agentic-based LLM baselines at a fraction of the agentic baseline’s inference cost.

Although our evaluation centers on Cangjie, the value of this case study lies in a methodology designed to transfer

beyond it. The same core of synthesizing executable rules offline from version documentation and changelogs, then applying them through deterministic rule-based migration with compiler-feedback LLM repair, directly targets the cold-start dilemma faced by any emerging ecosystem that lacks historical migration data. Provided a target language exposes parseable documentation and compiler diagnostics, the same paradigm is designed to apply, sidestepping the data-driven deadlock of traditional tools. We therefore see HAMEL as a blueprint for migration and, in time, broader software-evolution tasks in emerging language ecosystems; we leave cross-language validation to future work.

ACKNOWLEDGMENT

This research was supported by the Natural Science Foundation of China (Grant Nos. 92582201 and 62302209), the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025XDXM118). The authors gratefully acknowledge the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China. We thank the Cangjie team at Huawei for providing language resources that supported this study. We also thank the anonymous reviewers for their constructive comments and suggestions.

REFERENCES

- [1] Huawei, “Cangjie programming language white paper,” 2025. [Online]. Available: <https://developer.huawei.com/consumer/cn/doc/cangjie-guides-V5/cj-wp-abstract-V5>
- [2] X.-Y. Feng, R.-X. Fu, L. Shi, L. Tu, and Y.-Y. Yang, “An overview of Cangjie programming language,” *Journal of Computer Science and Technology*, vol. 41, pp. 217–244, 2026.
- [3] Huawei, “Cangjie community version lifecycle management specification,” 2025. [Online]. Available: <https://gitcode.com/org/Cangjie/discussions/129>
- [4] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, “Do developers update their library dependencies?” *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, Feb. 2018.
- [5] A. A. Sawant, R. Robbes, and A. Bacchelli, “On the reaction to deprecation of 25, 357 clients of 4+1 popular Java APIs,” in *Proceedings of the 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Raleigh, NC, USA: IEEE Computer Society, 2016, pp. 400–410.
- [6] S. Xu, Z. Dong, and N. Meng, “Meditor: Inference and application of API migration edits,” in *Proceedings of the 27th International Conference on Program Comprehension (ICPC)*. Montreal, QC, Canada: IEEE / ACM, 2019, pp. 335–346.
- [7] M. Lamothe, W. Shang, and T. P. Chen, “A3: Assisting android API migrations using code examples,” *IEEE Transactions on Software Engineering*, vol. 48, no. 2, pp. 417–431, 2022.
- [8] L. Kuang, Q. Xie, H. Yang, Y. Yang, X. Wei, H. Kang, and Y. Xia, “APIMig: A project-level cross-multi-version API migration framework based on evolution knowledge graph,” in *Proceedings of the 34th International Joint Conference on Artificial Intelligence (IJCAI)*. Montreal, QC, Canada: International Joint Conferences on Artificial Intelligence Organization, 2025, pp. 7455–7463.
- [9] A. Almeida, L. Xavier, and M. T. Valente, “Automatic library migration using large language models: First results,” in *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. Barcelona, Spain: ACM, 2024, pp. 427–433.
- [10] F. Li, J. Jiang, J. Sun, and H. Zhang, “Hybrid automated program repair by combining large language models and program analysis,” *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 7, pp. 202:1–202:28, 2025.
- [11] K. Cheng, X. Shen, Y. Yang, T. Wang, Y. Cao, M. A. Ali, H. Wang, L. Hu, and D. Wang, “CODEMENV: Benchmarking large language models on code migration,” in *Findings of the Association for Computational Linguistics: ACL 2025*. Vienna, Austria: Association for Computational Linguistics, 2025, pp. 2719–2744.
- [12] Q. Han, Y. Guang, H. Shu, and F. Kang, “A systematic evaluation of large language models in API mapping,” *IEICE Transactions on Information and Systems*, vol. E109.D, no. 7, pp. 1085–1099, 2026.
- [13] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan, “SWE-bench: Can language models resolve real-world github issues?” in *Proceedings of the 12th International Conference on Learning Representations (ICLR)*. Vienna, Austria: OpenReview.net, 2024.
- [14] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, “Large language models for software engineering: A systematic literature review,” *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 220:1–220:79, 2024.
- [15] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, “SWE-agent: Agent-computer interfaces enable automated software engineering,” in *Advances in Neural Information Processing Systems 37 (NeurIPS)*. Vancouver, BC, Canada: Curran Associates, Inc., 2024, pp. 50528–50652.
- [16] J. Yang, A. Prabhakar, K. Narasimhan, and S. Yao, “InterCode: Standardizing and benchmarking interactive coding with execution feedback,” in *Advances in Neural Information Processing Systems 36 (NeurIPS)*. New Orleans, LA, USA: Curran Associates, Inc., 2023, pp. 23826–23854.
- [17] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, “Lost in the middle: How language models use long contexts,” *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 157–173, 2024.
- [18] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, “Demystifying LLM-based software engineering agents,” *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, pp. 801–824, 2025.
- [19] Q. Zhao, L. Zhang, F. Liu, X. Lian, Q. Meng, Z. Jiao, Z. Zhou, J. Li, and L. Shi, “FastCoder: Accelerating repository-level code generation via efficient retrieval and verification,” in *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Seoul, Korea: IEEE, 2025, pp. 2299–2311.
- [20] R. Koçi, X. Franch, P. Jovanovic, and A. Abelló, “Classification of changes in API evolution,” in *Proceedings of the 23rd IEEE International Enterprise Distributed Object Computing Conference (EDOC)*. Paris, France: IEEE, 2019, pp. 243–249.
- [21] S. L. Abebe, N. Ali, and A. E. Hassan, “An empirical study of software release notes,” *Empirical Software Engineering*, vol. 21, no. 3, pp. 1107–1142, 2016.
- [22] M. Fazzini, Q. Xin, and A. Orso, “Automated API-usage update for Android apps,” in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. Beijing, China: ACM, 2019, pp. 204–215.
- [23] C. S. Xia, Y. Wei, and L. Zhang, “Automated program repair in the era of large pre-trained language models,” in *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*. Melbourne, Australia: IEEE, 2023, pp. 1482–1494.
- [24] J. Wang, C. Su, Y. Ou, Y. Li, J. Tan, L. Chen, and Y. Zhou, “Translating to a low-resource language with compiler feedback: A case study on Cangjie,” *IEEE Transactions on Software Engineering*, vol. 51, no. 9, pp. 2671–2692, 2025.
- [25] C. S. Xia and L. Zhang, “Automated program repair via conversation: Fixing 162 out of 337 bugs for \$0.42 each using chatgpt,” in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. Vienna, Austria: ACM, 2024, pp. 819–831.
- [26] M. Brunsfeld, A. Qureshi, A. Hlynskyi, ObserverOfTime, W. Lillis, J. Vera, dundargoc, P. Turnbull, T. Clem, D. Creager *et al.*, “tree-sitter/tree-sitter: v0.25.10,” Sep. 2025. [Online]. Available: <https://doi.org/10.5281/zenodo.17180150>
- [27] Microsoft, “Language server protocol specification - 3.17,” 2022. [Online]. Available: <https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/>
- [28] H. Fu, S. Eldh, K. Wiklund, A. Ermedahl, P. Haller, and C. Artho, “Auto-repair without test cases: How LLMs fix compilation errors in large industrial embedded code,” in *Proceedings of the 28th Euromicro Conference on Digital System Design (DSD)*. Salerno, Italy: IEEE, 2025, pp. 97–105.

- [29] Z. Zhang, C. Wang, Y. Wang, E. Shi, Y. Ma, W. Zhong, J. Chen, M. Mao, and Z. Zheng, "LLM hallucinations in practical code generation: Phenomena, mechanism, and mitigation," *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, pp. 481–503, 2025.
- [30] M. Islam, A. K. Jha, M. Mahmoud, I. Akhmetov, and S. Nadi, "An empirical study of Python library migration using large language models," in *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Seoul, Korea: IEEE, 2025, pp. 867–879.
- [31] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, and Y. Cao, "ReAct: Synergizing reasoning and acting in language models," in *Proceedings of the 11th International Conference on Learning Representations (ICLR)*. Kigali, Rwanda: OpenReview.net, 2023.
- [32] Huawei, "Cangjie-TPC," 2024. [Online]. Available: <https://gitcode.com/Cangjie-TPC>
- [33] —, "Cangjie-SIG," 2024. [Online]. Available: <https://gitcode.com/Cangjie-SIG>
- [34] R. Bairi, A. Sonwane, A. Kanade, V. D. C., A. Iyer, S. Parthasarathy, S. K. Rajamani, B. Ashok, and S. Shet, "CodePlan: Repository-level coding using LLMs and planning," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 675–698, 2024.
- [35] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language agents with verbal reinforcement learning," in *Advances in Neural Information Processing Systems 36 (NeurIPS)*. New Orleans, LA, USA: Curran Associates, Inc., 2023, pp. 8634–8652.
- [36] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan *et al.*, "The Llama 3 herd of models," 2024. [Online]. Available: <https://arxiv.org/abs/2407.21783>
- [37] DeepSeek-AI, "DeepSeek-V3.2: pushing the frontier of open large language models," 2025. [Online]. Available: <https://arxiv.org/abs/2512.02556>
- [38] Google DeepMind, "Gemini 3.1 pro - model card," 2026. [Online]. Available: <https://deepmind.google/models/model-cards/gemini-3-1-pro/>
- [39] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, A. Marcus, and G. Canfora, "ARENA: An approach for the automated generation of release notes," *IEEE Transactions on Software Engineering*, vol. 43, no. 2, pp. 106–127, 2017.
- [40] H. Jiang, J. Zhu, L. Yang, G. Liang, and C. Zuo, "DeepRelease: Language-agnostic release notes generation from pull requests of open-source software," in *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*, 2021, pp. 101–110.
- [41] F. Daneshyan, R. He, J. Wu, and M. Zhou, "SmartNote: An LLM-powered, personalised release note generator that just works," *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, pp. 1663–1686, 2025.
- [42] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2016, pp. 2073–2083.
- [43] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," in *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 795–806.
- [44] J. Yang, Y. Wang, Y. Lou, M. Wen, and L. Zhang, "A large-scale empirical review of patch correctness checking approaches," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. San Francisco, CA, USA: ACM, 2023, pp. 1203–1215.
- [45] D. Venturini, F. R. Cogo, I. Polato, M. A. Gerosa, and I. S. Wiese, "I depended on you and you broke me: An empirical study of manifesting breaking changes in client packages," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, pp. 94:1–94:26, 2023.
- [46] D. Kong, J. Liu, L. Bao, and D. Lo, "Toward better comprehension of breaking changes in the NPM ecosystem," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 4, pp. 111:1–111:23, 2025.
- [47] W. Li, F. Wu, C. Fu, and F. Zhou, "A large-scale empirical study on semantic versioning in Golang ecosystem," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Luxembourg: IEEE, 2023, pp. 1604–1614.
- [48] H. A. Nguyen, T. T. Nguyen, G. W. Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to API usage adaptation," in *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Reno/Tahoe, Nevada, USA: ACM, 2010, pp. 302–321.
- [49] W. Wu, Y. Guéhéneuc, G. Antoniol, and M. Kim, "AURA: A hybrid approach to identify framework evolution," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE)*. Cape Town, South Africa: ACM, 2010, pp. 325–334.
- [50] K. Shuster, S. Poff, M. Chen, D. Kiela, and J. Weston, "Retrieval augmentation reduces hallucination in conversation," in *Findings of the Association for Computational Linguistics: EMNLP 2021*. Punta Cana, Dominican Republic: Association for Computational Linguistics, 2021, pp. 3784–3803.
- [51] S. Ilager, L. F. Briem, and I. Brandic, "Green-Code: Learning to optimize energy efficiency in LLM-based code generation," in *Proceedings of the 2025 IEEE 25th International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. Tromsø, Norway: IEEE Computer Society, May 2025, pp. 559–569.