IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026

DOI: 10.17148/IJARCCE.2026.1511132

MigrateAI: An Intelligent Framework Migration and Compatibility Assistant

Rushikesh Pawar[1], Tanvi Yeola[2], Pranjali Yadav[3], Snehal Kale[4], B. A. Sonkamble[5]

Department of Computer Engineering, SCTR·s Pune Institute of Computer Technology, Pune, India[1-5]

Abstract: Moving an application to a newer framework release is rarely a matter of changing a few method names; this issue lies in discerning which is changing matter to that application, choosing replacements supported by the destination version, and deciding whether the changed software still behaves as anticipated; this review is studying four matching strands of research, such as API change detection, environment-sensitive migration evaluation, application-level library upgrades, and AIassisted repo refactoring; these studies are employed to formulate a six-part account of migration, from collecting release evidence through the assessing the trustworthiness of a completed patch, in actual application. A central value is that detecting a break, making an edit, and showing a safe migration are separate achievements, to a significant degree; and thus, we tackle where structural examination, documentation retrieval, language models, and testing can (to some degree) contribute. What's more, their proof remains insufficient; the resulting assurance model is, in fact, a proposal for future inquiry rather than a tested setup. As the selection of studies was stressed rather than complete, this article is introduced as a critical review, not a systematic survey. Index Terms: software evolution; API compatibility; library migration; repo examination; large language models; validation; traceability.

I. INTRODUCTION

A framework update can (in actual operation) look harmless in a dependency file and still create major work elsewhere in an application, in real-world scenarios. A removed method is relatively easy to know. Changes to defaults, error handling, return values, or interactions between packages are, of course, less visible, on the whole. Let alone, the applied question for maintainers is therefore not simply whether an API changed, in many respects; as well, a suggested repair would be explained by what indication, so that said, whether the change reaches their code, broadly speaking. The preparatory AI Framework Migration and Compatibility Assistant project [B1] is offering the motivating setting, on the whole. It's imagining documentation retrieval, structural comparison of APIs, syntaxaware repository inspection, dependency analysis, and creating patches followed by validation, broadly speaking. We don't introduce those planned components as an operational framework. More like, we ask how established research could (to a meaningful extent) support them and where major uncertainties remain, in actual application. APIDiff [1], CODEMENV [2], PyMigTool [3], and RepoAI [4] are particularly useful because they address different units of work, from API declarations to repository edits. The paper interpretively contributes by connecting those separate tasks through a six-stage taxonomy and explains why proof collected at one stage can't automatically establish success at another, in real-world scenarios; indeed, what we're doing is to carefully base our checking migration assistants, not to announce a new performance result, in real-world scenarios.

II. TECHNICAL FOUNDATIONS

Compatibility isn't one attribute. An application may still gather after an upgrade yet return distinct findings. But a changed interface may be repaired mechanically without changing intended behaviour. Source compatibility concerns whether client code remains valid; binary compatibility matters where compiled components interact; behavioral compatibility concerns observable outcomes; these dimensions shouldn't be reported as interchangeable. Beyond that, an upstream change has no impact on a given application until the pertinent reliance or use is established. A. From changed APIs to impacted files: Imports and calls that a text search might (in practice) confuse with comments or unrelated names can be shared by syntax trees in actual application; their usefulness is still bounded by the language and application; in turn, dependency graphs are helping widen the search beyond a single file, in actual application. A graph walk may need $O(V + E)$ time; building a consistent graph and deciding which reachable components truly need alteration are separate costs, and though graph reachability is only a candidate impact signal, in turn, dynamic imports, reflection, created code, and aliasing can, to some extent, hide relationships. B. Employing documentation without overtrusting it. Retrieval-augmented generation [5] can bring release notes and migration guides into an editing workflow. Sure enough, the firm detail is usually whether a retrieved passage applies to the exact source and target versions, on the whole. A highly similar example from the wrong release can (in practical terms) be actively misleading. Sentence embeddings [6] may help find relevant passages, while supervised models such as XGBoost [7] could potentially rank issues if suitable labels exist. Neither retrieval nor ranking, though, suggests that a patch preserves application behavior.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026

DOI: 10.17148/IJARCCE.2026.1511132

III. REVIEW APPROACH AND QUESTIONS

Representative studies are chosen to study separate stages of migration and to expose differences in their indication; this is an emphasized critical synthesis; it doesn't claim a complete search of IEEE, ACM, ACL, or other bibliographic databases, and no study-selection counts or formal quality scores are reported. The project description [B1] informs the issue setting but isn't autonomously tested support. Six questions guide the assessment: RQ1 asks what kinds of API incompatibility can be detected; RQ2 concerns how changed client code is found; RQ3 analyzes rules and retrieval-model editing; RQ4 asks what validation can actually establish; RQ5 studies why benchmark scores are difficult to compare; and RQ6 spots what would decide reproducibility and reviewability, broadly speaking. A future systematic extension should, in effect, publish its search dates and strings, eligibility criteria, screening decisions, pulled outcomes, and studyquality assessment, to a significant degree.

IV. WHAT THE SELECTED STUDIES ESTABLISH

A. APIDiff and the boundary of structural proof APIDiff breaking changes between Java API versions are, indeed, being investigated by [1]; its part in this discussion is to show the value of assessing a library before finding its clients, on the whole. A list of changed declarations is an advantageous indication; so, that said, it doesn't say which applications rest on them, on the whole. Nor can declaration comparison alone represent every semantic change; a migration assistant must therefore connect upstream differences to actual downstream use rather than treating every API change as a client defect.

B. CODEMENV and the importance of the target environment Under changing package environments, migration is being evaluated by CODEMENV [2]; in fact, the significant lesson is qualitative as well as numerical, such as code that looks reasonable may, in practice, be invalid, under the particular dependency versions needed by a task, to a significant degree. The mentioned study is outlining 922 instances covering 19 Python and Java packages and reports an average pass@1 of 26.50% across the seven language models; those figures belong to that benchmark and shouldn't be described as a whole-repository upgrade rate, on the whole.

C. PyMigTool and application-level outcomes The unit of assessment is more closely aligned with PyMigTool [3] to complete Python applications, broadly speaking. 717 applications are being outlined by the mentioned preprint and reports 32% completely correct migrations, under its evaluation definition; that outcome can't be checked directly with an instance-level pass@1 score, in real-world scenarios; indeed, the strength of the proof rests on how applications were chosen, what counted as correct. What's more, how carefully the tests scrutinized behaviour; in turn, its preprint title and version metadata also need inspecting before formal submission.

D. RepoAI and the limits of novelty claims Retrievalsupported, multi-agent cleaning up at repo scale is being examined by RepoAI [4], and it is therefore inaccurate to suggest that combining agents and retrieval for repo edits is inherently new, on the whole; what's more, framework upgrades introduce a narrower need, namely every replacement must fit a named release, broadly speaking. What's more, edits must, in practice, remain homogeneous across the dependent files and setups; in turn, a check of broad rewriting can't simply be relabeled as proof of trustworthy framework migration.

E. Architecture components versus demonstrated results RAG [5], sentence embeddings [6], and XGBoost [7] can motivate parts of a suggested architecture. But still, none is itself a check of migration success; in turn, in the same way, Qdrant, LangGraph, and severity estimation in [B1] are design choices. They should be judged by future retrieval, localization, prioritization, and validation experiments rather than outlined as verified enhancements.

V. A SIX-STAGE VIEW OF MIGRATION

Six activities are respected, because each is asking a diverse question; for starters, change discovery is asking what changed between releases, in many respects. As well, whether the proof is version-specific; second up, impact examination is asking where the application employs altered behaviour, including indirect reliance, and then, planning is asking which replacement is backed up by the destination framework, to a significant degree. Fourth, transformation is employing coordinated edits and data their origin; in fact, assurance reporting is stating what those checks support, what remains unresolved, when the dust settles up, in real-world scenarios. Fifth, validation is running available static and dynamic checks in the intended environment; what's more, whether a reviewer accepts the finding. The stages shouldn't be mistaken for a rigid implementation recipe; and thus, some tools are combining them or repeating them. Their value is analytical, like a deviation can be traced to a missing change, an overlooked use, a poor replacement, an inconsistent edit, a weak test, or an exaggerated conclusion; this separation also is preventing a patchgeneration success rate from being offered as proof of behavioral safety.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026

DOI: 10.17148/IJARCCE.2026.1511132

VI. READING RESULTS ACROSS STUDIES

A single evaluation target isn't shared by the four representative works; in turn, APIDiff concentrates on API-level differences; CODEMENV measures environment-conditioned migration instances; PyMigTool studies application upgrades; to a significant degree, and more, repoAI finds repo cleaning up, so in response, the denominator, success criteria, available test oracles, and practical roadblocks differ, in actual application. Ranking these works by one reported percentage would hint at a comparability the proof doesn't offer; in fact, missing measurements should be marked as unreported, not treated as failures or silently estimated. A more informative comparison would record, for each study, the release pairs assessed, the scope of code inspected, change-detection quality, localization accuracy, patch applicability, test results, and developer intervention.

VII. CRITICAL INTERPRETATION

A trustworthy migration workflow needs to answer two related but diverse questions. Code analysis can show why a location might be impacted; documentation can explain what a framework author recommends instead, and honestly neither is enough alone. An analysis result can be too broad, and documentation can be outdated or vague. Their combination is promising only when the system keeps source information, solves version conflicts, and is prepared to say no to an unsupported edit. A successful build checks another set of bounds. Specific syntax errors are ruled out by successful reading. The second issue is the meaning of validation. Passing tests offer indication only for performed cases and their assertions. These checks are useful. Reports should say which checks ran, under which connections, what coverage was available, and what remained untested. And yet, none proves general behavioral equivalence. The labels 'patch suggested', 'checks passed'. And 'reviewer permitted' communicate materially distinct levels of confidence.

VIII. UNRESOLVED RESEARCH PROBLEMS

A. Indication tied to releases Recommendations need to point to identifiable documentation and clearly state the original and destination releases. Retrieval should be checked for version correctness as well as topical relevance, or else an unsupported assumption can be hidden by a confident explanation.

B. Repositorywide effects A single repaired call may, more accurately, probably leave a second caller, a configuration setting, or a transitive reliance broken, on the whole. Useful benchmarks must comprise realistic multi-file cases and measure both overlooked impact and false notifications, on the whole.

C. Confidence in runtime behavior Differential tests, interface contracts, and property-based checks may uncover regressions that ordinary compilation misses; in turn, their value rests on the available oracle, though, in actual application. But an account of what the validation can't gather must be covered by assurance, in many respects.

D. Human effort and operating cost Comparisons with manual or rule-based migration should report time, compute use, interventions, and recovery effort rather than technical success alone, broadly speaking, and real costs are incurred by model inference, repeated retrieval, test execution, and failed tries, broadly speaking.

IX. PROPOSED ASSURANCE RECORD

Attaching a traceable record is put forward to every suggested migration; it would spot the applicable release change, affected employees, the reasoning behind a replacement, the exact patch, the validation environment and findings, and the eventual reviewer decision, in actual application. Linking these entries would let maintainers inspect what changed. As well, why the framework believed the edit was fitting; this is a conceptual proposal; it hasn't been built or found in the present study, in many respects; indeed, we claim which condition would win. Localization precision and recall would be measured by the check, accepted patches, gathered regressions, elapsed time, expense, and human corrections, in actual application; the same repo and release pairs should be employed by all conditions. A apt future experiment could compare manual work, deterministic rewrite rules, a languagemodel editor without documentation grounding, and an editor using versioned documentation and repository examination.

X. THREATS TO VALIDITY

This review is shackled by its selective source set and the absence of complete screening; it may miss ecosystemspecific methods or contrary findings. On the whole, preprints and archival publications don't carry the same review status, and reported quantitative outcomes need inspecting against the final primary texts, in real-world scenarios. Differences in benchmarks and validation criteria further limit generalization; indeed, the taxonomy is intended for the explanation of the field, not to establish experimentally that one architecture is superior



International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026
DOI: 10.17148/IJARCCE.2026.1511132

XI. DIRECTIONS FOR FURTHER STUDY

The immediate priority is a recorded, reproducible literature search followed by a study-by-study evidence table, in many respects. Real repository-version pairs with clear ground truth should be employed by experimental work and include difficult cases such as dynamic imports and behavior changes without signature changes. Checks should distinguish finding a pertinent guide from using it correctly, and a successful edit from a trustworthy migration. Bluntly, studies with developers could then test whether provenance and uncertainty reporting improve decisions in practice.

XII. CONCLUSION

Framework migration entails a chain of judgments about changes, altered code, replacements and remaining risk, broadly speaking; research has continued on individual parts of that chain, but diverse tasks are being outlined by its findings and can't be merged into a single success score, to a significant degree; in fact, the six-stage perspective devised here's offering a way to state what proof exists at each point and where confidence should stop, to a significant degree. The next step is to create more patches, and to assess whether migration recommendations are suitable for the stipulated versions, coherent across the store, and supported by transparent validation.

REFERENCES

[1]. A. Brito, L. Xavier, A. Hora, and M. T. Valente, APIDiff: Detecting API Breaking Changes? IEEE SANER, 2018, pp. 507- 511. doi:10.1109/SANER.2018.8330249.
[2]. K. Cheng et al., CODEMENV: Benchmarking Large Language Models on Code Migration? Findings of ACL, 2025, pp. 2719- 2744. doi:10.18653/v1/2025.findingsacl.140.
[3]. M. Islam, A. K. Jha, M. Mahmoud, and S. Nadi, PyMigTool: A Tool for End-to-End Python Library Migration? arXiv:2510.08810, 2025. Preprint.
[4]. N. Chondamrongkul et al.RepoAI: Automated Code Refactoring through Multi-Agent LLM Orchestration and Retrieval-Augmented Generation, Science of Computer Programming, 2026, article 103477. doi:10.1016/j.scico.2026.103477.
[5]. P. Lewis et al., ?Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks? NeurIPS, vol. 33, 2020.
[6]. N. Reimers and I. Gurevych, Sentence-BERT: Sentence Embeddings employing Siamese BERTNetworks, EMNLP-IJCNLP, 2019.
[7]. T. Chen and C. Guestrin, XGBoost: A Scalable Tree Boosting System, ACM SIGKDD, 2016, pp. 785? 794.
[8]. AI Framework Migration & Compatibility Assistant, Unpublished project design document. Preparatory project report, Pune Institute of Computer Technology, 2026- 2027.

 This work is licensed under a Creative Commons Attribution 4.0 International License