

Fine-Tuning or RAG? An Empirical Evaluation of Text-to-Cypher Translation for Code Graph Analysis

Tim But^{†§}, Nan Yang^{*†}, Joseph Reynolds^{*}, Laurens Prast^{*}, Mauricio Verano Merino^{††}, Rosilde Corvino^{*}

^{*}*TNO-ESI, Eindhoven, The Netherlands*

[†]*Eindhoven University of Technology, Eindhoven, The Netherlands*

[‡]*Vrije Universiteit Amsterdam, Amsterdam, The Netherlands*

[§]*University College London, London, United Kingdom*

Abstract—Large language models (LLMs) have become central to code generation tasks; however, effectively adapting them to specialized and low-data domains remains challenging. Although prior work has explored Text-to-Cypher generation, existing datasets largely target domains outside software engineering. Furthermore, the high computational cost of fine-tuning these models motivates the investigation of alternative adaptation strategies, such as Retrieval-Augmented Generation (RAG).

This paper presents an empirical evaluation of adaptation strategies for converting software developers’ questions about their codebase into Cypher queries for code graph. First, we explore how fine-tuned models with a cross-domain dataset by Neo4j could perform code graph analysis tasks. We then evaluate pre-trained models, their fine-tuned variants, and RAG-based architectures using a custom code graph dataset. To support this study, we expanded our prior benchmark dataset *CodeGraphQA* from 420 to 790 annotated questions by collecting questions from professional developers, evaluating performance based on executability, schema adherence, and semantic accuracy.

Our contributions are three-fold. First, we extend previous benchmark to examine LLMs’ ability in Text-to-Cypher translation for code graph analysis. Using this benchmark, we demonstrate that fine-tuning on a small and software engineering-specific code graph dataset is more effective than fine-tuning on a much larger cross-domain dataset for code graph analysis tasks. Furthermore, we show that when code graph data is scarce, RAG consistently outperforms fine-tuning in terms of overall performance. This work provides practical suggestions to practitioners and researchers who develop and deploy LLM-based solutions for code graph analysis in practice.

Index Terms—code graph, Text-to-Cypher, LLMs

I. INTRODUCTION

Large Language Models (LLMs) have become a key enabler of AI-assisted software engineering, supporting diverse tasks such as code completion, generation, summarization, test generation, refactoring, and requirements engineering [32]. Many commercial tools, including IDEs (e.g., JetBrains IDEs [16]) and coding assistants (e.g., GitHub Copilot [12], Tabnine [26], Cursor [5], Claude, and Antigravity), offer LLM-powered chatbot assistants that support developers in a wide range of programming tasks. Most of these assistants use a Retrieval-Augmented Generation (RAG) architecture [11], where source code snippets are indexed within a vector database. For example, when a developer asks a question through the chat interface, the assistant retrieves relevant code fragments from different sources such as the user selected code snippet, current file, recently modified components, or results obtained via a

semantic similarity search. These fragments provide context for the LLMs to answer the user’s question. However, this mechanism often lacks a holistic understanding of the system’s architecture and dependencies; the retrieved snippets used for inference capture only the local context. Consequently, the assistant may struggle with questions that require system-wide reasoning, leading to incomplete or imprecise answers. Examples of such questions include identifying transitive dependencies across modules and understanding how changes in one component propagate through the system.

In prior work [30], we proposed a *GraphRAG* approach, which uses static analysis to build a code graph capturing relationships between entities, stored in Neo4j and queried via Cypher [6]. An LLM translates natural language into Cypher queries to retrieve dependency information, enabling more accurate and context-aware responses than relying on isolated code snippets. To evaluate this approach, we conducted a benchmark study with 420 question–query pairs collected from software engineers, along with a user study involving seven software engineers at Philips Healthcare. The results from both studies demonstrate the preliminary feasibility of LLMs for Text-to-Cypher translation in addressing questions about codebase structure and dependencies. However, the results also reveal that LLM-generated queries may be incorrect, exhibiting issues such as Cypher syntax violations, schema mismatches with the underlying graph, and semantic errors that misrepresent the user’s intent. The collected Text-to-Cypher dataset [30] enables two new strategies to improve the performance of LLMs: i) *fine-tuning*: fine-tuning LLMs to improve translation accuracy, ii) *few-shot prompting*: Integration of the dataset into a RAG framework where relevant examples are retrieved in response to user questions.

In this study, we investigate these two approaches to improving the accuracy of LLM-based Cypher query generation for code graph analysis. Our results provide actionable insights for both researchers and practitioners working with LLMs and graph-based code analysis in low-data environments, supported by findings from an industry-derived benchmark. Specifically, the contributions of this paper can be summarized as follows:

- **Extended benchmark evaluation of LLMs for code graph analysis tasks.** Building on our previous study, which established a baseline of 420 question–query pairs

collected from Philips Healthcare software engineers and evaluated pre-trained models such as GPT-4o, GPT-3.5 - Turbo, and Llama3-70B, we extended the dataset to 790 pairs by collecting more questions from these engineers. Moreover, we employ additional evaluation metrics, which leverage a Cypher parser to detect syntactic and schema-related errors and employ LLMs as judges to assess semantic adherence. We performed further experiments with the GPT-4.1 series and Gemma series. Among these models, gpt-4.1 achieved the highest semantic accuracy. As a result, this expanded evaluation provides a *more comprehensive benchmark* for assessing LLM performance in code graph analysis tasks, which is used in subsequent experiments. The evaluation dataset and script are publicly available ¹.

- **Assess the impact of fine-tuning on code graph analysis tasks.** We examine LoRA (Low-Rank Adaptation) [14], a Parameter-Efficient Fine-Tuning (PEFT) technique, to address the challenges of Cypher code generation. Specifically, we compare the effectiveness of fine-tuned Gemma models on the Neo4j cross-domain Text-to-Cypher dataset [23] with model variants fine-tuned on our domain-specific code graph dataset. Our results show that fine-tuning on the small code graph dataset achieves better performance despite being over 500 times smaller than the cross-domain dataset, highlighting the critical importance of domain relevance in fine-tuning for code graph analysis. The fine-tuned models with the code graph dataset is publicly available ^{2 3}.
- **Assess the impact of a RAG-based solution on code graph analysis tasks.** We conduct an empirical evaluation of a RAG-based approach compared to fine-tuning strategies. Our results show that RAG significantly outperforms fine-tuning across both GPT-4.1 and Gemma series, indicating that retrieval can more effectively compensate for limited domain-specific training data. Notably, gpt-4.1 combined with RAG achieves a sufficiency rate of 58%. This finding suggests practical implications for industrial software engineering contexts, where high-quality and task-specific data is often scarce and costly to curate. In such settings, RAG can provide a scalable and cost-effective alternative, enabling improved model performance without requiring large datasets or extensive fine-tuning.

II. RELATED WORK AND BACKGROUND

A. Text-to-Cypher Generation

Efforts have been made to address the challenge of Text-to-Cypher generation through the open-sourcing of datasets and LLM-based solutions [19, 13, 27]. In particular, Neo4j has made substantial contributions by curating

open-source datasets for benchmarking large language models on Text-to-Cypher tasks [23]. Their benchmark consolidates 16 publicly available datasets into a unified collection comprising approximately 44K instances, which they use to fine-tune gemma-2-9b-it, gemma-3-4b-it, and gemma-3-27b-it for Text-to-Cypher translation. Evaluation results show that these fine-tuned Gemma models outperform their pre-trained counterparts. However, the available datasets are largely designed for other applications with simpler schemas that differ from ours, such as the entertainment industry, rather than code graphs.

Our work offers several distinct contributions. First, we explore Text-to-Cypher generation for industrial code graphs, shedding light on its applicability to real-world software engineering scenarios. Secondly, we assess the impact of fine-tuning with Neo4j’s cross-domain dataset on model performance in code analysis tasks. Finally, we analyze the benefits of fine-tuning with a small code graph dataset and compare this approach with RAG methods to understand their relative effectiveness.

B. Parameter-Efficient Fine-Tuning

Parameter-Efficient Fine-Tuning (PEFT) tailors pre-trained language models to new tasks by updating only a small number of task-specific parameters while leaving the backbone model unchanged, which greatly lowers computational and memory costs. Low-Rank Adaptation (LoRA) [14] is a popular PEFT approach that inserts low-rank trainable adapters into selected layers (e.g., attention projections) to efficiently model task-specific changes. However, although LoRA freezes the backbone weights, they must still be stored and used in full precision during training, making memory consumption a bottleneck for large models. Quantized Low-Rank Adaptation (QLoRA) [9] addresses this limitation by quantizing the frozen backbone weights to low-bit precision while keeping the LoRA adapters in higher precision, enabling memory-efficient fine-tuning under strict GPU constraints with performance comparable to standard LoRA.

Both LoRA and QLoRA have been widely adopted for fine-tuning large language models in software engineering tasks. Weyssow et al. [29] applied PEFT to code generation and demonstrated that both LoRA and QLoRA are effective across a diverse set of models. Afrin et al. [1] employed QLoRA for code summarization, showing that it can surpass full model fine-tuning. Macháček [18] investigated both PEFT and full fine-tuning approaches for program repair and concluded that full fine-tuning often degrades benchmark performance due to distributional mismatch and overfitting. In contrast, PEFT methods consistently yield improved results across models. Researchers at Neo4j [23] employed LoRA to enhance the fine-tuning of large language models for Text-to-Cypher generation using a large cross-domain dataset, demonstrating its feasibility in the task outside the software engineering domain. One of the contributions of this study is to assess how well these fine-tuned models perform for code graph.

¹<https://github.com/NotANumberYang/text-to-cypher-evaluation-code-graph>

²<https://huggingface.co/LLM4Legacy-Project/dsv4-gemma-3-4b-it-finetuned-cypher>

³<https://huggingface.co/LLM4Legacy-Project/dsv4-gemma-2-9b-it-finetuned-cypher>

C. Retrieval-Augmented Generation

When a domain-specific dataset is available, fine-tuning and RAG [17] are two widely used approaches for improving pre-trained large language models. Fine-tuning adapts model parameters to a specific task or domain, allowing the model to internalize domain-specific patterns and behaviors. In contrast, RAG enhances model outputs by retrieving relevant external information at inference time, enabling access to up-to-date or domain-specific knowledge without modifying model weights. These approaches involve different trade-offs. Fine-tuning is often used to obtain strong task-specific performance but incurs additional computational cost and risks overfitting, particularly when training data is limited, whereas RAG avoids parameter updates and supports dynamic knowledge updates at the expense of increased inference-time latency.

These two approaches are frequently compared in the literature and the results vary from case to case. Weyssow et al. [29] evaluated fine-tuning and retrieval-augmented generation for code generation using a dataset of 8,827 training examples and 1,921 test inputs, finding that parameter-efficient fine-tuning outperforms RAG in this setting. In contrast, Wang et al. [28] compared fine-tuning and RAG for code completion using 120K training examples and 20K test inputs, and showed that RAG achieves higher accuracy than fine-tuning alone. Additional evidence of this variability is provided by Bassamzadeh et al. [3], who studied code generation for a domain-specific language, showing that fine-tuning achieved the highest accuracy, while an optimized RAG approach attained comparable results and better compilation rates, but with slightly higher hallucination rates. Consistent with these findings, Soudani et al. [25] reported that RAG is more effective for question answering tasks when the required knowledge is less popular or underrepresented in pre-training data. Taken together, these findings indicate that the relative effectiveness of fine-tuning and RAG is not consistent across tasks, evaluation metrics, domains and data sizes, underscoring the importance of evaluating both approaches across diverse application settings.

III. STUDY DESIGN

This section presents the experimental setup, including datasets, models used for evaluation, and metrics. The goal of this study is to provide both conceptual insight and practical recommendations for researchers and practitioners working on code graph analysis using LLMs, with particular emphasis on low-data contexts. To achieve this goal, we answer the following research questions:

- **RQ1: To what extent can pre-trained LLMs accurately (syntactically and semantically) translate dependency questions into Cypher queries for retrieving dependency information from code graphs?**

RQ1 establishes a baseline for measuring the performance of LLMs in translating dependency questions into Cypher queries. Building on our previous work [30], we expand our dataset from 420 data points to 790 data points. We evaluate executability, schema adherence and semantic

accuracy of the generated queries. This benchmark provides a more comprehensive comparison across model types and sizes including pre-trained GPT-4.1 series and Gemma series.

- **RQ2: How do existing fine-tuned models generalize to unseen and code graph schemas?**

We examine the fine-tuned Text-to-Cypher models provided by Neo4j. The model `gemma-2-9b-it` was fine-tuned using an open-source dataset containing approximately 44.4K question–query pairs. Neo4j later curated and improved this dataset, resulting in a revised version with 40.4K pairs. This updated dataset was subsequently used to fine-tune two additional models: `gemma-3-4b-it` and `gemma-3-27b-it`. We conduct experiments with these models using our dataset to evaluate whether their fine-tuned Text-to-Cypher capabilities generalize effectively to our application.

- **RQ3: To what extent does fine-tuning LLMs on a code graph Text-to-Cypher dataset improve their performance on the generation task?**

In practice, application-specific datasets are often small, which raises questions about the effectiveness of fine-tuning under such constraints. This study investigates the impact of using a domain-specific Text-to-Cypher dataset compared to a large, cross-domain dataset. The ultimate goal is to provide actionable insights and recommendations for practitioners on when and how fine-tuning is beneficial for code graph analysis under low-data availability constraints.

- **RQ4: How does the performance of RAG compare to fine-tuning approaches on a small code graph dataset?**

Fine-tuning adapts a model specifically for Text-to-Cypher in the context of code graph analysis. Meanwhile, RAG retrieves relevant question–query pairs to use as few-shot examples, guiding generation without the overhead of retraining. This RQ offers practical recommendations to practitioners; we empirically evaluate and compare both approaches’ performance when training and evaluation data are scarce.

A. Dataset

1) *Obfuscated Code Graph*: Following [30], the dataset consists of a graph representation of the `jsoncpp` C++ project, generated using the static analysis tool *Renaissance*, a purpose-built framework for industry-scale code analysis and refactoring previously applied to extract structured code graphs from C++ codebase in industry [8, 7]. To avoid possible bias due to inclusion of the `jsoncpp` codebase in the LLM’s training data, the entity names in the resulting graph were then replaced with obfuscated versions. The obfuscation process replaces concrete code artifacts (e.g., file, class, and function names) with generic placeholders (e.g., `writer.h` is replaced with `headerfile_1.h`), yielding an anonymized and structurally consistent code graph. The resulting code graph has 2751 nodes, 7722 relationships, 11 distinct node types, 18 relationship types, and an average of 3.64 properties per node.

2) *Original Dataset*: The original dataset ($\mathcal{D}_a$) consists of 84 question-query pairs collected from software engineers at Philips Healthcare in our prior study. Please refer to our prior work [30] for a detailed description of the dataset construction process. These queries were originally formulated to analyze dependencies in their proprietary codebase. To preserve confidentiality while enabling reproducible evaluation, we do not use the original industrial code graph, but rather the obfuscated code graph. To align the industrial questions with this obfuscated graph, we normalize each question-query pair by replacing Philips-specific identifiers with valid identifiers drawn from the obfuscated graph. Following normalization, we expand each natural language question using Gemini 1.5 Pro to obtain four paraphrased variants, increasing linguistic diversity while preserving query intent. This results in a dataset of 420 question-query pairs, corresponding to 84 unique Cypher queries, each paired with its original phrasing and four generated variants. Finally, all generated data points were manually reviewed by two authors to ensure correctness and consistency.

3) *Extended Dataset*: To extend the existing dataset ($\mathcal{D}_a$), we incorporate 74 additional question-query pairs from our prior user study with seven software engineers at Philips Healthcare [30]. In that study, engineers interacted with a GraphRAG-based chatbot powered by GPT-4o to answer questions about software dependencies in their codebase. Similar to the original dataset, these newly added pairs capture a diverse set of real-world questions arising from daily development activities. We followed the same dataset construction methodology as in our previous work by normalizing the questions, employing the obfuscated code graph, and generating four additional question variants. This process results in a new dataset, $\mathcal{D}_b$, comprising 370 data points, corresponding to 74 unique Cypher queries, each paired with its original phrasing and four generated variants. Therefore, the final dataset $\mathcal{D}$ used in this paper is defined as:

$$\mathcal{D} := \mathcal{D}_a \cup \mathcal{D}_b, \quad |\mathcal{D}| = 790 \text{ data points}$$

4) *Difficulty-aware semantic splitting of $\mathcal{D}$* : For fine-tuning, the dataset is split into training, validation, and test sets. In a RAG setting, the training set is indexed in the retrieval database, while the test set is used to evaluate system performance. To support and compare both scenarios, the dataset is partitioned accordingly to ensure that queries of varying difficulty are balanced across splits. Moreover, semantically similar queries must not be distributed across different sets, as this can lead to overfitting and artificially inflated performance. Random splits are known to be problematic in Text-to-Query settings because paraphrases of the same task may appear in different splits. For example, “what header files include header file $\mathbf{x}$ ” and “list all of the header files that includes header file $\mathbf{x}$ ” encode identical Cypher queries. On the other hand, “what header files include header file $\mathbf{x}$ ” and “what header files include header file $\mathbf{y}$ ” fundamentally encode the same query template, only changing one variable. Such cross-contamination can lead to over-performance on test metrics, resulting in biased readings of a model’s performance. This

issue has been extensively documented in the Text-to-SQL literature and has motivated query-level split protocols [10, 31]. To mitigate this, we apply a two-step splitting procedure that (i) groups semantically similar questions into clusters (*Semantic clustering*), and (ii) balances a difficulty metric across splits (*Difficulty scoring and stratified splitting*).

(i) Semantic clustering

For each natural language question q_i , we compute a sentence embedding $e_i = f(q_i) \in \mathbb{R}^{384}$ using the all-MiniLM-L6-v2 model, a 6-layer MiniLM encoder that maps sentences to 384-dimensional vectors.⁴ We compute pairwise cosine distances between embeddings and apply agglomerative clustering with average linkage and a fixed distance threshold to group semantically similar questions into clusters, typically capturing paraphrases of the same intent. Since questions with substantially different wording may still be associated with the same underlying Cypher query, we further merge clusters that share the same base query identifier. Each resulting cluster is treated as an indivisible unit during the dataset splitting.

(ii) Difficulty scoring and stratified splitting

To balance query complexity across splits, we assign each question a difficulty score based on the number of Cypher-specific clauses (e.g., MATCH, WHERE, WITH) in its associated query, following Neo4j’s practice [24]. For clusters containing multiple question-query pairs, we compute a cluster-level difficulty score as the *mean* of the query-level difficulty scores within the cluster. Clusters are then partitioned into five quantile-based difficulty bins. Training, validation, and test splits are drawn proportionally from each bin to ensure a balanced difficulty distribution across splits. Using this procedure, we obtain 510 question-query pairs for training, 105 for validation, and 175 for testing.

B. Models

Currently, many organizations leverage Infrastructure as a Service (IaaS) platforms to deploy proprietary LLMs, such as those from OpenAI, within controlled environments. However, proprietary solutions are not always viable. Some domains have strict legal regulations or financial constraints that mandate the use of open-weights alternatives. Since we are interested in studying both deployment paradigms, we evaluate both approaches as shown in Table I.

1) *Proprietary models*: We run our experiments using three general-purpose models i) gpt-4.1, ii) gpt-4.1-mini, and iii) gpt-4.1-nano. These models were chosen because they provide strong baselines for reasoning, code understanding, and query generation. Experimenting with the smaller variants (mini and nano) assesses scalability and performance degradation as model capacity decreases, offering insight into the trade-offs between performance and cost.

⁴<https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2>

2) *Open-weight Models*: For the open-weight models, we focus on the Gemma family because it allows us to directly compare our fine-tuning and RAG approaches with Neo4j’s fine-tuning results on the same model family. Neo4j has fine-tuned several Gemma variants using a large-scale cross-domain dataset [23], which contains Cypher queries from various domains, such as medical, film, and sports domains. Specifically, gemma-2-9b-it-ft-neo4j⁵ was fine-tuned on the text2cypher-2024v1 dataset⁶, consisting of 44.4k question–query pairs. Additionally, gemma-3-27b-it-neo4j and gemma-3-4b-it-ft-neo4j were fine-tuned in the updated text2cypher-2025v1 dataset⁷ that contains 40.4k question–query pairs. We evaluate these fine-tuned models on code graph analysis. This setup enables a two-fold comparison: assessing fine-tuning against RAG on a small, domain-specific dataset, and evaluating both against Neo4j’s large-scale fine-tuning approach. Using our code graph dataset $\mathcal{D}$, we fine-tune two Gemma models (gemma-2-9B and gemma-3-4B). Due to computational resource constraints, larger variants such as gemma-3-27B were excluded from our experiments.

C. Evaluation criteria

We evaluate queries generated by LLMs using three key dimensions: *executability*, *schema adherence*, and *accuracy*.

a) *Executability*: Executability determines whether a query can be successfully executed on the Neo4j engine. We run queries on the Neo4j engine and classify results into categories. Successful queries return the subgraph without errors. A *Syntax Error* occurs when the response indicates an invalid query. We process 790 queries using a script that enforces a 10-second timeout per query to ensure system responsiveness. The majority of valid queries complete within this bound, while those exceeding it generally reflect greater complexity. A *Timeout Error* is reported when execution surpasses the limit. Neo4j enforces a transaction memory cap to prevent large transactions from consuming excessive resources. Queries exceeding this threshold often involve graphs too large for practical interpretation. In our experiment, we used Neo4j 5.18⁸ with the default 512 MB limit. A *Size Limit Error* occurs when memory usage exceeds this cap.

b) *Schema adherence*: In our previous work [30], we observed that an executable query may still violate the schema. Therefore, we extend our evaluation with a schema adherence check. Schema adherence ensures that the query conforms to the defined graph schema, preventing structural inconsistencies. Schema violations are identified using a script based on a Cypher parser (i.e., the OpenCypher grammar [22]), which analyzes and traverses the generated syntax trees. Consider the following query:

```
MATCH (fd:Folder {name: "Folder_255"})-[:  
ParentFolder]->(hf:HeaderFile)  
RETURN fd, hf
```

This query attempts to match a node of type `Folder` with a specific name, connected via a `ParentFolder` relationship to a node of type `HeaderFile`. We use this query as a reference to illustrate common schema violations detected by our parser-based script:

- **NodeTypeDoesNotExist**: This violation occurs when a node type in the query does not exist in the schema. In the example above, the parser checks whether `Folder` and `HeaderFile` are valid node types. If, for instance, `HeaderFile` is not defined in the schema, the query contains this violation.
- **RelationshipTypeDoesNotExist**: Occurs when the relationship type is not defined in the schema. In the example, the parser verifies that `ParentFolder` exists.
- **InvalidRelationship**: Occurs when the relationship type is not valid between the specified node types. For instance, if `ParentFolder` is not allowed between `Folder` and `HeaderFile`, this violation is triggered.
- **WrongArrowDirection**: Occurs when the relationship direction does not match the schema. If the schema requires `Folder` to point to `HeaderFile`, reversing the arrow would cause this violation.

c) *Semantic accuracy*: This metric evaluates how closely the execution results of a generated query align with the ground truth, thereby capturing its semantic correctness and overall quality. Because semantically equivalent Cypher queries may differ substantially in syntactic structure, direct query comparison is unreliable; we therefore evaluate queries based on the correctness and sufficiency of their execution results. To assess the accuracy of the retrieved dependency information, we adopt an LLM-as-a-judge approach. Specifically, for each natural language question, the corresponding generated query, and its execution results, an LLM determines whether the ground-truth answer (manually constructed by the authors) can be derived from the provided information. The LLM is instructed to classify the execution results as sufficient if the information fully supports the answer, partially sufficient if it is incomplete but still informative, or insufficient otherwise, and to provide a brief rationale explaining its classification. This evaluation strategy is inspired by execution-aware Text-to-SQL evaluation practices, including IBM’s Text-to-SQL Evaluation Toolkit, which combine query execution with LLM-based semantic judgment to assess answer [15]. In particular, We use GPT-5 as the LLM judge, as it is a more advanced model than those evaluated in this study and offers stronger reasoning capabilities, making it suitable for assessing semantic correctness. To validate the reliability of LLM-based judgments, we manually examined the evaluation results for 40 randomly selected questions. We observed a high level of agreement between the LLM judgments and manual assessments, supporting the validity of using LLMs as judges in our evaluation framework.

⁵<https://huggingface.co/neo4j/text2cypher-gemma-2-9b-it-finetuned-2024v1>

⁶<https://huggingface.co/datasets/neo4j/text2cypher-2024v1>

⁷<https://huggingface.co/datasets/neo4j/text2cypher-2025v1>

⁸<https://neo4j.com/developer/kb/neo4j-supported-versions/>

TABLE I
EVALUATED LLMs

Pre-trained LLMs	
gpt-4.1	gemma-2-9b-it
gpt-4.1-mini	gemma-3-4b-it
gpt-4.1-nano	gemma-3-27b-it
Cross-domain fine-tuned LLMs (dataset <i>text2Cypher-2024/2025</i>)	
gemma-2-9b-it-ft-neo4j	gemma-3-4b-it-ft-neo4j
gemma-3-27b-it-ft-neo4j	
Code-graph fine-tuned LLMs (dataset $\mathcal{D}$)	
gpt-4.1-ft-cg	gpt-4.1-mini-ft-cg
gpt-4.1-nano-ft-cg	gemma-3-4b-it-ft-cg
gemma-2-9b-it-ft-cg	
Pre-trained LLMs with RAG (dataset $\mathcal{D}$)	
gpt-4.1-rag	gemma-2-9b-it-rag
gpt-4.1-mini-rag	gemma-3-4b-it-rag
gpt-4.1-nano-rag	

D. Experiments: Baseline, Fine-tuning, and RAG

To evaluate the effectiveness of different approaches for Text-to-Cypher translation, we designed a series of experiments to compare PEFT and RAG. Table I summarizes the LLM configurations evaluated in this study, along with the naming conventions used throughout the rest of the paper.

For PEFT, from the complete dataset $\mathcal{D}$, we used 510 question–query pairs for training, 105 for validation, and 175 for testing. In contrast, the RAG approach utilized only the training and test sets because it does not require a held-out validation set; retrieval operates directly over the indexed training examples, and no model parameters are updated during inference. The baseline experiment does not contain a fine-tuning step so that we can compare this baseline with PEFT and RAG approaches to determine the effect of fine-tuning the models.

1) *Baseline*: Our baseline evaluates the model’s ability to understand both natural-language questions and Cypher without any additional training. Given a natural-language question, we prompt the pretrained model to generate the corresponding Cypher query that retrieves the required information from the code graph. In this baseline setting, the prompt consists of four elements: (i) A task instruction describing the Text-to-Cypher translation objective, (ii) the schema of the code graph, (iii) few-shot examples, and (iv) the user’s question. The few-shot examples are constructed manually to demonstrate common code entity relationships.

2) *PEFT with QLoRA*: To assess the impact of task-specific adaptation, we employed QLoRA for PEFT of selected open-weight models. QLoRA enables the fine-tuning of LLMs with reduced computational requirements while preserving model performance [9]. During fine-tuning, each training example is formatted as a prompt–completion pair to guide the model toward producing the desired output. The expected Cypher query serves as the completion, while the question forms the prompt. This structure conditions the model to map user questions directly to their corresponding Cypher representations.

To be memory efficient, the model was stored in a highly compressed 4-bit form. LoRA adapters were applied to all linear layers except the `lm_head` to maximize adaptation capacity while preserving the original output head. The fine-tuning procedure consisted of two phases: an automated hyperparameter search and a final training run with the best-performing configuration.

To efficiently explore the hyperparameter space, we used Optuna for Bayesian optimization over 10 trials per model. The trial budget was chosen given the low dimensionality of the search space (four dimensions: LoRa rank r , scaling factor α , dropout and learning rate) and the sample efficiency of TPE-based Bayesian optimizations, which according to [2] identify strong configurations within tens of trials.

To reduce computational cost, hyperparameter search was performed on a 25% subsample of the training data, with each trial running for a fixed 4 epochs. The LoRa rank was selected from $\{8, 16, 32\}$, scaling factor α from $\{32, 64\}$, and dropout from $[0.05, 0.10]$. The learning rate was searched in the range $[1 \times 10^{-5}, 5 \times 10^{-5}]$. Model performance was evaluated against the held-out validation set, and the configuration with the lowest validation loss was selected for final training. Hyperparameter search yielded the following optimal configurations: ($r = 32, \alpha = 64, \text{dropout} = 0.07$) for `gemma-3-4b-it-ft-cg`, and ($r = 32, \alpha = 64, \text{dropout} = 0.05$) for `gemma-2-9b-it-ft-cg`. The configuration was then used to fine-tune each model on the full training set. The final training was performed for 15 epochs with early checkpoint selection based on minimum validation loss. We use a per-device batch size of 1 with gradient accumulation over 8 steps, which results in a batch size of 8. We use paged AdamW with 8-bit optimizer states to reduce memory consumption during fine-tuning. The model updates were applied after each accumulation step. All experiments were conducted on an NVIDIA Tesla V100 GPU with 48GB VRAM.

3) *Fine-tuning with OpenAI*: To evaluate the effectiveness of supervised fine-tuning on proprietary models Section III-B1), we utilized OpenAI’s fine-tuning API, which allows for task-specific adaptation of large language models through supervised learning [20]. OpenAI has not publicly disclosed whether GPT-4.1 fine-tuning updates all model parameters or employs PEFT. Therefore, we treat the underlying optimization mechanism as proprietary and unspecified. In total, 510 question-query pairs were used for training, 105 for validation, and 175 for testing. We train GPT-4.1, GPT-4.1-mini and GPT-4.1-nano for three epochs, which we found sufficient for convergence given the small training set and batch size of one. Using a limited number of epochs helps reduce the risk of overfitting, particularly since OpenAI’s fine-tuning API does not support early stopping by default. The validation set was used to monitor for overfitting by computing validation loss at the end of each epoch. We used the default learning rate settings provided by OpenAI: 2 for GPT-4.1 and GPT-4.1-mini, and 0.1 for GPT-4.1-nano.

TABLE II
EXECUTABILITY BREAKDOWN FOR BASE AND FINE-TUNED (FT-NEO4J)
VARIANTS (FULL DATASET, N=790, %)

Model variant	Exec.	Syntax	Timeout	Size
gpt-4.1	93.8	4.9	1.3	0.0
gpt-4.1-mini	94.4	4.1	1.4	0.1
gpt-4.1-nano	85.9	12.8	0.8	0.5
gemma-3-27b-it	93.3	6.5	0.3	0.0
gemma-3-27b-it-ft-neo4j	92.8	6.7	0.5	0.0
gemma-2-9b-it	81.6	18.1	0.3	0.0
gemma-2-9b-it-ft-neo4j	71.0	26.8	1.8	0.4
gemma-3-4b-it	78.0	22.0	0.0	0.0
gemma-3-4b-it-ft-neo4j	78.7	15.7	4.2	1.4

4) *RAG*: We implement a RAG pipeline to enhance model performance by providing similar query examples retrieved for the training set. Specifically, we construct a vector index from the 510 training examples using ChromaDB [4]. Each example consists of a question–query pair. We use OpenAI’s text-embedding-3-large model [21] to generate question embeddings. For each test question, we retrieve the top-6 most similar training examples based on cosine similarity between the embedding of test questions and the embeddings of questions. The retrieved question–query pairs are appended to the original prompt used in the baseline setting.

IV. BASELINE AND NEO4J FINE-TUNING (RQ1 AND RQ2)

In this section, we show the baseline and Neo4j fine-tuning results by analyzing the performance of the evaluated GPT and Gemma models with respect to executability, schema adherence, and accuracy.

a) *Executability*: Table II presents the executability of queries generated by the variants. gpt-4.1 and gpt-4.1-mini perform almost identically, achieving 93.8% and 94.4% executability, respectively, with very low syntax error rates (4.9% and 4.1%) and negligible timeout errors. gpt-4.1-nano shows a slight drop to 85.9%, primarily due to a higher syntax error rate (12.8%). Among Gemma models, gemma-3-27b-it reaches 93.3% executability, comparable to GPT-4.1, but smaller variants degrade consistently: gemma-2-9b-it falls to 81.6%, and gemma-3-4b-it to 78.0%, with syntax errors rising to 18.1% and 22.0%, respectively. *Timeout* and *Size Limit* errors remain negligible across all base models, suggesting that syntax correctness is the dominant factor affecting executability.

Fine-tuned Gemma models by Neo4j exhibit mixed results. For gemma-2-9b-it-ft-neo4j, executability drops significantly, and syntax errors increase from 6.7% to 18.1%, suggesting that fine-tuning negatively impacts its syntax capability. In contrast, fine-tuning improves gemma-3-4b-it, reducing syntax errors from 22.0% to 16.7%. This discrepancy may stem from differences in training data: gemma-2-9b-it-ft-neo4j was fine-tuned on a 2024 dataset, whereas gemma-3-4b-it-ft-neo4j used a

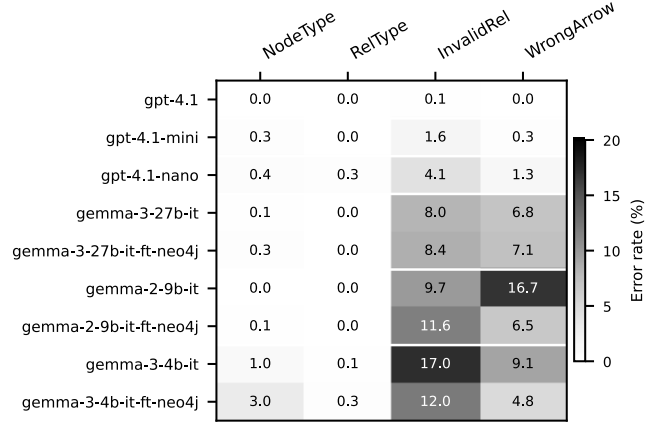


Fig. 1. Schema error rate for base and ft-neo4j variants across model families (full dataset, n = 790).

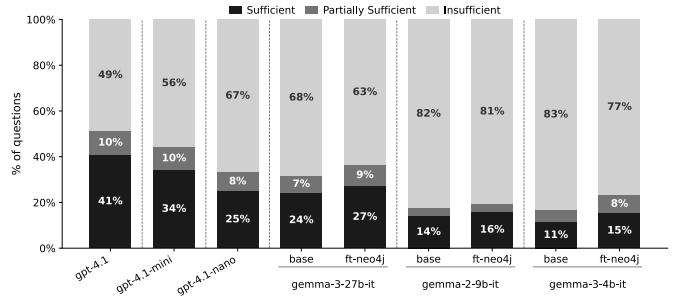


Fig. 2. Semantic accuracy of base and fine-tuned (ft-neo4j) variants across model families (full dataset, n = 790)

cleaned-up 2025 version. Furthermore, fine-tuning consistently increases *Timeout* and *Size Limit* errors, indicating that fine-tuned models tend to generate more complex queries that lead to timeout and memory issues.

b) *Schema adherence*: Figure 1 presents a heatmap of the schema errors presented in the generated queries. Overall, every model occasionally produces queries with schema errors. Violations such as *NodeTypeDoesNotExist* and *RelationshipTypeDoesNotExist* are relatively rare, whereas *InvalidRelationship* and *WrongArrowDirection* remain the most common issues. The gpt-4.1 family demonstrates the strongest schema adherence. Both gpt-4.1 and gpt-4.1-mini exhibit minimal errors (0.1% and 2.1%, respectively), while gpt-4.1-nano performs worse, primarily struggling with selecting the correct relationship type between node types.

Gemma models show significantly higher error rates, particularly for *InvalidRelationship* and *WrongArrowDirection*, reaching up to 16.7% and 17.0%. Specifically, gemma-2-9b-it performs poorly in maintaining correct arrow direction, while gemma-3-4b-it frequently fails to select appropriate relationship types. Fine-tuning by Neo4j yields mixed results: some improvements are observed in certain categories, but overall schema consistency remains a challenge for smaller Gemma variants.

c) *Semantic Accuracy*: Figure 2 shows the semantic accuracy of generated queries. GPT-4.1 models consistently demonstrate stronger semantic accuracy, with gpt-4.1 achieving the highest. gpt-4.1-mini and gpt-4.1-nano show a noticeable drop. Interestingly, gpt-4.1-nano performs on a par with gemma-3-27b-it, despite the significantly larger size of the latter, suggesting that architecture and training strategy may outweigh the sheer parameter count in determining semantic accuracy. Gemma models, on the contrary, exhibit a lower rate of sufficiency. Even the largest model, gemma-3-27b-it, achieves only a sufficiency rate of 24%, while smaller models such as gemma-2-9b-it and gemma-3-4b-it fall below 14% sufficiency. This observation indicates that Gemma models struggle to produce queries that retrieve the critical information, often generating outputs that diverge substantially from the ground truth.

We observe that fine-tuning Gemma models with a cross-domain dataset offers modest improvements in sufficiency rate. For example, gemma-3-27b-it-ft-c increases from 24% to 27%. However, most queries still fail to retrieve the correct nodes and edges.

RQ1 and RQ2 summary

gpt-4.1 stands out as the most reliable LLM from our test set for translating dependency questions into Cypher queries, given it has the highest semantic accuracy and minimal syntax and schema violations. The largest Gemma model in our experiment, gemma-3-27b-it, attains similar executability to gpt-4.1, but incurs 14.8% more schema violations and shows a 17% drop in semantic accuracy. Smaller Gemma variants deteriorate further on both metrics. Interestingly, fine-tuning Gemma models on cross-domain data yields only marginal improvements, suggesting that domain-specific adaption is required to obtain meaningful performance benefits.

V. FINE-TUNING AND RAG (RQ3 AND RQ4)

In this section, we present the results of fine-tuning experiments conducted on the studied base models, including GPT-4.1 and Gemma models. We also compare the performance of fine-tuned models with a RAG setup. Similarly to Section IV, we examine the executability, adherence to the schema, and semantic accuracy of the generated queries. Table III, Figure 3 and Figure 4 report the result.

A. Fine-tuning

Overall, fine-tuning with a code graph dataset consistently improves executability, schema adherence, and semantic accuracy for most models, although the magnitude of improvement varies across model families. gpt-4.1-ft-cg shows a small drop, suggesting that large models may require more extensive fine-tuning data to leverage domain-specific supervision. In contrast, gpt-4.1-mini and GPT-4.1-nano

TABLE III
EXECUTABILITY BREAKDOWN FOR BASE, FT-CG, AND RAG VARIANTS (TEST SPLIT, $n = 175$, %).

Model	Variant	Exec.	Syntax	Timeout	Size
gpt-4.1	base	93.1	6.3	0.6	0.0
	ft-cg	96.0	2.9	1.1	0.0
	rag	98.3	1.1	0.6	0.0
gpt-4.1-mini	base	93.7	2.9	2.9	0.6
	ft-cg	98.9	0.6	0.0	0.6
	rag	98.3	1.7	0.0	0.0
gpt-4.1-nano	base	86.3	13.1	0.6	0.0
	ft-cg	86.9	12.0	1.1	0.0
	rag	92.6	7.4	0.0	0.0
gemma-2-9b-it	base	74.9	24.6	0.6	0.0
	ft-neo4j	68.6	31.4	0.0	0.0
	ft-cg	90.3	9.7	0.0	0.0
	rag	84.0	16.0	0.0	0.0
gemma-3-4b-it	base	74.9	25.1	0.0	0.0
	ft-neo4j	76.6	17.7	5.7	0.0
	ft-cg	87.4	12.6	0.0	0.0
	rag	88.6	11.4	0.0	0.0

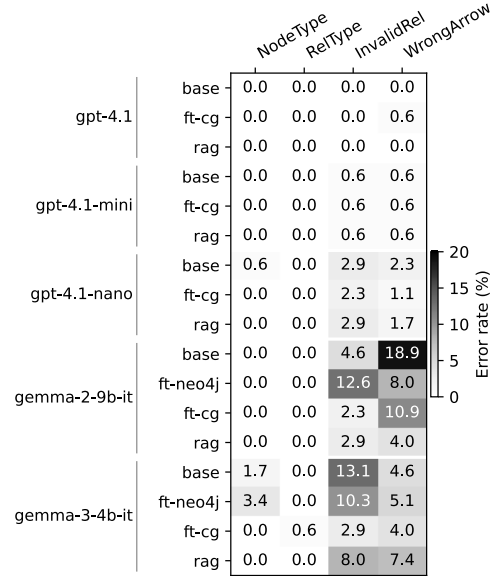


Fig. 3. Schema error rate of base, fine-tuned (ft-cg), and RAG variants across model families (test split, $n = 175$).

benefit greatly from fine-tuning; gpt-4.1-mini-ft-cg achieves near-perfect (98.9%) executability and schema adherence with only 1.2% queries with schema violations. It outperforms gpt-4.1 with a small improvement in semantic accuracy, with a higher partial sufficiency rate. gpt-4.1-nano is also improved through fine-tuning, especially in terms of semantic accuracy, though its accuracy remains lower relative to larger GPT variants.

As discussed in Section IV, fine-tuning Gemma models on the cross-domain dataset by Neo4j fails in achieving better executability and schema adherence. In contrast, Gemma models exhibit substantial gains after fine-tuning

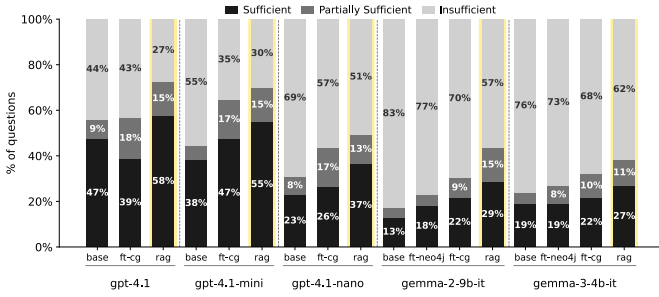


Fig. 4. Semantic accuracy of base, fine-tuned (ft-cg), and RAG variants across model families (test split, $n = 175$). Yellow shading marks the best-performing variant per family.

with a code graph dataset, with both `gemma-2-9b-it` and `gemma-3-4b-it` showing large increases in executability and schema adherence. When it comes to schema adherence, Gemma models show substantially higher error rates, concentrated in the *InvalidRelationship* and *WrongArrowDirection* columns. The base `gemma-2-9b-it` variant produces 18.9% wrong-arrow errors, the highest value across models. The cross-domain Neo4j fine-tune (ft-neo4j) does not consistently improve schema adherence and can even increase errors. For instance, `gemma-2-9b-it-ft-neo4j` increases the error rate of *InvalidRelationship* from 4.6% to 12.6%. In contrast, fine-tuning with the code graph dataset (ft-cg) yields the largest reductions for both Gemma families, reducing *InvalidRelationship* from 13.1% to 2.9% and *WrongArrowDirection* from 4.6% to 4.0% for `gemma-3-4b-it`. Improvements in semantic accuracy are also observed, although the magnitude varies: `gemma-2-9b-it-ft-cg` achieves approximately a 9% increase in sufficiency rates, whereas `gemma-3-4b-it-ft-cg` shows a more modest improvement of about 3%. The improvements surpass those obtained through fine-tuning on a large cross-domain dataset by Neo4j, indicating that domain relevance is more critical than dataset scale for Text-to-Cypher translation.

B. RAG setup

Overall, the RAG setup outperforms fine-tuning for both GPT-4.1 models and Gemma models. Although, fine-tuning yields larger improvements for `gpt-4.1-mini` and `gpt-4.1-nano` than for `gpt-4.1`, the RAG setup consistently improves the semantic accuracy of all GPT-4.1 models, with gains that exceed those achieved through fine-tuning. These results indicate that, in low-data settings, such as the one described in Section III-A, a RAG setup delivers more effective and consistent performance improvements than fine-tuning for code graph analysis. However, with respect to executability and schema violations, the improvement gain varies. Since the GPT-4.1 base models already exhibit high executability and schema adherence, the RAG setup leads to only marginal improvements; nevertheless, these gains are generally greater than those achieved through fine-tuning, with the exception of `gpt-4.1-mini`.

RQ3 and RQ4 summary

Although fine-tuning on a small code graph dataset generally outperforms both the base models and models fine-tuned on large cross-domain datasets, this improvement is not consistent across all models. Overall, the improvements from fine-tuning remain modest compared to those achieved with RAG, which delivers more consistent gains across both GPT-4.1 and Gemma models in low-data settings.

VI. IMPLICATION

A. Base model selection in a low-data context

Our experiments demonstrate that base model selection is the key indicator of performance across all evaluations of Cypher generation dimensions (executability, schema adherence and semantic accuracy) and that this choice cannot be compensated for by fine-tuning or RAG alone on a weaker model. Among all evaluated models, the closed-source GPT-4.1 variants have the most capable base for Text-to-Cypher translation on code graphs. The largest, `gpt-4.1`, achieves a baseline executability of 93.1% and a sufficiency rate of 41% with minimal schema violations, demonstrating strong out-of-the-box generalization to both the Cypher and the code graph structure. This baseline already approaches or exceeds the ceiling reached by smaller Gemma models even after fine-tuning or RAG enhancement. The best-performing open-source configuration, `gemma-2-9b-it` with RAG, achieves only 29% sufficiency rate, underscoring that model size and pre-training quality are decisive indicators of performance in a low-data context. Our benchmark provides a systematic framework for measuring these differences, enabling a comparison of models and adaptation strategies for Text-to-Cypher generation in code graph analysis.

B. Fine-tuning vs. RAG in low-data context

Our experiments show a consistent pattern: RAG outperforms fine-tuning in semantic accuracy across all GPT-4.1 and Gemma variants in our low-data regime (510 question-query pairs). This finding is in line with results reported in the broader literature [25], which show that RAG is particularly effective when task-specific knowledge, such as Cypher query patterns for code-graph analysis, is not well represented in a model’s pre-training distribution.

A key practical advantage of RAG is that it does not require retraining. The vector index can be updated incrementally as new question-query pairs become available, and the approach naturally adapts to evolving graph schemas without repeated fine-tuning cycles. This makes RAG particularly attractive in industrial settings, where codebases evolve rapidly and labeled data accumulates gradually. In contrast, fine-tuning offers lower inference-time latency and cost, as it does not rely on an additional retrieval step. In scenarios where responsiveness and computational efficiency are critical, a fine-tuned smaller model may therefore be a more practical choice than a larger RAG-based system, despite the trade-off in accuracy.

C. Cross-domain vs. domain-specific fine-tuning

A noteworthy finding is that the Gemma models fine-tuned on Neo4j’s large cross-domain dataset (40–44K examples), i.e., `gemma-2-9b-it-ft-neo4j` and `gemma-3-4b-it-ft-neo4j`, consistently underperform their counterparts fine-tuned on our small domain-specific dataset (510 examples). This underscores that dataset relevance outweighs dataset scale for Text-to-Cypher translation in specialized domains. Code graphs differ structurally from the entertainment or medical graphs that populate Neo4j’s benchmark; node and relationship types, query patterns, and domain vocabulary are all distinct. Fine-tuning on mismatched data can reinforce irrelevant query structures and harm generalization to the target domain. This aligns with observations in the PEFT literature [18] and carries a clear practical message: practitioners should prioritize collecting even a modest number of domain-relevant labeled examples over relying on large but off-domain datasets.

VII. LIMITATIONS

a) *Small dataset*: Our study is based on a small and highly specialized dataset. Although this limits the breadth of empirical generalization, it reflects common conditions in industrial software engineering environments, where annotated examples are scarce and domain-specific code bases are protected by confidentiality constraints. The goal of this study is to shed light on LLM-based Cypher generation for code graph analysis in precisely these low-data settings, where data availability is limited and models must operate with minimal supervision. Future work should investigate whether the trends observed here hold when larger or more diverse datasets become available.

b) *Limited prompting strategy*: We explore one possible prompt configuration for RAG, retrieving six semantically similar examples and appending them to the baseline prompt. Future work could investigate the impact of varying the number of retrieved examples and compare this approach with alternative few-shot example selection strategies, such as random sampling and fixed example sets.

c) *Single complex graph-schema*: Our experiments rely on a single graph representation extracted using the static analysis tool Renaissance, which produces a specific and relatively complex code-graph schema. This schema has been adopted by several of our industry collaborators, which supports its practical relevance, but it does constrain the generality of our findings. Different organizations may use alternative graph extraction pipelines or customized schemas tailored to their systems. Evaluating model performance across multiple schema designs and extraction methods is an important area for future work.

d) *Rapid LLM Architectural Evolution*: The landscape of large language models is evolving rapidly. New architectures, retrieval mechanisms, and training strategies may change the relative effectiveness of RAG, fine-tuning, or hybrid approaches. Our evaluation focuses exclusively on GPT-4.1 and the Gemma family of models. The GPT-4.1

family was selected because it represents one of the strongest proprietary model lines available at the time of experimentation. The Gemma models were included to enable a direct comparison with the Gemma models fine-tuned by Neo4j using a cross-domain dataset. These models differ significantly in architecture, training data, and optimization objectives, which affects how their performance should be interpreted. Our conclusions cannot be assumed to generalize across other architectures, such as models trained primarily on structured code corpora, models optimized for tool use, or models equipped with native graph-reasoning capabilities. A broader comparison across diverse architectures is needed to determine whether the observed advantages of RAG over fine-tuning persist across model classes with different inductive biases. Comparing a broader range of model families remains an important direction for future work to guide model selection.

VIII. CONCLUSION AND FUTURE WORK

This paper presented an empirical study of adaptation strategies for translating natural language questions into Cypher queries for code graph analysis. We evaluated pretrained models, fine-tuned models, and RAG setup, using an expanded version of the CodeGraphQA dataset that contains 790 annotated questions. Model performance was assessed in terms of *executability*, *schema adherence*, and *semantic accuracy*. Across all evaluated pre-trained models, `gpt-4.1` demonstrated the highest reliability and semantic accuracy, with minor syntax or schema violations. The Gemma-based model `gemma-3-27b` achieved comparable levels of executability but exhibited notably lower semantic accuracy, performing approximately 17% below `gpt-4.1`. Our experiments also showed that fine-tuning improves performance for smaller models. In contrast, larger models gain less from limited domain data. Moreover, the results highlight the importance of domain relevance in adaptation: a small code graph dataset consistently outperforms cross-domain Text-to-Cypher datasets that are over 500 times larger when used for fine-tuning. Finally, in low-data contexts, RAG provides more stable and consistent results than fine-tuning in both the GPT-4.1 and Gemma model families.

Future work should evaluate newer proprietary and open-source models to determine whether advances in architecture or training alter the relative strengths of RAG and fine-tuning. Expanding the dataset across additional systems and graph schemas would provide a stronger assessment of the robustness and generalizability of the model. Additionally, we plan to integrate these approaches into integrated development environments (IDEs) to evaluate how developers use the tool within their daily workflows, and its real-world impact on code graph analysis and developer productivity.

IX. ACKNOWLEDGMENT

This work was conducted as part of the LLM4Legacy project, in collaboration with Philips, and funded by Holland High Tech — TKI HSTM through the Public-Private Partnership Innovation Scheme (PPP-I).

REFERENCES

- [1] Saima Afrin et al. “Resource-Efficient & Effective Code Summarization”. In: *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*. IEEE. 2025, pp. 224–235.
- [2] Takuya Akiba et al. *Optuna: A Next-generation Hyperparameter Optimization Framework*. 2019. arXiv: 1907.10902 [cs.LG].
- [3] Nastaran Bassamzadeh and Chhaya Methani. “A comparative study of DSL code generation: Fine-tuning vs. optimized retrieval augmentation”. In: *arXiv preprint arXiv:2407.02742* (2024).
- [4] Chroma. *Chroma: Open-Source Embeddings Database*. Accessed: 2025-10-22. 2025. URL: <https://www.trychroma.com>.
- [5] Cursor. *cursor*. Accessed: 2026-1-23. URL: <https://cursor.com/>.
- [6] cypher. *cypher*. Accessed: 2026-1-23. URL: <https://neo4j.com/docs/cypher-manual/current/introduction/>.
- [7] Dennis Dams et al. “Developing and Applying Custom Static Analysis Tools for Industrial Multi-Language Code Bases.” In: *BENEVOL*. 2021.
- [8] Dennis Dams et al. “Model-based software restructuring: Lessons from cleaning up COM interfaces in industrial legacy code”. In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2018, pp. 552–556.
- [9] Tim Dettmers et al. *QLoRA: Efficient Finetuning of Quantized LLMs*. 2023. arXiv: 2305.14314 [cs.LG].
- [10] Catherine Finegan-Dollak et al. “Improving Text-to-SQL Evaluation Methodology”. In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 2018, 351–360.
- [11] Yunfan Gao et al. “Retrieval-augmented generation for large language models: A survey”. In: *arXiv preprint arXiv:2312.10997* (2023).
- [12] GitHub. *GitHub Copilot*. Accessed: 2024-11-03. URL: <https://github.com/features/copilot>.
- [13] Markus Hornsteiner et al. “Real-time text-to-cypher query generation with large language models for graph databases”. In: *Future Internet* 16.12 (2024), p. 438.
- [14] Edward J Hu et al. “Lora: Low-rank adaptation of large language models”. In: *arXiv preprint* (2021). DOI: 10.48550/arXiv.2106.09685.
- [15] IBM Research. *IBM Text-to-SQL Evaluation Toolkit*. Accessed: 2026-06-03. URL: <https://github.com/IBM/text-to-sql-eval>.
- [16] JetBrains. *JetBrains*. Accessed: 2026-1-23. URL: <https://www.jetbrains.com/>.
- [17] Patrick Lewis et al. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. 2020. arXiv: 2005.11401 [cs.CL].
- [18] Roman Macháček et al. “The Impact of Fine-tuning Large Language Models on Automated Program Re-pair”. In: *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2025, pp. 380–392.
- [19] Ioanna Mandilara et al. “Decoding the Mystery: How Can LLMs Turn Text Into Cypher in Complex Knowledge Graphs?” In: *IEEE Access* 13 (2025), pp. 80981–81001. DOI: 10.1109/ACCESS.2025.3567759.
- [20] OpenAI. *Supervised Fine-Tuning Guide*. Accessed: 2025-10-22. URL: <https://platform.openai.com/docs/guides/supervised-fine-tuning>.
- [21] OpenAI. *Text-embedding-3-large Model Documentation*. Accessed: 2025-10-22. URL: <https://platform.openai.com/docs/models/textembedding-3-large>.
- [22] opencypher. *openCypher Grammar*. Accessed: 2024-11-10. URL: <https://github.com/opencypher/opencypher/tree/master/grammar>.
- [23] Makbule Gulcin Ozsoy et al. “Text2cypher: Bridging natural language and graph databases”. In: *Proceedings of the Workshop on Generative AI and Knowledge Graphs (GenAIK)*. 2025, pp. 100–108.
- [24] Maksims Sirojevs. *Text2Cypher: Impact of Hard Example Selection on Model Performance*. Accessed: 2025-6-20. URL: <https://neo4j.com/blog/developer/text2cypher-impact-of-hard-example-selection/>.
- [25] Heydar Soudani, Evangelos Kanoulas, and Faegheh Hasibi. “Fine tuning vs. retrieval augmented generation for less popular knowledge”. In: *Proceedings of the 2024 Annual International ACM SIGIR Conference on Research and Development in Information Retrieval in the Asia Pacific Region*. 2024, pp. 12–22.
- [26] Tabnine. *Tabnine*. Accessed: 2024-11-03. URL: <https://www.tabnine.com/>.
- [27] Aman Tiwari et al. “Auto-Cypher: Improving LLMs on Cypher generation via LLM-supervised generation-verification framework”. In: *arXiv preprint arXiv:2412.12612* (2024).
- [28] Chaozheng Wang et al. “RAG or Fine-tuning? A comparative Study on LCMS-based Code Completion in Industry”. In: *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. 2025, pp. 93–104.
- [29] Martin Weyssow et al. “Exploring parameter-efficient fine-tuning techniques for code generation with large language models”. In: *ACM Transactions on Software Engineering and Methodology* 34.7 (2025), pp. 1–25.
- [30] Nan Yang et al. “AskGraph: A Dependency-Aware Code Assistant Powered by Code Graphs and LLM-Generated Cypher Queries”. In: *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2025), pp. 644–655.
- [31] Tao Yu et al. *Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task*. 2019. arXiv: 1809.08887 [cs.CL].

- [32] Qunjun Zhang et al. “A survey on large language models for software engineering”. In: *Science China Information Sciences* 69.4 (2026), p. 141102.