Schema and Natural Language Aware In-Context Learning for Improved GraphQL Query Generation

Nitin Gupta
IBM Research, India
ngupta47@in.ibm.com

Manish Kesarwani
IBM Research, India
manishkesarwani@in.ibm.com

Sambit Ghosh
IBM Research, India
sambit.ghosh@ibm.com

Sameep Mehta
IBM Research, India
sameepmehta@in.ibm.com

Carlos Eberhardt
IBM StepZen
carlose@ibm.com

Dan Debrunner
IBM StepZen
Dan.Debrunner@ibm.com

Abstract

GraphQL offers a flexible alternative to REST APIs, allowing precise data retrieval across multiple sources in a single query. However, generating complex GraphQL queries remains a significant challenge. Large Language Models (LLMs), while powerful, often produce suboptimal queries due to limited exposure to GraphQL schemas and their structural intricacies. Custom prompt engineering with in-context examples is a common approach to guide LLMs, but existing methods, like randomly selecting examples, often yield unsatisfactory results. While semantic similarity-based selection is effective in other domains, it falls short for GraphQL, where understanding schema-specific nuances is crucial for accurate query formulation.

To address this, we propose a Schema and NL-Aware In-context Learning (SNAIL) framework that integrates both structural and semantic information from GraphQL schemas with natural language inputs, enabling schema-aware in-context learning. Unlike existing methods, our approach captures the complexities of GraphQL schemas to improve query generation accuracy. We validate this framework on a publicly available complex GraphQL test dataset, demonstrating notable performance improvements, with specific query classes showing up to a 20% performance improvement for certain LLMs. As GraphQL adoption grows, with Gartner predicting over 60% of enterprises will use it in production by 2027, this work addresses a critical need, paving the way for more efficient and reliable GraphQL query generation in enterprise applications.

1 Introduction

GraphQL is a powerful query language and runtime for APIs, offering a flexible alternative to REST by allowing clients to request precise data from interconnected sources in a single query. At its core, GraphQL relies on a schema that defines object types, their relationships, and supported operations (queries and mutations). While this schema-driven approach enhances flexibility and efficiency, its complexity in larger systems can make generating accurate queries challenging.

According to a Gartner report (gra, 2024), by 2027, over 60% of enterprises are expected to use GraphQL in production, up from less than 30% in 2024. This rapid adoption underscores GraphQL's growing significance and the need for researchers and developers to address challenges in scalability, and usability. These advancements are essential for GraphQL to meet the evolving demands of modern enterprises.

Large language models (LLMs) can assist by generating GraphQL queries from natural language (NL) inputs, leveraging the schema to fulfill user requests. However, as noted in (Kesarwani et al., 2024), the scarcity of publicly available GraphQL datasets limits LLM exposure to schema-specific patterns, reducing their effectiveness in producing valid queries. Incorporating in-context examples in prompts has been shown to improve performance, but selecting these examples effectively is critical. Existing methods for few-shot selection typically rely on semantic similarity between the input query and NL representations of the few-shot examples. However, in the context of GraphQL, the schema's structure and relationships play a pivotal role in query formulation. This raises a key research question: *Can the GraphQL schema be leveraged to refine few-shot selection and enhance contextual relevance?* In this paper, we investigate this pivotal question and propose a novel framework that significantly enhances LLM performance for GraphQL query generation.

Contribution

We propose a Schema and NL-Aware In-context Learning (SNAIL) framework (shown in Figure 1) to enhance LLM performance in GraphQL query

Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Industry Track), pages 1009–1015

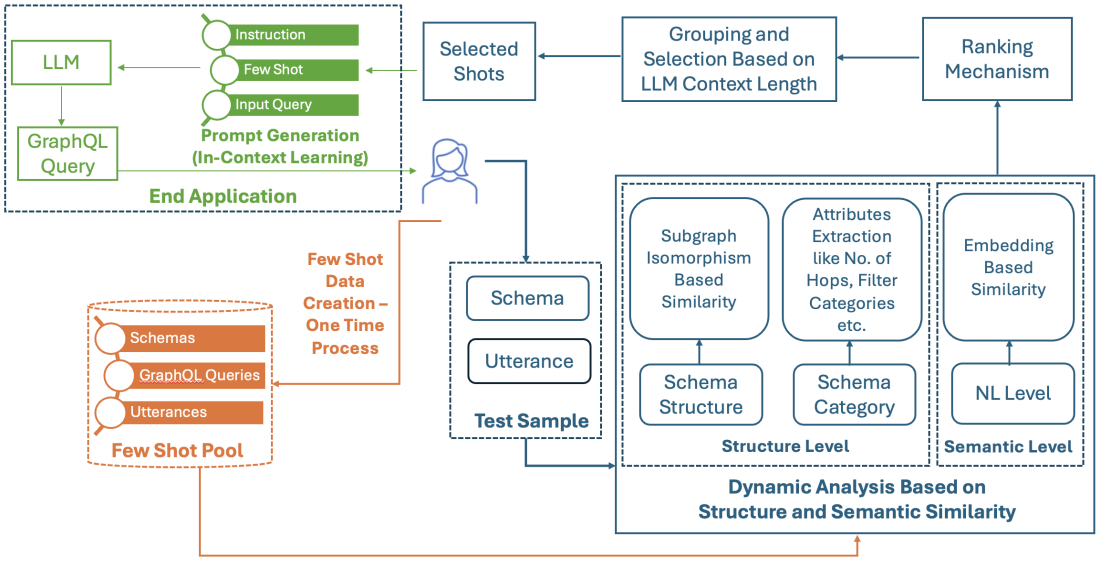

Figure 1: Proposed SNAIL Framework for GraphQL Generation.

generation by refining in-context example selection. Unlike traditional semantic similarity-based methods, SNAIL dynamically selects examples by incorporating both structural and semantic similarities tailored to each NL query. Structural similarity is evaluated using two components: (1) subgraph isomorphism to align the input schema with schemas in the few-shot pool, capturing hierarchical and entity relationships, and (2) a category-based similarity metric that incorporates schema nesting and filter relationships, ensuring comprehensive schema representation.

We implemented both the semantic similarity method and the SNAIL framework for in-context example selection and evaluated GraphQL query generation performance across 9 open-source LLMs using the test set from the only available GraphQL benchmark (Kesarwani et al., 2024). Experimental results show that SNAIL consistently improves accuracy over the semantic similarity approach across models and scenarios.

2 Related Work

GraphQL has gained significant attention in academia and industry for its flexibility and efficiency in managing data interactions. While studies have explored the advantages of GraphQL over REST APIs—such as reduced client-server interactions (Brito et al., 2019), improved maintainability (Brito and Valente, 2020), and optimized data fetching (Seabra et al., 2019; Mikuła and Dzieńkowski, 2020)—technical challenges remain. For example, (Belhadi et al., 2024) investigates testing methodologies for query validation, while (Quiña Mera et al., 2023) examines its capacity to represent complex data structures. GraphQL's role in real-world applications, including data integration across heterogeneous sources, is highlighted in (Li et al., 2024).

Recent efforts to leverage large language models (LLMs) for GraphQL query generation include several notable approaches (Levin, 2023; gql, 2023b,a; gor, 2023). However, the introduction of a GraphQL-specific dataset in (Kesarwani et al., 2024) marks the first attempt to systematically address training and evaluation for such tasks. Despite this, the study does not fully address the need for adaptive few-shot learning techniques that incorporate both semantic and structural schema characteristics.

Existing query generation systems typically rely on semantic similarity between natural language (NL) queries and examples, overlooking the critical role of schema structure. In summary, while prior research highlights GraphQL's strengths and challenges in API management, our work improves the GraphQL query generation performance of the state of the art LLMs by introducing an adaptive few-shot learning framework. This approach bridges gaps in existing methodologies, enabling LLMs to better handle the complexity and diversity of real-world GraphQL schemas.

3 Proposed SNAIL Framework

We propose the Schema and NL-Aware In-context Learning (SNAIL) framework (Figure 1) for gen-

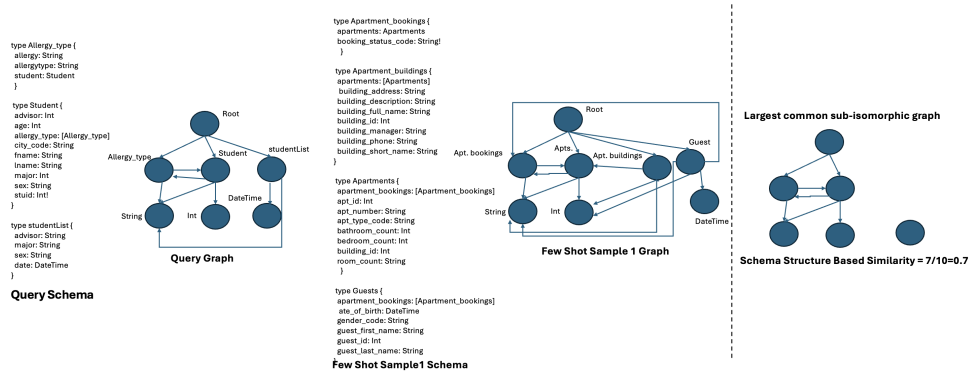

Figure 2: Illustration of Schema Structure Extraction.

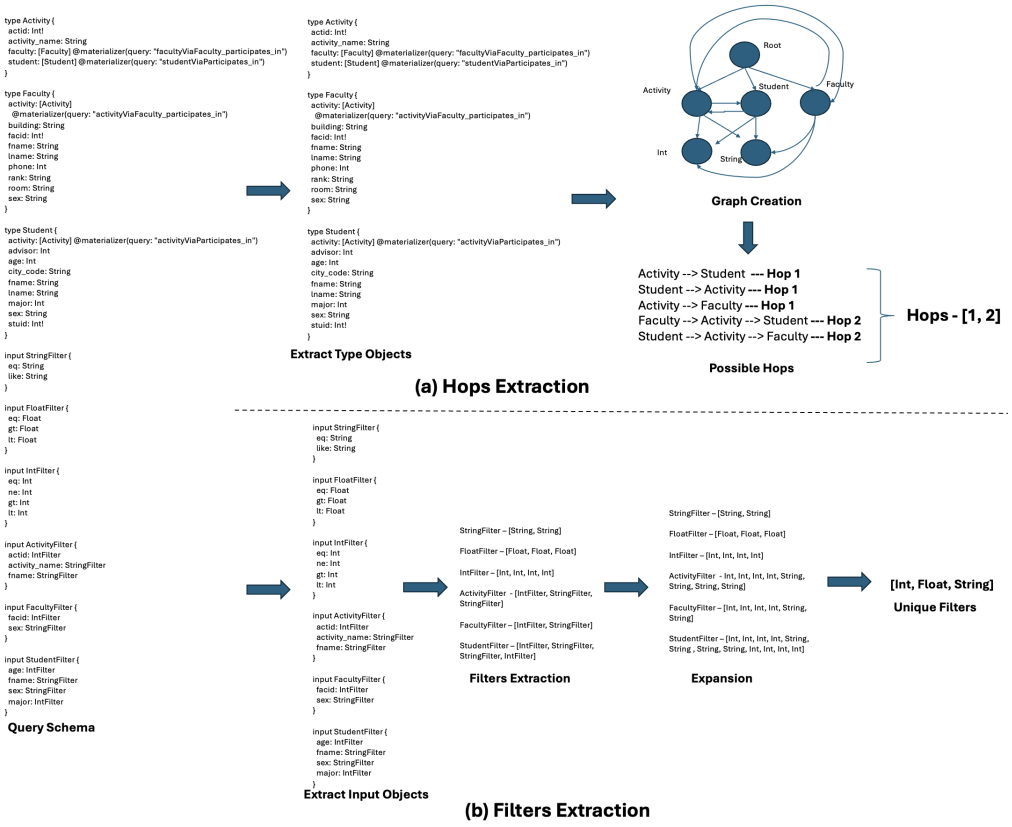

Figure 3: Illustration of Schema Categories Extraction.

erating GraphQL queries from Natural Language (NL) utterances. Unlike traditional methods relying on semantic similarity, our framework dynamically selects few-shot examples based on both structural and semantic similarities, tailored to each query.

As shown in Figure 1, the process starts by sampling and storing k few-shot examples (schemas, GraphQL queries, and corresponding utterances) from the dataset (Kesarwani et al., 2024). Upon receiving test samples, the framework assesses structural similarity between the test schema and few-shot examples, while also evaluating semantic similarity with the test utterance. These similarity metrics rank and group the examples for in-context learning, which, along with instructions and the input query, is passed to large language models (LLMs) for final GraphQL query generation. By incorporating both structural and semantic similarities, SNAIL framework improves the precision and adaptability of query generation.

3.1 Structural Similarity

We assess structural similarity through two components. First, we use subgraph isomorphism to compare the alignment between the query schema and the samples, capturing relationships like hier-

archies and entity connections. Second, we define a category-based similarity metric to account for attributes such as the number of hops (depth of nested relationships) and filter conditions, which determine data inclusion. This approach allows us to consider both high-level schema properties and operational characteristics relevant to the query.

3.1.1 Schema Structure Analysis

The Schema Structure Level focuses on assessing the structural similarity between the query sample schema and the schemas of the few-shot examples. This step is crucial for identifying samples that exhibit similar structural characteristics. To achieve this, we convert the schemas into graph representations, allowing us to analyze their structural properties more effectively.

Lets G denotes the query graph, and let $S = [S_1, S_2, ..., S_n]$ denotes the graphs corresponding to n few shot samples. The process involves finding the maximum size query schema subgraph i.e $G^{'}$ that is isomorphic to the subgraphs of the few-shot examples as shown in Figure 2. This isomorphism check enables us to determine which few-shot samples share similar structural patterns with the query schema. We have designed specific similarity metric that quantify this relationship, facilitating a more refined selection of relevant few-shot examples based on their structural alignment with the query. By leveraging these metrics, we ensure that the selected samples are not only relevant in content but also in their underlying structural organization. Schema structure similarity, ST can be calculated as:

$$ST(G, S_k) = \frac{|E(G')|}{|E(G)|} \quad (1)$$

Where $|E(G')|$ and $|E(G)|$ denote the number of edges in the isomorphic subgraph G' and the query graph G, respectively.

3.1.2 Schema Categories Analysis

The Schema Category module predicts potential scenarios from the test schema, such as filter types (Figure 3 (b)) and multi-hop relationships (Figure 3 (a)). This categorization helps select few-shot examples that match the structural complexity of the schema being queried. By analyzing the test schema, we identify key attributes and select few-shot examples from the pool that align with these categories, ensuring consistency in schema complexity. Let $Q(S)$ denote the query schema, and $F(S) = F(S)_1, F(S)_2, \ldots, F(S)_n$ represent the n schema samples in the few-shot pool. The hop category similarity (HC) is:

$$HC(Q(S), F(S)_k) = \frac{|\text{Overlap}(Q(S)^h, F(S)_k^h)|}{|Q(S)^h|}$$

Where, $Q(S)^h$ and $|F(S)_k^h$ denote the set of hops detected in $Q(S)$ and $F(S)_k$ respectively, and $Overlap(Q(S)^h, F(S)_k^h)$ denotes the overlap between these two sets.

Similarly, the filter category similarity (FC) can be calculated as:

$$FC(Q(S), F(S)_k) = \frac{|\text{Overlap}(Q(S)^f, F(S)_k^f)|}{|Q(S)^f|}$$

Where, $Q(S)^f$ and $F(S)_k^f$ denote the set of filters in $Q(S)$ and $F(S)_k$ respectively, and $Overlap(Q(S)^f, F(S)_k^f)$ denotes the overlap between these two sets.

3.2 Semantic Similarity

Semantic similarity (SS) is calculated using traditional similarity measures on embeddings, where NL queries are mapped to a high-dimensional vector space. A pre-trained LM generates these embeddings, capturing the contextual meaning and nuances of the query.

$$SS(Q(NL), F(NL)_k) = CD(Emb(Q(NL)), Emb(F(NL)_k))$$

Where, $Emb(Q(NL))$ and $Emb(F(NL)_k)$ denote the embedding of test query $Q(NL)$ and the k^{th} few-shot sample F_k respectively, and $CD(,)$ denotes the cosine similarity between these two embedding vectors.

3.3 Ranking Mechanism

To achieve effective sample selection, we propose a systematic approach utilizing three similarity metrics: schema structure, schema category, and semantic similarity. A circular selection strategy ranks samples by each metric, iteratively selecting the highest-ranked sample from schema structure, schema category, and semantic similarity in sequence. This process continues until the user-defined few-shot sample limit is met, ensuring balanced consideration of all metrics.

In cases of identical similarity scores, diversity is prioritized to ensure a comprehensive representation of structural patterns and semantic nuances.

This approach enhances the model's capacity for robust in-context learning, leading to improved accuracy in generating GraphQL queries from natural language inputs.

3.4 Grouping and Selection based on LLM Context Length

Once the samples are selected, we regroup them to fit more few-shot examples into the context. For instance, if selected samples 1 and 2 share the same schema, we combine them into a single schema with multiple queries. This approach makes more efficient use of the available context space and ensures that the model has a richer set of examples to learn from.

4 Experiments

4.1 Datasets

We use the GraphQL test set from (Kesarwani et al., 2024), which consists of 986 test triplets (GraphQL Schema, NL Query, GraphQL Query) and consider test samples from seven categories: (a) Zero Hop, (b) One Hop, (c) Two Hop, (d) Zero Hop + Filter, (e) One Hop + Filter, (f) Two Hop + Filter, and (g) Filter. The distribution of these categories is presented in Table 1. For few-shot learning, we curated 23 few-shot samples that capture different complexities of data, ensuring no schema overlap between the test and few-shot samples.

4.2 Models

We test our proposed system on 9 widely known LLMs: codellama-34b-instruct (Rozière et al., 2023), deepseek-coder-33b-instruct (Guo et al., 2024), ibm-granite-8b-code-instruct (Mishra et al., 2024), llama-3-8b-instruct (Facebook), llama-3-70b-instruct (Facebook), mixtral-8x7b-instruct-v01 (Jiang et al., 2023), llama-3-1-70b-instruct (Facebook), qwen2-72b-instruct (qwe, 2024), and prometheus-8x7b-v2 (Kim et al., 2024). GPT-4 was not included in the evaluation due to the cost associated with its API. Greedy decoding was employed to obtain outputs from the LLMs for reproducibility. The all-distilroberta-v1 [1] BERT model was used to extract embeddings for the semantic similarity module. The number of few-shot examples was set to 5. We evaluated the accuracy of the generated GraphQL queries based on their correctness.

[1] https://huggingface.co/sentence-transformers/all-distilroberta-v1

4.3 Baselines

We compared our approach against the following two baselines:

Base Model without Few-shot: This baseline uses only the instruction and test sample as input to the LLMs, without incorporating any few-shot examples.

Semantic Few-shot: This baseline uses semantic similarity to select few-shot examples. While no existing work in GraphQL explicitly applies this, we include it as a variation of our approach. Few-shot samples are retrieved based on semantic similarity for in-context learning.

4.4 Results and Discussions

The results across various complexity sub-datasets are presented in Tables 2-8, with Table 9 summarizing the overall system performance. The proposed framework shows a 10-50% improvement over the base model without few-shot examples, indicating that base models lack sufficient GraphQL exposure during training. This suggests two research directions: (a) leveraging in-context learning to provide relevant information, or (b) fine-tuning on a GraphQL-specific dataset. Given the scarcity of GraphQL training data, in-context learning with dynamic sample selection, as implemented in the SNAIL framework, emerges as the more practical approach. We also compared our approach with the standard semantic-based few-shot selection, which had not been benchmarked previously. Our method improved performance by 3-5% on average by incorporating structural and categorical similarity.

Among the evaluated models, the llama-3-70b-instruct model consistently outperformed others, with a maximum margin of 21% and a minimum of 4%. Compared to the base model, it showed a 45% overall improvement. The mixtral-8x7b-instruct-v01 model outperformed smaller models (<8B parameters) by 2-8%. Some LLMs showed improvements exceeding 10% in specific dataset complexities, demonstrating the effectiveness of our framework over the semantic-based approach. In some cases, semantic similarity performed well, likely due to the limited size of our few-shot pool. Future work will expand the pool with more complex categories to further enhance performance.

5 Conclusion

We introduce a novel few-shot learning approach for generating GraphQL queries from natural lan-

Zero Hop	One Hop	Two Hop	Filter + Zero Hop	Filter + One Hop	Filter + Two Hop	Filter
490	320	176	195	97	72	364

Table 1: Overlapping Category-wise Composition in Test Dataset.

Models	Base Model w/o Few-shot	Semantic Few-shot	SNAIL Few-shot
codellama-34b-instruct	65.1	90.2	**91.84**
deepseek-coder-33b-instruct	84.29	88.16	**91.63**
ibm-granite-8b-code-instruct	40.2	74.08	**77.35**
prometheus-8x7b-v2	37.14	85.1	**86.53**
llama-3-8b-instruct	4.29	81.43	**84.69**
llama-3-70b-instruct	44.08	86.94	**91.43**
mixtral-8x7b-instruct-v01	52.45	82.04	**84.9**
qwen2-72b-instruct	54.08	88.16	**92.45**
llama-3-1-70b-instruct	60.2	82.45	**86.53**

Table 2: Results for Zero-hop queries

Models	Base Model w/o Few-shot	Semantic Few-shot	SNAIL Few-shot
codellama-34b-instruct	57.81	72.19	**73.12**
deepseek-coder-33b-instruct	69.38	71.94	**72.81**
ibm-granite-8b-code-instruct	21.56	**56.25**	53.44
prometheus-8x7b-v2	23.75	54.69	**66.56**
llama-3-8b-instruct	2.5	53.75	**57.81**
llama-3-70b-instruct	29.38	76.25	**77.81**
mixtral-8x7b-instruct-v01	32.5	61.25	**68.44**
qwen2-72b-instruct	47.81	**75.94**	71.25
llama-3-1-70b-instruct	63.75	70.31	**74.69**

Table 3: Results for One-hop queries

Models	Base Model w/o Few-shot	Semantic Few-shot	SNAIL Few-shot
codellama-34b-instruct	36.36	46.59	**52.84**
deepseek-coder-33b-instruct	52.27	57.39	**64.2**
ibm-granite-8b-code-instruct	31.82	34.66	**43.18**
prometheus-8x7b-v2	9.66	28.98	**38.64**
llama-3-8b-instruct	0.57	28.41	**35.23**
llama-3-70b-instruct	31.82	76.14	**76.7**
mixtral-8x7b-instruct-v01	13.07	40.34	**45.45**
qwen2-72b-instruct	21.02	57.39	**61.36**
llama-3-1-70b-instruct	58.52	70.45	**73.86**

Table 4: Results for Two-hop queries

Models	Base Model w/o Few-shot	Semantic Few-shot	SNAIL Few-shot
codellama-34b-instruct	32.82	79.49	**84.1**
deepseek-coder-33b-instruct	69.23	71.79	**83.59**
ibm-granite-8b-code-instruct	27.18	**57.44**	55.38
prometheus-8x7b-v2	58.46	68.21	**74.36**
llama-3-8b-instruct	10.77	73.33	**76.92**
llama-3-70b-instruct	51.79	85.13	**89.23**
mixtral-8x7b-instruct-v01	57.95	67.69	**71.79**
qwen2-72b-instruct	30.77	71.79	**82.05**
llama-3-1-70b-instruct	53.33	82.05	**87.69**

Table 5: Results for Filter with zero-hop queries

Models	Base Model w/o Few-shot	Semantic Few-shot	SNAIL Few-shot
codellama-34b-instruct	34.02	59.79	**63.92**
deepseek-coder-33b-instruct	47.42	**61.86**	57.73
ibm-granite-8b-code-instruct	20.62	**39.18**	29.9
prometheus-8x7b-v2	26.8	38.14	**59.79**
llama-3-8b-instruct	7.22	**43.3**	**43.3**
llama-3-70b-instruct	52.58	70.1	**72.16**
mixtral-8x7b-instruct-v01	30.93	48.45	**65.98**
qwen2-72b-instruct	13.4	**61.86**	55.67
llama-3-1-70b-instruct	43.3	**71.13**	70.1

Table 6: Results for Filter with one-hop queries

Models	Base Model w/o Few-shot	Semantic Few-shot	SNAIL Few-shot
codellama-34b-instruct	13.89	15.28	**25.0**
deepseek-coder-33b-instruct	12.5	20.81	**41.67**
ibm-granite-8b-code-instruct	2.78	5.56	**6.94**
prometheus-8x7b-v2	6.94	16.67	**29.17**
llama-3-8b-instruct	1.39	**20.83**	18.06
llama-3-70b-instruct	26.39	54.17	**58.33**
mixtral-8x7b-instruct-v01	6.94	18.06	**31.94**
qwen2-72b-instruct	4.17	20.83	**27.78**
llama-3-1-70b-instruct	16.67	**58.06**	54.17

Table 7: Results for Filter with two-hop queries

Models	Base Model w/o Few-shot	Semantic Few-shot	SNAIL Few-shot
codellama-34b-instruct	29.4	61.54	**67.03**
deepseek-coder-33b-instruct	52.2	59.17	**68.41**
ibm-granite-8b-code-instruct	21.43	**42.31**	38.19
prometheus-8x7b-v2	39.84	50.0	**61.54**
llama-3-8b-instruct	7.97	54.95	**56.32**
llama-3-70b-instruct	46.98	75.0	**78.57**
mixtral-8x7b-instruct-v01	40.66	52.75	**62.36**
qwen2-72b-instruct	20.88	59.07	**64.29**
llama-3-1-70b-instruct	43.41	**76.37**	**76.37**

Table 8: Results for Filter queries

Models	Base Model w/o Few-shot	Semantic Few-shot	SNAIL Few-shot
codellama-34b-instruct	57.61	76.57	**78.8**
deepseek-coder-33b-instruct	73.73	78.8	**80.63**
ibm-granite-8b-code-instruct	32.66	61.26	**63.49**
prometheus-8x7b-v2	27.89	65.21	**71.5**
llama-3-8b-instruct	3.04	62.98	**67.14**
llama-3-70b-instruct	37.12	81.54	**84.38**
mixtral-8x7b-instruct-v01	38.95	67.85	**72.52**
qwen2-72b-instruct	46.15	78.7	**80.02**
llama-3-1-70b-instruct	61.05	76.37	**80.43**

Table 9: Results for Overall queries

guage descriptions. Our method dynamically selects relevant samples based on multi-level similarity metrics: schema structure similarity (SS), category-level similarity (HC), and natural language similarity (NL). This dynamic selection ensures that the chosen examples align with the input query's structural and semantic nuances, enhancing model performance. Evaluation across 9 widely-used LLMs shows that our approach outperforms traditional methods for few-shot selection.

References

2023a. Gqlpt.

2023b. Graphql explorer.

2023. Weaviate gorilla part 1 graphql apis.

2024. Gartner report - graphql.

2024. Qwen2 technical report.

Asma Belhadi, Man Zhang, and Andrea Arcuri. 2024. Random testing and evolutionary testing for fuzzing graphql apis. *ACM Transactions on the Web*, 18(1):1–41.

Gleison Brito, Thais Mombach, and Marco Tulio Valente. 2019. Migrating to graphql: A practical assessment. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 140–150.

Gleison Brito and Marco Tulio Valente. 2020. Rest vs graphql: A controlled experiment. In *2020 IEEE International Conference on Software Architecture (ICSA)*, pages 81–91.

Facebook. Introducing meta llama 3: The most capable openly available llm to date. `https://ai.meta.com/blog/meta-llama-3/`.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7b.

Manish Kesarwani, Sambit Ghosh, Nitin Gupta, Shramona Chakraborty, Renuka Sindhgatta, Sameep Mehta, Carlos Eberhardt, and Dan Debrunner. 2024. Graphql query generation: A large training and benchmarking dataset. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 1595–1607.

Seungone Kim, Jamin Shin, Yejin Cho, Joel Jang, Shayne Longpre, Hwaran Lee, Sangdoo Yun, Seongjin Shin, Sungdong Kim, James Thorne, and Minjoon Seo. 2024. Prometheus: Inducing fine-grained evaluation capability in language models.

Yonatan V. Levin. 2023. A developer's journey to the ai and graphql galaxy.

Huanyu Li, Olaf Hartig, Rickard Armiento, and Patrick Lambrix. 2024. Ontology-based graphql server generation for data access and data integration. *Semantic Web*, 15(5):1639–1675.

Mateusz Mikuła and Mariusz Dzieńkowski. 2020. Comparison of rest and graphql web technology performance. *Journal of Computer Sciences Institute*, 16:309–316.

Mayank Mishra, Matt Stallone, Gaoyuan Zhang, Yikang Shen, Aditya Prasad, Adriana Meza Soria, Michele Merler, Parameswaran Selvam, Saptha Surendran, Shivdeep Singh, Manish Sethi, Xuan-Hong Dang, Pengyuan Li, Kun-Lung Wu, Syed Zawad, Andrew Coleman, Matthew White, Mark Lewis, Raju Pavuluri, Yan Koyfman, Boris Lublinsky, Maximilien de Bayser, Ibrahim Abdelaziz, Kinjal Basu, Mayank Agarwal, Yi Zhou, Chris Johnson, Aanchal Goyal, Hima Patel, Yousaf Shah, Petros Zerfos, Heiko Ludwig, Asim Munawar, Maxwell Crouse, Pavan Kapanipathi, Shweta Salaria, Bob Calio, Sophia Wen, Seetharami Seelam, Brian Belgodere, Carlos Fonseca, Amith Singhee, Nirmit Desai, David D. Cox, Ruchir Puri, and Rameswar Panda. 2024. Granite code models: A family of open foundation models for code intelligence.

Antonio Quiña Mera, Pablo Fernandez, José María García, and Antonio Ruiz-Cortés. 2023. Graphql: A systematic mapping study. *ACM Comput. Surv.*, 55(10).

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code.

Matheus Seabra, Marcos Felipe Nazário, and Gustavo Pinto. 2019. Rest or graphql? a performance comparative study. In *Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse*, page 123–132.