RAILS-DK: Dynamic Knowledge Management for Retrieval-Augmented Code Repair in Scientific Computing

Wali Mohammad Abdullah*[a], Sayeed Islam*[b], Md Morshedul Islam*[a], Baidya Nath Saha*[a], Hossain Shahriar†[c]
*Mathematics & Information Technology, Concordia University of Edmonton, Canada
†Center for Cybersecurity, University of West Florida, USA
[a]{wali.abdullah, mdmorshedul.islam, baidya.saha}@concordia.ab.ca
[b]sislam@student.concordia.ab.ca
[c]hshahriar@uwf.edu

***Abstract*—Retrieval-augmented techniques improve automated code repair by grounding generation in external programming knowledge, but most prior approaches assume a static knowledge base. This assumption is problematic in scientific computing, where libraries, APIs, and dependencies evolve continuously. This paper presents RAILS-DK, a dynamic knowledge management framework for retrieval-augmented code repair that treats programming knowledge as an explicit, evolving artifact. RAILS-DK uses a topic-driven registry to control domain-specific APIs and deterministically regenerate documentation and retrieval indices, ensuring verified, environment-aligned, and reproducible repair. We evaluate RAILS-DK on two benchmarks comprising over six hundred compilation- and execution-validated repair instances across Java and Python. The first extends a Java import-resolution benchmark with a synthetic scientific API to isolate knowledge effects from pretrained memorization; the second evaluates execution-level import failures in representative Python scientific kernels. Across both settings, prompt-only and static retrieval fail on unseen or evolving APIs, while RAILS-DK consistently retrieves correct bindings and achieves successful repair, typically within two iterations. These results show that dynamic knowledge management is critical for reliable retrieval-augmented code repair in scientific computing workflows.**

***Index Terms*—automated code repair, scientific computing software, knowledge management, software maintenance, retrieval-augmented systems**

I. Introduction

Scientific computing software is long-lived, heterogeneous, and tightly coupled to evolving programming ecosystems. Simulation codes, data-analysis pipelines, and HPC applications depend on language runtimes, domain-specific libraries, and legacy APIs. As these dependencies change, failures caused by missing or reorganized imports become frequent and costly, undermining reproducibility.

Large language models (LLMs) have enabled new approaches to automated program repair, including import-resolution. Retrieval-Augmented Generation (RAG) improves repair when accurate external documentation is available [1], and prior work has shown near-perfect accuracy for Java import-resolution under this assumption [2]. However, such success depends on a correct and up-to-date knowledge base, which is difficult to maintain in scientific computing environments.

In scientific computing, this assumption is fragile. Libraries evolve, APIs are deprecated or reorganized, and project-specific abstractions may exist outside public documentation. Static knowledge snapshots therefore become stale, leading to incorrect repairs or unavailable dependencies [3]. Similar findings in adjacent domains such as P4OMP [4] motivate explicit knowledge lifecycle management.

We evaluate RAILS-DK in two settings: an extended Java import-resolution benchmark with synthetic scientific APIs to isolate knowledge effects from pretrained memorization, and a Python instantiation on representative scientific kernels using the same architecture and validation logic without language-specific heuristics [5].

II. Background and Motivation

A. Retrieval-Augmented Code Repair

LLMs perform well on software engineering tasks such as code completion, synthesis, and automated repair [6], [7], but prompt-only repair remains unreliable when correctness depends on precise, environment-specific API and configuration knowledge. This limitation is acute in scientific and HPC software, where evolving libraries and project-local abstractions are often underrepresented in public training data.

RAG addresses this limitation by grounding generation in external knowledge. In code repair, retrieval provides documentation or API references that constrain generation to valid symbols and usage patterns. Prior work shows that retrieval quality directly affects correctness for unresolved-symbol and missing-import tasks [8]–[10]. However, retrieval helps only when the retrieved documentation is accurate, complete, and aligned with the target environment [11], [12].

B. Static Knowledge Assumptions in Prior Work

Most retrieval-augmented repair systems assume a static knowledge base constructed offline and used unchanged during inference. This assumption underlies prior work on Java

2026 IEEE COMPSAC Camera-Ready Version
To cite this paper, please use the final published version
(Check: https://ieeecompsac.computer.org/2026/)

import-resolution and related retrieval-based code repair systems [2], [10]. However, it limits applicability in long-lived scientific software, where APIs are deprecated, libraries reorganized, and recommended usage patterns change over time. When knowledge bases are not updated, retrieval may surface stale or incomplete context, leading to incorrect repairs or unavailable dependencies [13].

C. Motivation for Dynamic Knowledge Management

These limitations motivate treating external knowledge as an explicitly managed, updatable component of retrieval-augmented repair rather than a fixed resource. Recent work has explored selective retrieval and agentic repair loops [14], [15], but generally still relies on static knowledge snapshots. Training-based specialization is also poorly suited to scientific computing, where dependencies and execution contexts evolve continuously; fine-tuning embeds assumptions into model parameters and complicates adaptation and reproducibility [16], [17]. In contrast, scientific workflows routinely manage evolving models, datasets, and software dependencies through explicit versioning and controlled updates, suggesting that programming knowledge should be managed in the same way.

III. Problem Formulation

We formulate import-resolution repair as a context-sensitive program transformation problem under evolving knowledge constraints. Although instantiated for Java and Python, the formulation applies to compilation or execution failures whose resolution depends on external symbol definitions outside the language model.

Let x denote a program that fails to compile or execute due to unresolved symbols, typically caused by missing, deprecated, or incorrect imports. The objective is to generate a repaired program $\hat{x}$ such that $\hat{x}$ compiles (Java) or executes successfully (Python) in the target environment while preserving the intended semantics of x.

Retrieval-augmented repair systems address this problem by augmenting large language models with external knowledge. Given an input program x, a retrieval function $R(\cdot)$ selects a contextual document set C from a knowledge base $\mathcal{K}$, which is then provided to a generation function $G(\cdot)$:

$$\hat{x} = G(x, C), \quad \text{where } C = R(x, \mathcal{K}). \tag{1}$$

Existing retrieval-augmented systems typically assume that $\mathcal{K}$ is static. While effective for well-established APIs, this assumption obscures the relationship between knowledge coverage, retrieval behavior, and repair correctness in environments where libraries and dependencies evolve. In such settings, retrieval may surface incomplete or outdated context, limiting repair reliability even when the repair logic and language model remain unchanged.

To address this limitation, we model knowledge as an explicitly managed, time-dependent state. Let $\mathcal{K}_t$ denote the knowledge base available at update step t. In RAILS-DK, $\mathcal{K}_t$ is constructed deterministically from an explicit topic set $\mathcal{T}_t$:

$$\mathcal{K}_t = f(\mathcal{T}_t). \tag{2}$$

Here, $\mathcal{T}_t$ consists of fully qualified API identifiers, such as classes, packages, or modules, that define the scope of supported programming knowledge. The function $f(\cdot)$ deterministically generates structured documentation artifacts and corresponding retrieval indices from this declared topic set.

This formulation makes retrieval behavior depend on declared knowledge coverage rather than implicit memorization and improves reproducibility by ensuring identical topic declarations yield identical knowledge construction and retrieval behavior.

Our central hypothesis is that, for import-resolution repair in scientific software, retrieval correctness is a stronger determinant of compilation or execution success than model capacity. When accurate symbol-to-package or symbol-to-module bindings are available, repair becomes a constrained transformation problem; missing or outdated knowledge cannot be compensated for by a stronger model. This motivates the dynamic knowledge architecture presented next.

IV. Dynamic Knowledge Architecture and Benchmark Design

A. Dynamic Knowledge Architecture

RAILS-DK extends retrieval-augmented repair by treating external knowledge as an explicitly managed and evolving system component rather than a static documentation snapshot. As formalized in Section III, the knowledge base at update step t is constructed deterministically from an explicit set of declared topics, making knowledge coverage transparent and reproducible.

Each topic corresponds to a fully qualified API element, such as a Java class or a Python module. Topics are stored in a structured registry that serves as the source of truth for knowledge coverage across languages; optional metadata such as deprecation status may also be included.

When the topic registry changes, RAILS-DK regenerates documentation artifacts and rebuilds the retrieval index so retrieval reflects the current software environment. Under fixed system settings, identical topic declarations produce identical artifacts and indices.

RAILS-DK also supports deprecation-aware knowledge updates. Deprecated APIs are explicitly annotated, and recommended alternatives are incorporated during regeneration. This mechanism is particularly relevant for long-lived scientific software systems, where libraries evolve independently of application code.

Figure 1 summarizes the repair pipeline. Compared to prompt-only and static retrieval-based configurations, RAILS-DK introduces an explicit knowledge update stage prior to repair. The repair logic itself remains unchanged; only the retrieved knowledge evolves.

Unlike ad hoc corpus curation, RAILS-DK maintains a versioned knowledge state with rollback support, enabling

2026 IEEE COMPSAC Camera-Ready Version
To cite this paper, please use the final published version
(Check: https://ieeecompsac.computer.org/2026/)

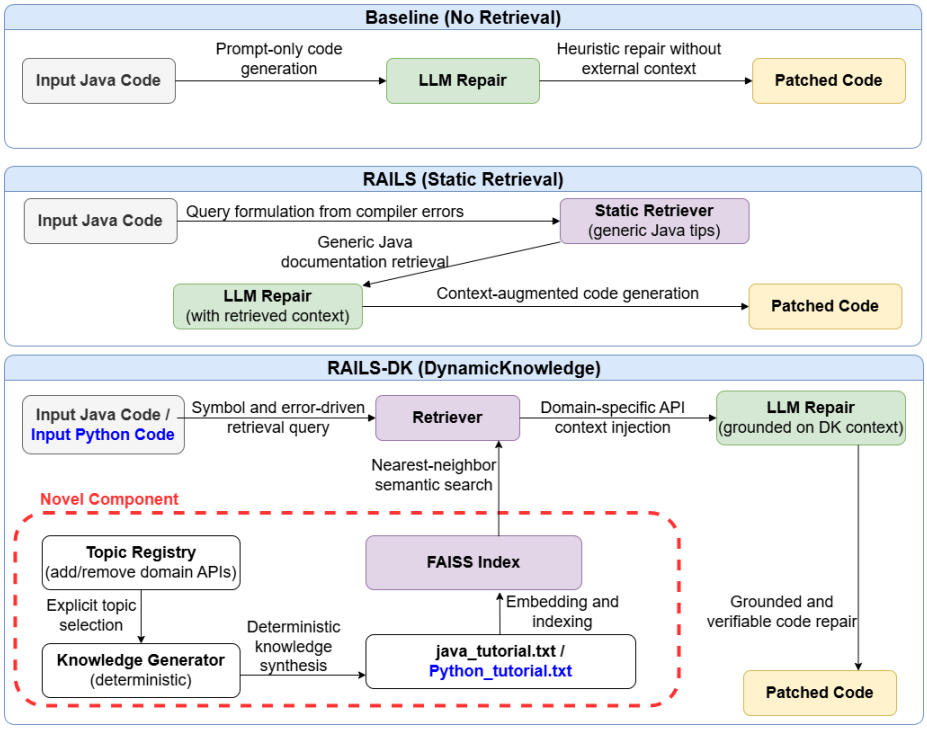

Fig. 1. Retrieval-augmented code repair pipeline comparing prompt-only, static retrieval, and dynamic knowledge-enabled configurations.

repair outcomes to be reproduced under the exact dependency configuration in which they were generated.

B. Operational Model and Scalability

Dynamic knowledge management in RAILS-DK is treated as an offline, amortized process. Knowledge regeneration and index rebuilding are triggered only when the topic registry changes, such as after dependency updates or library version changes. As a result, dynamic knowledge management does not affect the latency of individual repair attempts.

The size of the knowledge base grows linearly with the number of declared topics rather than with the volume of available documentation. This bounded growth aligns well with scientific computing practice, where only a narrow subset of domain-specific APIs is typically relevant to a given project.

C. Benchmarks

We evaluate RAILS-DK using two complementary benchmarks designed to assess both causal correctness and architectural generality.

a) Synthetic Java Benchmark.: To isolate the effect of dynamic knowledge updates from pretrained model memorization, we introduce a synthetic scientific Java library, `scidk`, which is absent from public documentation and model training data. Import-resolution tasks depend exclusively on these APIs. Successful repair, therefore, requires explicit retrieval of correct API-to-package mappings from the dynamically generated knowledge base.

b) Python Scientific Computing Benchmark.: To assess language generality and practical relevance, we instantiate the same architecture for Python. We construct representative scientific kernels drawn from numerical linear algebra, spectral analysis, numerical integration, and stochastic computation. Each kernel omits a required import or symbol binding, triggering runtime failures characteristic of real scientific workflows. The Python evaluation reuses the same topic registry mechanism, deterministic documentation generation, FAISS-based retrieval, and execution-driven validation loop. No language-specific heuristics are introduced.

V. Experimental Methodology

A. Benchmarks

We evaluate RAILS-DK using two complementary benchmarks designed to balance causal isolation with practical relevance.

a) Java Benchmark.: The Java evaluation includes 78 real-world import-resolution cases from prior work [2], covering failures such as missing imports, deprecated APIs, and ambiguous references. To evaluate scenarios beyond static coverage, we extend this benchmark with systematically generated variants based on a synthetic scientific Java library (`scidk`) absent from public documentation and model training data. Successful repair across these compilation-validated instances therefore requires explicit retrieval of correct API-to-package mappings from dynamically generated knowledge artifacts.

b) Python Benchmark.: For Python, we instantiate the same architecture and evaluate over two hundred execution-validated import-resolution failures derived from representa-

tive scientific kernels spanning numerical linear algebra, spectral analysis, numerical integration, and stochastic computation. Each instance omits a required import or symbol binding. This evaluation is intended as an architectural portability check rather than a comprehensive comparison of Python repair techniques.

B. Experimental Configurations

We evaluate three system configurations:

- **Prompt-only**: Prompt-only repair without retrieval.
- **Static Retrieval**: Retrieval-augmented repair using a fixed knowledge base.
- **RAILS-DK**: Retrieval-augmented repair with dynamically regenerated, topic-driven knowledge artifacts.

Retrieval uses FAISS with top-$k = 1$ nearest-neighbor search over 300-token chunks with 50-token overlap, OpenAI `text-embedding-3-small` embeddings, and a flat index; generation uses `gpt-3.5-turbo` with temperature set to 0.2.

C. Evaluation Metrics

We report four metrics to separate retrieval behavior from downstream generation:

- **Retrieval correctness**: presence of the required API or module binding in retrieved context.
- **Repair success**: successful compilation (Java) or execution (Python).
- **Repair efficiency**: number of iterations to reach success.
- **Hallucination rate**: frequency of invalid or environment-incompatible APIs.

VI. Results and Discussion

A. Illustrative Repair Workflow

Figure 2 illustrates the execution-driven repair workflow, showing the interaction between failure detection, dynamic retrieval, repair generation, and validation. Although the example is drawn from the Python evaluation, the same workflow is used for Java; only the retrieved artifacts differ.

Upon a compilation or execution failure, the system retrieves a verified API binding for the unresolved symbol and injects it into the repair prompt. The resulting candidate fix is then validated through compilation (Java) or execution (Python); only successful repairs are persisted, while failed attempts generate new error signals for subsequent iterations. This validation discipline is applied uniformly across both languages.

Repairs are not accepted speculatively. Each candidate is validated through compilation (Java) or execution (Python), and only successful repairs are persisted. Failed attempts generate new error signals that drive subsequent iterations. This execution-driven validation discipline prevents partially correct or environment-incompatible fixes from being archived and is applied uniformly across both languages.

B. Retrieval Quality

To assess retrieval behavior independently of generation success, we define retrieval correctness as:

$$Q(C) = \begin{cases} 1, & \text{if the required API binding is present in } C \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

a) Incomplete Knowledge Sensitivity.: To evaluate sensitivity to incomplete topic specification, we ran RAILS-DK with one required API (`scidk.util.Timer`) intentionally omitted from the topic registry. Under this setting, retrieval correctness dropped from 100% to 90%, and compilation success decreased to 95%. The single failure occurred in the case that explicitly depended on the omitted API, while all other cases compiled successfully, requiring an average of 1.05 iterations. Importantly, failures were attributable to the explicitly missing topic rather than hallucinated or environment-incompatible bindings, confirming that RAILS-DK degrades predictably when knowledge coverage is incomplete.

A similar pattern is observed for Python, where retrieval consistently surfaces correct module-level bindings once the relevant topics are registered.

b) Real-World Python API Drift Case Study.: We evaluated RAILS-DK on a documented Python API removal in which `scipy.misc.imread` is no longer available in recent SciPy versions. Prompt-only and static retrieval configurations failed to resolve the import. After updating the topic registry to reflect the replacement binding (`imageio.v2.imread`), RAILS-DK retrieved the correct API definition and proposed a valid repair. Execution succeeded once the corresponding dependency was present in the environment, consistent with RAILS-DK's design choice to avoid installing or modifying external packages.

C. Repair Success and Efficiency

Figure 3 summarizes repair outcomes for prompt-only, static retrieval, and dynamic knowledge configurations across the Java and Python benchmarks. Across 400 Java compilation-validated instances and 200 Python execution-validated instances, prompt-only and static retrieval fail when required bindings are absent, whereas RAILS-DK succeeds by grounding generation in explicitly managed and regenerated knowledge artifacts. The same trend appears in the Python API drift case.

TABLE I
Performance on the synthetic scientific API repair benchmark (Java).

System	Retrieval Correctness (%)	Compilation Success (%)	Avg. Iterations (success)	Hallucination (%)
Prompt-only	0	10	1.0	high
Static Retrieval	0	25	1.4	moderate
RAILS-DK	100	100	1.0	0

Table I summarizes performance on the base benchmark. Prompt-only repair resolves only a small fraction of cases and frequently hallucinates invalid imports. Static retrieval improves success but remains limited by incomplete domain

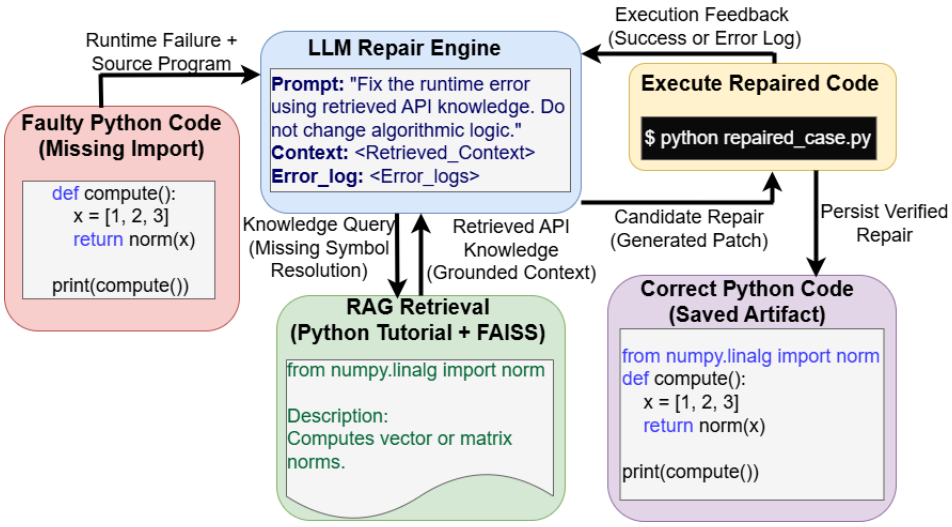

Fig. 2. Execution-driven repair workflow in RAILS-DK. The same pipeline is used for Java and Python, differing only in the indexed knowledge artifacts.

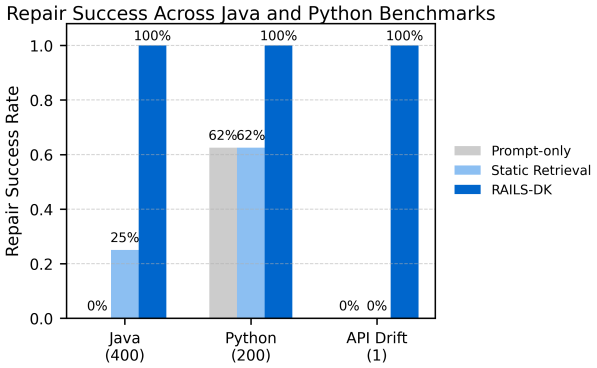

Fig. 3. Repair success rates across Java and Python benchmarks for prompt-only, static retrieval, and RAILS-DK (400 Java, 200 Python, 1 API drift case).

knowledge. In contrast, RAILS-DK achieves 100% compilation success and converges in a single iteration by grounding repair in verified API bindings. These results remain invariant under large-scale evaluation: across 400 systematically generated Java variants, RAILS-DK repaired all cases successfully in the first iteration, with no retrieval errors or hallucinated imports observed.

For Python, RAILS-DK repaired all 200 execution-validated instances successfully. Most repairs converged within two iterations and the remainder within three. In contrast, the prompt-only configuration succeeded on only 62.5% of cases. As in Java, all accepted repairs were execution-validated and no speculative fixes were persisted.

D. Discussion

Across Java and Python, the results support the central claim of this work: reliable maintenance of scientific software requires precise, verifiable external knowledge, and managing that knowledge is as important as the repair logic itself. By modeling knowledge coverage through a topic registry and regenerating retrieval artifacts deterministically, RAILS-DK turns retrieval from a heuristic aid into a controlled grounding mechanism. The Java benchmark demonstrates this effect under conditions that eliminate pretrained memorization, while the Python evaluation shows that the same architecture operates unchanged on realistic scientific workloads.

Dynamic knowledge management is especially important when dependency evolution invalidates previously correct retrieval. When libraries reorganize APIs or deprecate symbols, static knowledge snapshots continue to surface obsolete bindings or require manual reconstruction. In RAILS-DK, such changes are expressed as topic updates that trigger deterministic regeneration of documentation and retrieval indices, grounding subsequent repairs in the updated environment while preserving reproducibility of earlier experiments. Compared with periodic full re-indexing without an explicit topic registry, this approach also constrains knowledge scope and preserves historical knowledge states.

Sensitivity and Update Cost.: RAILS-DK makes knowledge boundaries explicit through its topic registry, so topic granularity affects retrieval precision and recall in a transparent and controllable way. Knowledge updates incur an offline regeneration cost, but in our experiments updating 20-30 APIs and rebuilding documentation and a FAISS index required under one minute on a local WSL-based environment (Intel $i7$-$8650U$, 16 GB RAM, Ubuntu 24.04), including documentation generation (20-25s), embedding (25-30 s), and index construction (under 10 s). This process is triggered only by dependency version changes and can be integrated into CI/CD pipelines.

VII. Threats to Validity and Reproducibility

a) Internal Validity.: The synthetic Java benchmark abstracts away complexities such as deep dependency graphs, heterogeneous build systems, and environment-specific configuration issues. This abstraction is intentional: it isolates

TABLE II
FACTORS INFLUENCING RETRIEVAL-AUGMENTED REPAIR BEHAVIOR IN RAILS-DK.

Factor	Primary Effect	Controlled By
Topic granularity	Retrieval precision/recall	Topic registry
API evolution	Context drift	Knowledge regeneration
Embedding model	Retrieval similarity	Model choice
Knowledge update cost	Offline regeneration overhead	Topic registry

the effect of knowledge availability on retrieval and repair behavior without confounding factors such as installation or version-resolution errors. The Python evaluation partially mitigates this limitation by instantiating the same architecture on representative scientific workloads with realistic execution-driven failures.

b) External Validity.: Our evaluation focuses on import- and binding-resolution failures in Java and Python. Although these do not exhaust the error modes found in scientific software, knowledge-evolution issues also arise in other ecosystems, including C/C++, domain-specific languages, and scientific frameworks. The topic-driven, deterministic knowledge management approach introduced by RAILS-DK is therefore applicable beyond Java and Python, though validation in those settings is left for future work [18].

c) Sensitivity to Knowledge Specification.: Retrieval behavior depends on how topics are specified. Overly coarse definitions may introduce irrelevant context, while incomplete topic sets may omit required dependencies. This sensitivity is intentional and transparent, enabling diagnosis and correction through declared knowledge boundaries rather than implicit model memorization.

d) Reproducibility.: All source code, experimental data, execution logs, and repaired artifacts are publicly available [19]. Knowledge updates are scripted, documentation regeneration is deterministic, retrieval indices are rebuilt from version-controlled artifacts, and rollback support ensures identical experimental starting states across runs.

VIII. CONCLUSION

This paper presented **RAILS-DK**, a dynamic knowledge management framework for retrieval-augmented code repair in scientific computing environments with evolving dependencies. By treating external programming knowledge as an explicitly managed and regenerable artifact, RAILS-DK provides reproducible, environment-aligned grounding beyond static documentation snapshots. Experiments on synthetic Java benchmarks and a complementary Python instantiation show that dynamic knowledge availability is decisive for resolving import- and binding-related failures: prompt-only and static retrieval fail on unseen or evolving APIs, whereas RAILS-DK achieves reliable compilation and execution success using the same repair architecture across languages. These findings show that robust AI-assisted maintenance of scientific software depends not only on model capacity, but also on explicit lifecycle management of external knowledge.

ACKNOWLEDGMENT

This research is funded by the generous support of Concordia University of Edmonton under the Seed Grant Program (CRG-SEED-2501-02).

REFERENCES

[1] Z. Li, Z. Wang, W. Wang, K. Hung, H. Xie, and F. L. Wang, "Retrieval-augmented generation for educational application: A systematic survey," *Computers and Education: Artificial Intelligence*, p. 100417, 2025.

[2] W. M. Abdullah, M. M. Islam, D. Parmar, H. H. Patel, S. Prabhakaran, and B. Saha, "RAILS: Retrieval-augmented intelligence for learning software development," in *2025 IEEE High Performance Extreme Computing Conference (HPEC)*, 2025, pp. 1–6.

[3] I. Saberi and F. Fard, "Context-augmented code generation using programming knowledge graphs," *arXiv preprint arXiv:2410.18251*, 2024.

[4] W. M. Abdullah and A. Kabir, "P4OMP: Retrieval-augmented prompting for openmp parallelism in serial code," in *2025 IEEE High Performance Extreme Computing Conference (HPEC)*, 2025, pp. 1–6.

[5] A. Nagy, Y. Spyridis, and V. Argyriou, "Cross-format retrieval-augmented generation in xr with llms for context-aware maintenance assistance," in *2025 IEEE International Conference on Artificial Intelligence and eXtended and Virtual Reality (AIxVR)*. IEEE, 2025, pp. 355–361.

[6] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[7] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[8] Z. Yang, S. Chen, C. Gao, Z. Li, X. Hu, K. Liu, and X. Xia, "An empirical study of retrieval-augmented code generation: Challenges and opportunities," *ACM Transactions on Software Engineering and Methodology*, 2025.

[9] H. Li, X. Zhou, and Z. Shen, "Rewriting the code: A simple method for large language model augmented code search," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024, pp. 1371–1389.

[10] D. Wu, W. U. Ahmad, D. Zhang, M. K. Ramanathan, and X. Ma, "Repoformer: selective retrieval for repository-level code completion," in *Proceedings of the 41st International Conference on Machine Learning*, 2024, pp. 53 270–53 290.

[11] Y. Zhao, S. Chen, J. Zhang, and Z. Li, "Recode: Improving llm-based code repair with fine-grained retrieval-augmented generation," in *Proceedings of the 34th ACM International Conference on Information and Knowledge Management*, 2025, pp. 4368–4378.

[12] Z. Z. Wang, A. Asai, F. F. Xu, Y. Xie, G. Neubig, D. Fried *et al.*, "Coderag-bench: Can retrieval augment code generation?" in *Findings of the Association for Computational Linguistics: NAACL 2025*, 2025, pp. 3199–3214.

[13] T. Zhang, D. Yang, C. Lopes, and M. Kim, "Analyzing and supporting adaptation of online code examples," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 316–327.

[14] M. Bhattarai, M. Cordova, M. Vu, J. Santos, I. Boureima, and D. O'Malley, "Arcs: Agentic retrieval-augmented code synthesis with iterative refinement," *arXiv preprint arXiv:2504.20434*, 2025.

[15] J. Han, Z. Mao, Y. Liu, Y. Che, Z. Fu, and Q. Wang, "Fine-grained knowledge enhancement for retrieval-augmented generation," in *Findings of the Association for Computational Linguistics: ACL 2025*, 2025, pp. 10 031–10 044.

[16] A. E. Hassan, D. Lin, G. K. Rajbahadur, K. Gallaba, F. R. Cogo, B. Chen, H. Zhang, K. Thangarajah, G. Oliva, J. Lin, W. M. Abdullah, and Z. Ming (Jack) Jiang, "Rethinking software engineering in the era of foundation models: A curated catalogue of challenges in the development of trustworthy fmware," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 294–305.

[17] X. Chen, Y. Li, and X. Wang, "Design principles and guidelines for llm observability: Insights from developers," in *Proceedings of the Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, 2025, pp. 1–9.

[18] Y. Wang, S. Guo, and C. W. Tan, "From code generation to software testing: Ai copilot with context-based rag," *IEEE Software*, 2025.

[19] https://anonymous.4open.science/r/rails-dynamic/, 2026.