Pairing LLM4KG and KG4LLM: Framework-Specific Knowledge Graphs for Semantically Correct Repository-Level Code Generation

Anjana C. Tissera[1,*], Chantri Polprasert[1] and Mongkol Ekpanyapong[1]

[1]Asian Institute of Technology, 58 Moo 9, Km. 42, Paholyothin Highway, Khlong Luang, Pathum Thani 12120, Thailand

Abstract

Large language models (LLMs) generate fluent code but struggle at the repository level, where changes must stay consistent across interdependent files and project conventions: they hallucinate APIs, substitute libraries and misuse state and data flow. Graph-based retrieval helps, but existing code graphs are built by static analysis over shallow, language-level schemas. This paper pairs LLM4KG with KG4LLM and makes three contributions: (1) an LLM-assisted, few-shot method that extracts a knowledge graph directly from source code, without framework-specific parser engineering; (2) a framework-specific schema for React.js with 11 node types and 16 relationship types; and (3) KG-grounded repository-level code generation, evaluated against no-context, oracle whole-file, BM25 and dense-retrieval baselines. Extraction reaches micro-averaged precision, recall and F1 of 0.95 on 28 hand-annotated files from five repositories. On six generation tasks, no context passes nothing. With Claude Sonnet 4.6, KG context passes 25/30 using a median of 699 tokens, while BM25 and dense retrieval pass 30/30 using medians above 6,000. At the KG's own per-task budget they fall to 18/30 and 15/30. With Qwen3-Coder no condition ordering holds, so the benefit depends on the generator. KG context is therefore competitive at 8–9× fewer tokens rather than superior, and the graph pays for itself only when reused across many tasks. These results come from a pilot-scale study; a larger controlled study is under way.

Keywords

Knowledge graph, Graph-based AI, Knowledge extraction, Retrieval-augmented generation, Large language models, LLM-assisted code generation

CEUR-WS.org/Vol-4316/pd-short-2.pdf

1. Introduction

Large Language Models (LLMs) can translate natural language descriptions into functional programs and perform well on isolated benchmarks such as HumanEval [1] and MBPP [2]. This fluency does not extend to repositories. Feature additions, defect corrections and refactorings in codebases with interdependent files and shared conventions require each modification to stay consistent across the project. At this scale, LLMs may hallucinate non-existent APIs, substitute libraries without indication, and misuse state-management and data-flow patterns, producing code that is syntactically correct but semantically flawed. The core issue is a mismatch between repository size and the LLM's effective context window: projects often exceed it, and supplying raw source code degrades generation quality. Existing approaches supply context through similarity-based retrieval (retrieval-augmented generation, RAG), which can return textually similar rather than structurally relevant code [3], or through costly, manually constructed tool interfaces.

Representing the repository as a graph is more promising, and recent systems [4, 5, 6, 7, 8] demonstrate that structured, dependency-aware context measurably improves repository-level reasoning. However, a consistent gap remains. All such systems construct their graphs using Abstract Syntax Tree (AST) parsing or static analysis, focus on Python, and employ generic schemas with three to seven node types. None capture the semantics of modern application frameworks, utilize LLMs for graph extraction, or demonstrate adaptability across frameworks.

IJCKG 2026 Posters, Demos and Workshops, November 19–21, 2026, Bangkok, Thailand

*Corresponding author.

st123459@ait.ac.th (A. C. Tissera); chantri@ait.ac.th (C. Polprasert); mongkol@ait.ac.th (M. Ekpanyapong)

0009-0007-0379-7202 (A. C. Tissera); 0000-0003-4312-3580 (C. Polprasert); 0000-0002-0192-6249 (M. Ekpanyapong)

© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-06

This work addresses this gap by integrating two complementary approaches centered on a single knowledge graph. In **LLM4KG**, an LLM prompted with few-shot examples processes source code and generates a deep, *framework-specific* knowledge graph directly, without reliance on AST parsing. This method captures application-level abstractions such as components, props, state, hooks, contexts, routing, and library provenance, rather than restricting itself to language-level symbols. In **KG4LLM**, context retrieved from this knowledge graph grounds the generation prompt, ensuring that the produced code adheres to the project's actual entities and conventions. Because extraction relies on the LLM's reading of the code rather than on a custom parser, adapting to another framework is designed to need only a new schema and few-shot examples, not a new parser; this design property is not yet evaluated. The methodology is instantiated on React.js, a structurally rich and widely adopted framework that has been largely overlooked in repository-graph research. Additionally, the knowledge graph serves as a reusable semantic index for the repository.

1.1. Contributions

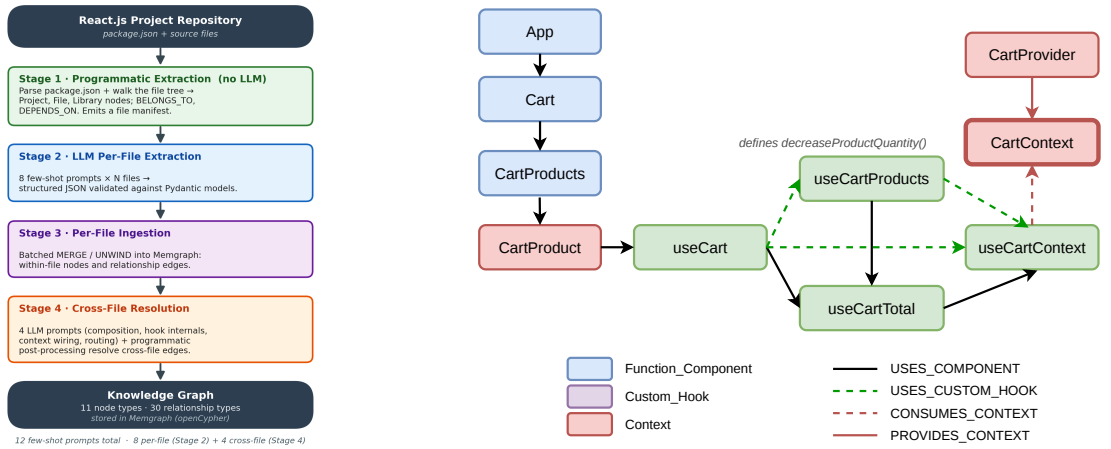
(C1) An LLM-assisted, few-shot method that extracts a knowledge graph directly from source code, without framework-specific parser engineering; on 28 hand-annotated files from five repositories it reaches micro-averaged precision, recall and F1 of 0.95. **(C2)** A framework-specific schema for React.js with 11 node types and 16 relationship types, designed from the framework's own abstractions on a project/file/library backbone, with its coverage measured against real code changes. **(C3)** KG-grounded repository-level code generation, evaluated against no-context, oracle whole-file, BM25 and dense-retrieval baselines, with a budget-matched ablation and a construction-cost and break-even analysis.

2. Related Work

Several systems represent a repository as a graph so an LLM can reason over explicit relationships rather than flat text [9]: GraphCoder builds a control-/data-flow context graph for completion [5], RepoGraph a line-level definition/reference graph for agentic frameworks [4], CodexGraph indexes a repository into a graph database queried by LLM-written Cypher [6], and KGCompass targets bug localization and repair [7]. Closest to us, Athale and Vaddina apply a knowledge graph to repository-level code *generation* with a competitive pass@1 score on EvoCodeBench [10, 8]. In all of them, the graph is built statically over a generic, language-level schema and targets Python. Where an LLM appears, as in Athale and Vaddina, it only writes retrieval descriptions over an otherwise statically built graph rather than extracting the graph. Published code-KG schemas are correspondingly shallow, about three node types in GraphGen4Code [11], six in FalkorDB's code graph [12], and roughly four relationship types in RepoGraph; ours has 11 node types and 16 relationship types. Finally, Meyer et al. [13] show that few-shot rather than zero-shot prompting is needed for dependable KG extraction, and work translating natural language to Cypher over repository metadata [14] backs up both graph grounding and few-shot reliability. None of this work is framework-aware or demonstrates cross-framework adaptability.

3. System Overview

Our system pairs two directions around a single knowledge graph. In **LLM4KG**, a four-stage pipeline reads a React.js repository and emits a typed, framework-specific graph into a Memgraph store. In **KG4LLM**, a retrieval component queries that graph to assemble grounding context for a code-generation prompt. Both live in one Python application over shared infrastructure that uses Memgraph over Bolt, an Instructor-over-LiteLLM client that validates every response against Pydantic schemas, Jinja2 prompts, and a Typer CLI. The model is reached through one provider-agnostic string, so swapping providers is a one line change. Figure 1(a) shows the extraction pipeline.



(a) (b)

Figure 1: (a) The four-stage LLM4KG extraction pipeline: structural extraction, per-file LLM extraction, ingestion, and cross-file resolution. (b) Cart-context knowledge-graph excerpt (pilot task P1): `CartProduct` reaches cart state through custom hooks and a shared context. Node fill/border denote node type; edge colour/dash denote relationship type.

3.1. Knowledge Graph Extraction (LLM4KG)

A deep, framework-specific schema. The graph is typed by 11 node types and 16 relationship types (Table 1), against the three to seven node types of published code-KG schemas. Nodes are named with abstractions a React developer works with (components, props, state, hooks, contexts, routing, and library provenance) rather than language-level symbols. Since Memgraph lacks composite unique constraints, every multi-field node carries a synthetic `uid` computed at insertion, so repeated MERGE runs are idempotent.

How the schema was designed and checked. The schema was designed top-down from React's core concepts (components, props, state, hooks, context and routing) on a framework-neutral project/file/library backbone. Its first version was then checked against a coverage map of 16 curated open-source React projects that vary in size and feature use, and each gap became an addition with a recorded rationale: class components, which several of the projects contain; event handlers; library provenance (`Library`, `PROVIDED_BY`); the direction of prop flow (`PASSES_PROP`); context edges; and custom-hook internal edges. The counts are therefore a design outcome, not a measured optimum. The design document counts 30 relationship types, listing function- and class-component endpoints separately; as implemented, the 16 types admit 36 source-target combinations, of which 26 occur in the five extracted graphs. Section 5.2 tests sufficiency empirically.

A four-stage extraction pipeline. Extraction is deterministic where possible and LLM-driven only where semantics demand it. Stage 1 (programmatic) parses `package.json` and the file tree into `Project`, `File`, and `Library` nodes with a file manifest. Stage 2 runs eight few-shot prompts per file and extracts function and class components, custom hooks, props and event handlers, imports, function/hook and class state, and contexts, returning Pydantic-validated JSON. Stage 3 ingests nodes and within-file edges via batched `MERGE`/`UNWIND` Cypher, roughly two orders of magnitude faster than row-at-a-time inserts. Stage 4, which needs the full node inventory, runs four cross-file prompts (composition and prop flow, custom-hook internals, context wiring, routing), after which a programmatic pass resolves import paths, joins library hooks to libraries, and flags the root component. We use 12 prompts in total: 8 per file and 4 across files.

This division of labor confines cost and non-determinism to the steps that need language understanding, and lets the pipeline retarget a framework by exchanging prompts and schema rather than

re-engineering a parser.

Table 1
The React.js knowledge-graph schema: 11 node types and 16 relationship types. A designed file-level IMPORTS edge was not built.

Category	Node types	Relationship names
Structure	`Project`, `File`, `Library`	BELONGS_TO, DEPENDS_ON, RENDERS_ROOT_COMPONENT
Components	`Function_Component`, `Class_Component`	DEFINED_IN, USES_COMPONENT
Props & events	`Prop`, `EventHandler`	ACCEPTS_PROP, PASSES_PROP, HAS_HANDLER
Hooks	`Library_Hook`, `Custom_Hook`	USES_LIBRARY_HOOK, USES_CUSTOM_HOOK, PROVIDED_BY
State	`State_Variable`	DECLARES_STATE, DECLARES_CLASS_STATE
Context & routing	`Context`	PROVIDES_CONTEXT, CONSUMES_CONTEXT, ROUTES_TO

3.2. KG-Grounded Code Generation (KG4LLM)

KG4LLM turns the graph into grounding context under a deliberately narrow, *retrieval-only* contract: across conditions the generator, its temperature and its prompt are fixed, and only the context varies, so any change in output is attributable to context.

Classification and anchors. One few-shot LLM call, the *classifier*, returns JSON at temperature 0 with the task type (bug fix, feature addition or refactoring), the anchor names (the components or custom hooks the task centres on) and any route paths it mentions. The response is validated against a Pydantic model; a malformed one falls back to a bug fix with no anchors. The task type selects one of three parameterised Cypher templates, with auxiliary queries adding a routing subgraph for features and caller-side props for refactorings. Anchors are bound as query parameters and matched exactly and case-sensitively, so they act as a hard filter rather than a ranking signal. The result renders anchor components with full source (read from disk via the graph's `filePath`) and neighbours as compact metadata, so the budget goes to the code most likely to be edited.

Wrong and ambiguous anchors. Because anchors filter, their failure modes matter. An anchor the graph does not hold returns no rows, and the context degrades to a 23–55-token header, effectively the no-context floor; this did not occur in the reported runs (Section 5.3). An opt-in hardened resolver falls back from exact to case-insensitive to fuzzy matching (similarity ≥ 0.80), records each resolution per anchor in the retrieval metadata, and, when nothing resolves, returns a flagged project overview of 198–594 tokens. Given deliberately misspelled anchors, it recovers the intended context exactly. Its handling of ambiguous names, such as two components with the same name, is covered by unit tests only, since no such ambiguity arose in our repositories. A wrong but real anchor, naming an existing but irrelevant component, resolves cleanly and cannot be detected from the graph alone. The reported results use the strict resolver.

4. Demonstration

The demonstration walks the full loop on a live repository using `react-shopping-cart`, a TypeScript React storefront; a second project, `TakeNote`, is configured identically. **Extract:** on 55 source files, the pipeline produces **132 nodes and 159 relationships** in Memgraph, extracted end to end by Claude

Sonnet 4.6 at temperature 0 with the twelve-prompt set (`TakeNote`: 398 nodes, 636 relationships, 60 files); each graph is committed as a CYPHERL dump, reloadable without re-running any LLM calls. Figure 1(b) shows the cart-context neighborhood, where `CartProduct` reaches cart state through custom hooks and a shared React context. **Retrieve:** given a bug in which decreasing a cart item to a quantity of zero leaves a $0.00 line item instead of removing the product, the classifier identifies a bug fix and the template returns a **626-token** context block centred on `useCartProducts`, the hook that defines the offending `decreaseProductQuantity` with neighboring cart-context hooks as metadata. **Generate:** the model emits a minimal SEARCH/REPLACE edit that removes zero-quantity products, reusing the project's own symbols:

```
const updatedProducts = products
  .map((product: ICartProduct) => {
    return updateQuantitySafely(product, productToDecrease, -1);
  })
  .filter((product: ICartProduct) => product.quantity > 0);
```

Verify: the harness applies the edit matched uniquely and placement-verified into a Docker image and runs the task's test suite, which passes (exit code 0); given no graph context, the same generator produces no applicable edit, because nothing supplied the real path, hook, and symbols the change depends on. Extracted graphs, pilot tasks, generated candidates are in the project repository.[1]

5. Evaluation

The evaluation is a pilot: it asks whether each part of the system works and what it costs, not how large the effects are.

5.1. Setup

Tasks and generators. Six tasks cover two of each type, three per repository: bug fixes P1 (`react-shopping-cart`) and P6 (`TakeNote`), feature additions P4 and P2, and refactorings P5 and P3. Each has a reference patch and a test validated in a Docker image of its repository. Two generators, Claude Sonnet 4.6 and the open-weight Qwen3-Coder-480B, emit SEARCH/REPLACE edits at temperature 0.0, five candidates per cell. At temperature 0 a cell's candidates are near-identical, so they are repetitions rather than independent samples.

Conditions. Five of the six designed conditions were run; the retriever for Nomic Embed Code is still being implemented. The generator, its prompt, the context renderer and the output format are fixed, and only the context varies: no context (*floor*); *whole-file*, an oracle given the files the reference patch edits and deliberately uncapped, so an upper bound rather than a competitor; *KG-augmented* (Section 3.2); BM25 [15]; and dense retrieval with `text-embedding-3-large` [16]. BM25 and dense index the KG's own file set, one chunk per file, and add whole files in rank order up to 7,000 tokens. Because the KG context is far smaller, a budget-matched ablation re-runs BM25 and dense with each task capped at the KG's own context size for that task (371–1,532 tokens); the KG and oracle are not re-run. The ablation separates selection from budget.

Grading. An automated Docker harness applies each edit, builds the project and runs the task's test, recording the furthest stage reached: *never applied*, *build failure*, *test failure* or *passed.* There is no human grading. The main runs comprise 300 candidates (2 generators $\times$ 5 conditions $\times$ 6 tasks $\times$ 5), none of which ended in a harness error, and the ablation adds 120.

[1] Project repository: https://github.com/LLM4KG/codebase-kg/tree/1-ijckg-2026

5.2. Extraction Accuracy and Schema Coverage

Ground truth is 28 hand-annotated source files from five repositories: the two above plus `SnapShot`, `Todoist Clone` and `Jira Clone`, sampled with a fixed seed and stratified so that every node and relationship type present in the graphs occurs at least once. Table 2 scores the LLM-extracted types only. The deterministic types (`Project`, `File`, `Library` and their edges) score 1.00 over 78 items and are kept out of the headline so they cannot inflate it.

Table 2
Extraction accuracy on 28 hand-annotated files, LLM-extracted node and relationship types only (micro-averaged).

Repository	Source files	Files annotated	TP	FP	FN	P	R	F1
SnapShot	15	5	22	1	4	0.96	0.85	0.90
Todoist Clone	19	5	95	0	5	1.00	0.95	0.97
react-shopping-cart	55	6	49	2	0	0.96	1.00	0.98
TakeNote	60	6	108	9	8	0.92	0.93	0.93
Jira Clone (client)	134	6	70	8	0	0.90	1.00	0.95
Micro-average	**283**	**28**	**344**	**20**	**17**	**0.95**	**0.95**	**0.95**

Micro-averaged precision, recall and F1 are all 0.95. The weakest types are `Prop` (F1 0.86), `EventHandler` (0.88), ROUTES_TO (recall 0.43) and USES_CUSTOM_HOOK (recall 0.38). Most of the error has one cause: whether an item belongs to a component or is only passed down to a child. Eighteen of the 20 false positives fall in `Prop`, `EventHandler` and their edges, and a cold re-extraction differs from the committed graphs only on those types. All five USES_CUSTOM_HOOK misses come from one re-exporting barrel file that the deterministic import resolver does not follow, so the LLM's own share of the 17 false negatives is 12. Three biases apply: annotators corrected drafts pre-filled from the extractor's output, which favours recall despite a blind first reading; a second annotator covered only 4 files (Cohen's κ [17] 1.00 on three, and 0.46 on one where the two agreed on 17 of 19 items); and the sample is small.

Schema coverage. To test whether 11 node and 16 relationship types are sufficient, every unit of change in the six reference patches was classed as fully, partially or not representable in the schema. Of 32 units, 22 are fully representable and 28 at least partly; the four that are not (a Redux reducer and action, an enum member, a utility import, a styled component) lie outside React's component model. No annotated false negative was caused by a missing type: the gaps surface instead as the precision loss above. Four gaps are named: **G1** Redux state and actions; **G2** functions that are neither handlers nor components, such as those a hook returns, the most frequent gap (5 of 6 tasks); **G3** symbols inside non-React modules; and **G4** styled-component wrappers. All 11 node types and 15 of the 16 relationship types occur in the five graphs (1,315 nodes and 2,011 edges over 283 files); DECLARES_CLASS_STATE never fires, as the corpus holds one class component.

5.3. Generation Outcomes

Table 3 gives the outcomes. **No context passes nothing**: 0 of 60 candidates on either generator, all never applied, because the model invents paths and contents. **At 7,000 tokens the stronger model saturates**: the oracle, BM25 and dense all pass 30/30, so this view separates conditions only where one falls short. The KG passes 25/30 on Claude with a median of 699 context tokens against 6,267 and 6,412, about 8–9$\times$ less. Its whole deficit is one task, P4, and it is a localisation miss rather than a budget effect: the oracle passes P4 with 599 tokens, while the KG fails with 371. On P2 the Redux action the change needs is absent from the KG context (G1); the stronger model supplies it unaided, and the open model does not.

Table 3
Passes out of 30 (6 tasks × 5 candidates) per condition and generator. Context tokens are per task. The oracle is uncapped. Rows below the rule are the budget-matched ablation: BM25 and dense capped at the KG's per-task context (371–1,532 tokens). [a]20/30 if each edit is applied individually rather than all-or-nothing; the other three matched totals do not change.

Condition	Context tokens per task (range)	Median across 6 tasks	Claude Sonnet 4.6	Qwen3-Coder
No context (floor)	0	0	0/30	0/30
Whole-file oracle (uncapped)	278–2,611	649	30/30	17/30
KG-augmented	**371–1,532**	**699**	**25/30**	**16/30**
BM25 @ 7,000 tokens	4,602–6,964	6,267	30/30	22/30
Dense @ 7,000 tokens	5,905–6,981	6,412	30/30	23/30
BM25 @ KG's budget	358–1,500	664	18/30	15/30
Dense @ KG's budget	358–1,497	664	15/30[a]	18/30

At the KG's own budget, BM25 on Claude falls to 18/30 and dense to 15/30, and the KG never loses a task to either. Outcomes then follow localisation quality, from the oracle through the KG to lexical and dense retrieval. Equal tokens are not equal information, however. The KG's context summarises several components, while the same budget holds a few raw files, so the two views bracket the comparison rather than settle it.

On Qwen there is no stable ordering. Even the oracle (17/30) scores below both 7,000-token baselines, so the generator rather than retrieval is the binding constraint, and dense retrieval improves on two tasks (P2, P3) under the tighter budget.

Anchor extraction. Across all 60 classifier calls, the task type was correct every time, and 8 of the 9 distinct anchors resolved to exactly one graph node; the miss is the hook P3 asks the model to create, which cannot yet be in the graph. No name ambiguity occurred. The classifier is not deterministic at temperature 0 (3 of 12 generator-task cells vary), and Qwen invented a non-existent route on two P2 candidates. Every call still produced at least one resolving anchor, so the unresolved-anchor path of Section 3.2 did not arise in these runs.

5.4. Cost and Break-Even

Building a graph with Claude Sonnet 4.6 cost $2.05–$17.44 per repository (6.6–52.4 minutes; about $0.11–$0.15 per source file) and $36.87 for all five, from logged tokens. Per task, the KG is the cheapest condition because its prompts are smaller: $0.0199 on Claude against $0.0582 for BM25 and $0.0617 for dense, and $0.0023 against $0.0058 and $0.0059 on Qwen, with the classifier call included. Amortising the build against that saving, the graph pays for itself after 158–191 tasks on Claude for the two repositories with tasks, and after more than 1,500 on Qwen. Break-even grows with repository size, because extraction cost scales with file count while baseline contexts are capped. For a one-time task, building a KG for generation alone is therefore not cost-effective. The figures are upper bounds: extraction used no prompt caching although 98% of its tokens are repeated few-shot input, and the Qwen costs omit cache discounts applied by its providers.

Beyond generation. The break-even above counts a single consumer of the graph. The same one-time extraction also yields a typed, queryable index of the repository: which components consume a context, which route renders a screen, or which callers a prop-interface change affects are each a single Cypher query (our retrieval templates run in 2–7 ms) rather than a repository-wide search. Such an index can support onboarding, repository question answering [14], dependency tracing, debugging and refactoring impact analysis, uses that CodexGraph demonstrates with repository-chat, debugging, test-generation and documentation agents over its code graph [6]. These benefits were not measured,

and they assume the graph is kept current (re-extracting a change to k files costs about $k\times$ \$0.11–\$0.15); they strengthen the case for reuse but do not change the conclusion for a one-time task.

6. Limitations

The generation study is a pilot: six tasks on two repositories (the other three have no task harness), five near-identical candidates per cell at temperature 0, and no pass@k or significance testing, so every ordering above is an observation rather than an estimate. The task set saturates for the stronger model at 7,000 tokens, and the repositories with tasks are small (55–60 files), so whether structure pays more on larger repositories is untested. Five of the six designed conditions were run. The dense baseline uses the same OpenAI embedder for both generators, and no query-rewriting baseline was run, although the KG spends one LLM call on localisation. Because equal tokens are not equal information, the budget-matched view bounds rather than settles the comparison.

Extraction accuracy rests on 28 files, pre-filled drafts and effectively one annotator. Gap G1 (Redux state and actions) is known, and an extension for it is under way. Library provenance (DEPENDS_ON, PROVIDED_BY) is extracted but not used by retrieval, so no generation result is credited to it. No non-React framework was extracted: adaptability by exchanging schema and few-shot examples is a design property, not an evaluated result.

7. Conclusion and Future Work

This paper paired LLM4KG with KG4LLM through three contributions: (1) an LLM-assisted, few-shot method that extracts a knowledge graph directly from source code, without framework-specific parser engineering, reaching micro-averaged F1 of 0.95 on five repositories; (2) a framework-specific schema for React.js with 11 node types and 16 relationship types, whose coverage of real code changes was measured; and (3) KG-grounded repository-level code generation, evaluated against no-context, oracle whole-file, BM25 and dense-retrieval baselines. The evaluation supports three claims. Context is necessary, and at an equal budget on the stronger model its quality decides the outcome. KG context is competitive at 8–9$\times$ fewer tokens and is the cheapest condition per task, but repays its construction only through reuse. Both findings depend on the generator. A larger controlled study (30–40 tasks, $n = 50$, pass@k and paired-bootstrap significance testing), Nomic Embed Code retrieval and the Redux schema extension are under way. Future work will test transfer to Angular and Rails, develop the graph as a reusable semantic index, and compare against fully agentic, CodexGraph-style [6] retrieval.

Disclosure of Interests

The authors have no competing interests to declare that are relevant to the content of this article.

Declaration on Generative AI

During the preparation of this work, the authors used Claude (https://claude.ai) for language editing and grammar correction. The authors reviewed and revised the resulting content and took full responsibility for the final content of the paper.

References

[1] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. 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).

[2] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, C. Sutton, Program synthesis with large language models, arXiv preprint arXiv:2108.07732 (2021).

[3] Y. Ding, Z. Wang, W. U. Ahmad, H. Ding, M. Tan, N. Jain, M. K. Ramanathan, R. Nallapati, P. Bhatia, D. Roth, B. Xiang, CrossCodeEval: A diverse and multilingual benchmark for cross-file code completion, in: Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track, 2023. URL: https://openreview.net/forum?id=wgDcbBMSfh.

[4] S. Ouyang, W. Yu, K. Ma, Z. Xiao, Z. Zhang, M. Jia, J. Han, H. Zhang, D. Yu, RepoGraph: Enhancing AI software engineering with repository-level code graph, in: The Thirteenth International Conference on Learning Representations, 2025. URL: https://openreview.net/forum?id=dw9VUsSHGB.

[5] W. Liu, A. Yu, D. Zan, B. Shen, W. Zhang, H. Zhao, Z. Jin, Q. Wang, GraphCoder: Enhancing repository-level code completion via coarse-to-fine retrieval based on code context graph, in: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, 2024, pp. 570–581.

[6] X. Liu, B. Lan, Z. Hu, Y. Liu, Z. Zhang, F. Wang, M. Q. Shieh, W. Zhou, CodexGraph: Bridging large language models and code repositories via code graph databases, in: Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), Association for Computational Linguistics, Albuquerque, New Mexico, 2025, pp. 142–160. URL: https://aclanthology.org/2025.naacl-long.7/.

[7] B. Yang, H. Tian, J. Ren, S. Jin, Y. Liu, F. Liu, B. Le, Enhancing repository-level software repair via repository-aware knowledge graphs, arXiv preprint arXiv:2503.21710 (2025).

[8] M. Athale, V. Vaddina, Knowledge graph based repository-level code generation, in: 2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code), IEEE, 2025, pp. 169–176. doi:10.1109/LLM4Code66737.2025.00026.

[9] I. Saberi, F. Fard, Context-augmented code generation using programming knowledge graphs, arXiv preprint arXiv:2410.18251 (2024).

[10] J. Li, G. Li, X. Zhang, Y. Dong, Z. Jin, EvoCodeBench: An evolving code generation benchmark aligned with real-world code repositories, arXiv preprint arXiv:2404.00599 (2024).

[11] I. Abdelaziz, J. Dolby, J. P. McCusker, K. Srinivas, A toolkit for generating code knowledge graphs, in: Proceedings of the 11th Knowledge Capture Conference, Association for Computing Machinery, 2021, pp. 137–144. doi:10.1145/3460210.3493578.

[12] FalkorDB Team, FalkorDB: A high-performance graph database, https://github.com/FalkorDB/falkordb, 2023. Accessed: 2026-08-09.

[13] L.-P. Meyer, C. Stadler, J. Frey, N. Radtke, K. Junghanns, R. Meissner, G. Dziwis, K. Bulert, M. Martin, LLM-assisted knowledge graph engineering: Experiments with ChatGPT, in: First Working Conference on Artificial Intelligence Development for a Resilient and Sustainable Tomorrow (AITomorrow) 2023, Springer Fachmedien Wiesbaden, Wiesbaden, 2023, pp. 103–115. doi:10.1007/978-3-658-43705-3_8.

[14] S. Abedu, S. Khatoonabadi, E. Shihab, Synergizing LLMs and knowledge graphs: A novel approach to software repository-related question answering, ACM Transactions on Software Engineering and Methodology (2026). doi:10.1145/3796510, just Accepted.

[15] S. Robertson, H. Zaragoza, The probabilistic relevance framework: BM25 and beyond, Foundations and Trends in Information Retrieval 3 (2009) 333–389. doi:10.1561/1500000019.

[16] OpenAI, New embedding models and API updates, https://openai.com/index/new-embedding-models-and-api-updates/, 2024. Accessed: 2026-09-29.

[17] J. Cohen, A coefficient of agreement for nominal scales, Educational and Psychological Measurement 20 (1960) 37–46. doi:10.1177/001316446002000104.