GraphQL Query Generation: A Large Training and Benchmarking Dataset

Manish Kesarwani[1], Sambit Ghosh[1], Nitin Gupta[1], Shramona Chakraborty[1], Renuka Sindhgatta[1], Sameep Mehta[1], Carlos Eberhardt[2], Dan Debrunner [2],

[1] IBM Research, India, [2] IBM StepZen,

Correspondence: manishkesarwani@in.ibm.com

Abstract

GraphQL is a powerful query language for APIs that allows clients to fetch precise data efficiently and flexibly, querying multiple resources with a single request. However, crafting complex GraphQL query operations can be challenging. Large Language Models (LLMs) offer an alternative by generating GraphQL queries from natural language, but they struggle due to limited exposure to publicly available GraphQL schemas, often resulting in invalid or suboptimal queries. Furthermore, no benchmark test data suite is available to reliably evaluate the performance of contemporary LLMs.

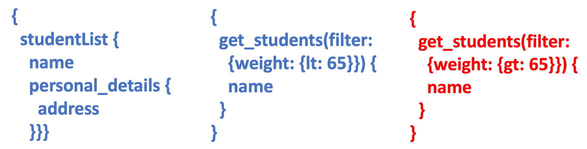
To address this, we present a large-scale, cross-domain Text-to-GraphQL query operation dataset. The dataset includes 10,940 training triples spanning 185 cross-source data stores and 957 test triples over 14 data stores. Each triple consists of a GraphQL schema, GraphQL query operation, and corresponding natural language query. The dataset has been predominantly manually created, with natural language paraphrasing, and carefully validated, requiring approximately 1200 person-hours. In our evaluation, we tested 10 state-of-the-art LLMs using our test dataset. The best-performing model achieved an accuracy of only around 50% with one in-context few-shot example, underscoring the necessity for custom fine-tuning. To support further research and benchmarking, we are releasing the training and test datasets under the MIT License. The dataset is available at https://github.com/stepzen-dev/NL2GQL.

1 Introduction

GraphQL is a query language and runtime for APIs, providing an efficient, powerful, and flexible alternative to REST APIs. It allows users to request precise data across multiple sources in a single query, minimizing extraneous information and optimizing network resource usage. This makes GraphQL ideal for applications on resource-limited devices.

```
interface Person{
  name: String
}
type Student implements Person {
  student_id: Int!
  weight: Float
  height: Float
  personal_details:[Personal_details]
}
type Personal_Details {
  address: String
  contact: String
}
input FloatFilter {
  lt: Float
}
input sFilter {
  weight: FloatFilter
}
type Query {
  studentList:[Student]
  get_students(filter:sFilter):[Student]
}
```

Figure 1: Sample GraphQL Schema



```
{
 studentList {
  name
  personal_details {
   address
  }}}
```

(a) Query 1 (Valid)

```
{
 get_students(filter:
  {weight: {lt: 65}}) {
  name
 }
}
```

(b) Query 2 (Valid)

```
{
 get_students(filter:
  {weight: {gt: 65}}) {
  name
 }
}
```

(c) Query 3 (Invalid)

Figure 2: Sample GraphQL Query Operations

Figure 1 illustrates an exemplar GraphQL schema that enables users to retrieve student information. For clarity, the resolver functions that fetch the actual data have been omitted. However, in deployment, data for type *Student* is fetched from a relational database, while their personal details are retrieved via a secure API endpoint. This illustrates how GraphQL can integrate disparate data sources into a single coherent interface for efficient data retrieval.

It is crucial to note that this schema provides only two access points for data retrieval: 1) *studentList*, which fetches details of all students, and 2) *get_students*, which filters students based on a weight predicate (less than). All permissible GraphQL query operations for this schema can utilize only these two access points. For example, the query in Figure 2a retrieves the name and address of all students, while Figure 2b returns the names of students with weight less than 65. However, the query in Figure 2c is invalid because the schema does not support a greater-than predicate for student weight. In general, challenges may also arise from inherent cyclic dependencies across type nodes, cross-source endpoints offering varying capabilities based on the data source, and custom schema extensions. This underscores the necessity for users to thoroughly understand their GraphQL schema to construct valid query operations.

An interesting alternative is to use state-of-the-art LLMs to generate GraphQL query operations from natural language queries. However, due to the limited availability of publicly accessible GraphQL schemas, these LLMs have not been sufficiently exposed to GraphQL data during their training. Consequently, they often struggle to generate valid and optimized GraphQL query operations. Furthermore, there is currently no comprehensive test dataset available to benchmark the performance of LLMs in generating GraphQL operations.

To remedy this, we present a large-scale, cross-domain Text-to-GraphQL query operation dataset. The dataset includes 10,940 training data triples spanning 185 cross-source data stores and 957 test triples spanning 14 cross-source data stores. Each data triple consists of a GraphQL schema, GraphQL query operation, and the corresponding natural language query. To facilitate this task, we designed and implemented a custom data generation pipeline along with a web-based user interface for the review and annotation of each data element. This process involved six researchers and required approximately 1,200 person-hours for data generation and validation.

Next, we developed an evaluation pipeline that takes as input the GraphQL schema, the ground-truth GraphQL query operation, and the LLM-generated operation to check for query equivalence. This pipeline was used to validate the effectiveness of our test dataset. Using this pipeline, we assessed the performance of 10 state-of-the-art LLMs, and surprisingly, the best model achieved only approximately 50% accuracy with one in-context few-shot example. These results underscore the necessity of creating complex GraphQL datasets for further model fine-tuning and benchmarking.

Contributions

To summarize, our contributions are the following:

1. A large-scale, complex, cross-domain, and cross-source Text-to-GraphQL query operation dataset, consisting of 10,940 training data elements. This dataset is designed to capture various GraphQL complexities, including Aliases, Filters, Fragments, Multiple Types, Multiple Endpoints, and Hops.

2. Prepared a separate benchmarking test dataset comprising 957 test data triples that span 14 cross-source data stores to evaluate the performance of query generation.

3. We evaluated the Text-to-GraphQL operation generation capabilities of a diverse set of contemporary LLMs and illustrated that our dataset presents a substantial challenge.

2 Dataset Creation

In this section, we begin by outlining the assumptions, followed by a comprehensive overview of the methodology employed to generate the training and test datasets. The entire dataset creation process is summarized in Figure 3.

2.1 Assumptions

We assume that users' natural language (NL) interactions would involve single-turn conversations, with all necessary parameters included in the NL text. To create a cross-source dataset, we developed custom APIs and manually embedded them into the GraphQL schema. Although schema linking and merging were beyond the scope of this paper, this approach allowed us to integrate data from multiple sources cohesively. We used IBM Stepzen as the GraphQL engine but ensured compatibility with other GraphQL engines by incorporating only those features commonly available and compliant with the latest GraphQL specifications (gql, 2021). For simplicity, we will use "GraphQL query operation" and "GraphQL query" interchangeably.

2.2 GraphQL Schema Curation

Publicly available GraphQL schemas are quite limited. For our purposes, we required a comprehensive set of schemas that could cover a diverse range

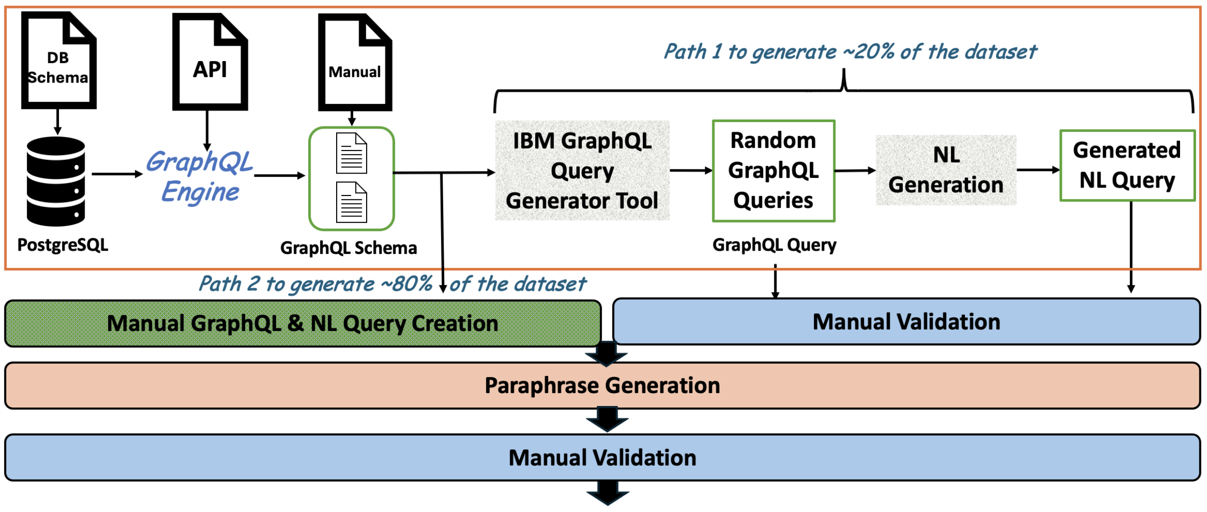


Figure 3: Dataset creation pipeline.

of GraphQL constructs. Additionally, we needed the ability to run queries on every schema to ensure valid query construction.

2.2.1 Database Selection

To address these requirements, we gathered approximately 200 relational databases from diverse sources to construct complex GraphQL schemas. The databases were selected as follows: (1) We chose 27 complex databases from the CoSQL dataset (Yu et al., 2019), (2) We populated the TPCH 1GB database (tpc, 2024), (3) We acquired 170 tables from the WikiSQL dataset (Zhong et al., 2017), and (4) We created a custom database to further enhance our schema diversity.

2.2.2 API Development

Given this set of relational databases, we created a pool of custom APIs and hosted them internally. Using the IBM Stepzen GraphQL engine, we ingested each relational database and API individually, resulting in the creation of separate GraphQL schemas for each database and API. Initially, these schemas were simple and supported only limited features, serving as the foundational structure for further enhancements.

2.2.3 Schema Enrichment

To extend the capabilities of these basic schemas, we manually edited, merged, and enriched them with a comprehensive set of available GraphQL constructs. This enrichment process aimed to expose the model to cross-source schemas comprising a variety of GraphQL constructs and enable support for a diverse range of GraphQL queries.

For instance, the basic GraphQL schema for our example schema (Figure 1) initially comprised only `Student` and `Personal_details` types, and the `studentList` field in type Query. Through our enrichment procedure, we incorporated components such as `interface Person`, `input FloatFilter`, `input sFilter`, and `get_students(filter)` to make the schema comprehensive and challenging. The detailed steps are outlined in Appendix E.

2.2.4 Schema Validation

After enriching the GraphQL schema, we deployed it to our local Stepzen instance. We then accessed and queried the schema using the Stepzen dashboard UI, enabling us to validate its accuracy and functionality.

Following this methodology, we developed a robust and diverse set of GraphQL schemas capable of supporting complex queries and varied constructs. This approach ensured that our schemas were not only functional but also enriched to cover a wide range of GraphQL features.

2.3 GraphQL Query Operation Creation

We adopted two distinct approaches for the creation of GraphQL queries.

2.3.1 IBM GraphQL Query Generator

We selected a subset of GraphQL schemas and utilized the open-source IBM rule-based GraphQL query generator tool (ibm, 2021). However, this tool has limited capabilities; it generates only basic projection GraphQL queries, lacking the ability to perform advanced query operations like deep nesting or filtering related objects.

Dataset	Alias	Multi Type	Multi Endpoint	Filter	Zero Hop	One Hop	Two Hops	Fragment
Training	20	9	15	86	25	16	4	2
Test	7	2	2	50	25	16	11	5

Table 1: Overlapping Category-wise Percentage Composition in Training and Test Datasets

Subsequently, we used the IBM Granite-20B-Code-Instruct LLM (g20, 2024) to generate corresponding NL text for these queries. Although the generated NL queries were not entirely accurate, they provided a useful baseline. Finally, we manually validated and corrected both the generated GraphQL queries and their corresponding NL text. This approach contributed to $\sim 20\%$ of our dataset.

2.3.2 Manual Query Creation

We deployed a local instance of the Stepzen engine (sz, 2024) with our GraphQL schemas. Using the Stepzen UI, we manually created the GraphQL queries and authored the corresponding NL queries. While creating, we executed each GraphQL query through the Stepzen interface to ensure they returned populated results, thereby simplifying our LLM evaluation discussed later in Section 3. Since both the GraphQL and NL queries were manually created, this process eliminated the need for the manual validation required in the first approach. This approach contributed to $\sim 80\%$ of our dataset.

2.3.3 Dataset Enhancement

After compiling the initial training dataset, we utilized the IBM Granite-20B-Code-Instruct LLM (g20, 2024) to generate approximately 8-10 paraphrases for each NL query. This strategy aimed to increase the linguistic variety within our dataset, thereby improving the robustness of our dataset. Following this, a subsequent round of manual validation was conducted, enabling us to refine the generated NL paraphrases as needed.

2.4 Training Data Distribution

We have categorized the training data into six overlapping categories. The category-wise percentage distribution, which includes intersections among these categories, is provided in Table 1. Due to space limitations, we present a brief description of each category below, with concrete examples available in Appendix A.

1. **Hops** - A hop in a GraphQL query refers to traversing from one type node to another while resolving nested object relationships. The number of hops determines the query's depth (gql, 2024). We created queries with varying number of hops, and sub-categorize them into four levels based on complexity: zero hop, one hop, two hops, and three or more hops.

2. **Filters** - It represents the conditions that data must meet to be included in the response. The filter category is subdivided into four classes based on the number of simultaneous filters an endpoint in a GraphQL schema can support: zero, one, two, and three or more.

3. **Alias** - By default, a field's response key in the response object uses the field's name. Aliases allow assigning a custom name to a response object, enabling the retrieval of the same field multiple times with different arguments or renaming fields to avoid naming conflicts.

4. **Fragments** - Fragments enable the reuse of common field selections, reducing duplicated text in queries. They are used with the spread operator (...).

5. **Multi Type** - A Multi Type GraphQL query fetches data from multiple distinct type nodes in a single query operation.

6. **Multiple-Endpoints** - A multiple-endpoint GraphQL query fetches data from multiple distinct fields in the type Query node in response to an NL request.

2.5 Test Dataset

The test dataset is generated predominantly using a different set of GraphQL schemas not present in the training dataset, ensuring minimal database overlap between the two sets. This choice was motivated by the need to ensure the generalizability of LLMs beyond their known proficiency in semantic tasks, with the expectation that, after fine-tuning with our dataset, the LLMs could generate precise and valid GraphQL queries for previously unseen schemas. The test dataset comprises 957 test data triples spanning 14 cross-source data stores, categorized into the same eight groups as the training dataset. The composition is provided in Table 1.

2.6 Final Dataset Review

To validate and ensure the quality of our dataset, we developed a custom web-based user interface for detailed review of each data element. This tool facilitated the examination of the dataset by displaying each GraphQL schema, its corresponding GraphQL query, and the associated NL query sequentially. By isolating each data element in this manner, we were able to systematically review and verify their accuracy.

In addition to enabling thorough reviews, the interface provided user-friendly, clickable options to annotate the dataset. These features included marking the number of hops, identifying filter predicates, and other key attributes. This systematic annotation process allowed for precise and consistent documentation of each data element's characteristics, thereby enhancing the overall quality and reliability of the dataset.

2.7 Team and Effort Estimate

The team comprises six researchers distributed across two geographical locations. They all possess professional fluency in the English language and bring together a diverse skill set, with expertise in GraphQL, natural language processing, software development, and large language models. Each team member was tasked with preparing a dataset encompassing various GraphQL complexities and rigorously validating each data element by reviewing the alignment between the natural language query and the GraphQL query operation against the schema. Subsequently, an independent validation round was conducted, where a set of data was reviewed by two researchers who had not previously seen it. Collectively, the team invested approximately 1,200 person-hours in the creation and refinement of the dataset.

2.8 Discussion

This current dataset is compatible with the StepZen GraphQL Engine (sz, 2024). Although we have ensured the use of standard directives, transformation scripts will still be needed to adapt the GraphQL schema and queries for reuse with other available engines. As part of our future work, we intend to develop these scripts tailored to various GraphQL engines and expand the scope of our dataset.

3 Experiments and Results

We conducted experiments with 10 state-of-the-art LLMs, with parameter sizes ranging from 3B to 34B. We employed three experimental settings: zero-shot, one-shot, and two-shot, where 'shot' refers to the number of in-context examples provided. We used greedy decoding to generate a maximum of 500 tokens for all LLMs. This study aims to: (1) demonstrate that our test dataset poses a significant challenge to the LLMs (2) assess whether providing a few in-context examples improves the models' ability to generate GraphQL queries, and (3) establish initial performance benchmarks for the Text-to-GraphQL query generation task.

During the evaluation, we constructed prompts that included instructions summarizing the generation task and incorporated 0, 1, or 2 few-shot samples, depending on the setting. Each prompt also included the test schema and the natural language query. A sample prompt is outlined in Appendix D. After obtaining results from the LLM, we executed both the generated GraphQL query and the ground truth query, and then compared their outputs to evaluate accuracy.

For each of the three experimental settings—Zero-shot, One-shot, and Two-shot—we have compiled the results in Tables 2, 3, and 4, respectively. In these tables, the first column displays the model name, while the next eight columns detail the performance of each model on individual categories of GraphQL queries. The penultimate column indicates the fraction of test cases that failed due to exceeding the available LLM context length, and the final column summarizes the overall performance across all categories. This category-wise presentation of results highlights the models' capabilities in generating respective GraphQL queries, providing insights that could guide the selection of the most appropriate model for specific use cases.

Furthermore, as shown in Table 2, the performance of pre-trained LLMs on the GraphQL query generation task is particularly poor in the zero-shot setting, with most models achieving an overall accuracy of less than 15%. This indicates that the LLMs had insufficient exposure to GraphQL data during the pre-training phase. A summary of the various types of errors in the generated GraphQL queries can be found in Appendix B.

Introducing one or two in-context examples led to marginal performance improvements in some

Model	Alias	Multi Type	Multi-endpoint	Filter	Zero Hop	One Hop	Two Hops	Fragment	Length Error	Overall
codellama-34b-instruct	0.0	0.0	33.33	13.21	19.39	8.06	6.9	0.0	0.0	12.37
flan-t5-xl	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	51.36	0.0
flan-t5-xxl	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	51.36	0.0
flan-ul2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	51.36	0.0
granite-8b-code-instruct	0.0	0.0	0.0	11.32	15.84	2.2	0.0	0.0	3.98	7.65
granite-20b-code-instruct	0.0	0.0	0.0	13.63	32.39	8.42	3.45	0.0	3.04	18.03
granite-34b-code-instruct	0.0	0.0	0.0	20.75	55.79	19.41	18.39	0.0	3.04	33.96
llama-3-8b	0.0	0.0	0.0	6.08	14.66	10.99	1.15	0.0	0.0	9.85
llama-3-8b-instruct	0.0	0.0	11.11	1.47	1.65	0.0	0.0	0.0	0.0	0.73
merlinite-7b	0.0	0.0	22.22	3.77	15.37	14.29	9.77	0.0	0.0	12.68
mistral-7b-v0-1	0.0	0.0	0.0	9.85	39.72	8.42	3.45	0.0	0.0	20.65

Table 2: Performance of LLMs on test dataset in zero shot setting.

Model	Alias	Multi Type	Multi-endpoint	Filter	Zero Hop	One Hop	Two Hops	Fragment	Length Error	Overall
codellama-34b-instruct	0.0	11.76	55.56	28.51	65.48	43.22	17.24	0.0	0.0	44.65
flan-t5-xl	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	51.36	0.0
flan-t5-xxl	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	51.36	0.0
flan-ul2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	51.36	0.0
granite-8b-code-instruct	0.0	0.0	0.0	13.21	43.26	10.99	8.62	0.0	51.36	23.9
granite-20b-code-instruct	0.0	0.0	0.0	19.08	60.05	37.36	21.84	0.0	3.04	41.93
granite-34b-code-instruct	0.0	0.0	44.44	38.36	78.01	42.12	20.11	0.0	0.0	50.31
llama-3-8b	0.0	0.0	0.0	18.45	54.37	16.48	2.3	0.0	0.0	29.25
llama-3-8b-instruct	0.0	0.0	0.0	15.51	51.54	18.68	2.3	0.0	0.0	28.62
merlinite-7b	0.0	0.0	22.22	14.47	49.88	16.12	12.64	0.0	0.0	29.04
mistral-7b-v0-1	0.0	0.0	0.0	8.18	39.24	6.23	0.0	0.0	0.0	19.18

Table 3: Performance of LLMs on test dataset in one shot setting.

Model	Alias	Multi Type	Multi-endpoint	Filter	Zero Hop	One Hop	Two Hops	Fragment	Length Error	Overall
codellama-34b-instruct	0.0	17.65	55.56	33.96	70.21	41.03	20.11	0.0	0.0	46.65
flan-t5-xl	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	51.68	0.0
flan-t5-xxl	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	51.68	0.0
flan-ul2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	51.68	0.0
granite-8b-code-instruct	0.0	0.0	0.0	13.0	42.55	12.09	13.79	0.0	51.36	24.84
granite-20b-code-instruct	0.0	0.0	0.0	21.38	62.65	41.39	21.26	0.0	3.04	43.61
granite-34b-code-instruct	0.0	0.0	44.44	36.69	75.65	45.79	19.54	0.0	0.0	50.21
llama-3-8b	0.0	0.0	0.0	17.61	52.96	16.12	5.17	0.0	3.04	29.04
llama-3-8b-instruct	0.0	0.0	0.0	16.98	48.46	19.05	6.9	0.0	3.04	28.2
merlinite-7b	0.0	0.0	11.11	18.66	53.43	20.88	11.49	0.0	0.0	31.76
mistral-7b-v0-1	0.0	0.0	0.0	8.6	40.9	8.42	1.15	0.0	0.0	20.75

Table 4: Performance of LLMs on test dataset in two shots setting.

models, with the *Granite-34B-Code-Instruct* LLM achieving the highest accuracy at around 50%. However, this improvement remains insufficient for real-world industrial applications. Models with lower context limits, such as Flan-T5, faced length errors due to prompts exceeding their token limits. This issue represents a significant bottleneck for employing in-context learning for GraphQL query generation, as GraphQL schemas are typically large and could easily surpass context limits, preventing the inclusion of even a single example. Thus, enhancing the base model's performance by tuning it with a comprehensive GraphQL dataset remains the only viable option in such cases.

This underscores the critical need for specialized GraphQL datasets to support the research community. Training datasets can be employed for fine-tuning, prompt-tuning, or enhancing prompt engineering techniques, all aimed at improving LLM performance in GraphQL query generation tasks. A brief discussion on the utility of the training dataset is provided in Appendix C. Meanwhile, the test dataset will be utilized for benchmarking generation capabilities, thereby establishing a measure of confidence in the LLM's ability to generate GraphQL queries.

4 Related Work

GraphQL has gained significant attention in both academia and industry. While there have been attempts to utilize LLMs for GraphQL query generation (Levin, 2023; gql, 2023b,a; gor, 2023), to our knowledge, there is no formal study or available training or benchmarking datasets to improve and evaluate this capability. Therefore, in this section, we briefly review the basic literature on GraphQL.

Studies comparing REST and GraphQL APIs highlight several advantages of GraphQL. For instance, (Brito et al., 2019) shows that GraphQL reduces client-server interactions and minimizes JSON payload sizes, while (Brito and Valente, 2020) demonstrates that GraphQL queries are easier to implement. Additional studies, such as (Seabra et al., 2019; Mikuła and Dzieńkowski, 2020), further explore the benefits of GraphQL.

Beyond the advantages over REST APIs, recent research has focused on testing GraphQL queries (Belhadi et al., 2024) and conducting mapping investigations (Quiña-Mera et al., 2023). GraphQL has also become a critical component in real-world applications (gq, 2024) and businesses (sz, 2024). Its potential for data access and integration across heterogeneous sources is shown in (Li et al., 2024).

Furthermore, from an industry perspective, the adoption of GraphQL is expected to increase in the near future. According to a recent Gartner report, 'By 2027, more than 60% of enterprises will use GraphQL in production, up from less than 30% in 2024' (gar, 2024).

5 Conclusion and Future Work

In this study, we created and validated a comprehensive Text-to-GraphQL query operation dataset to enhance and benchmark the performance of LLMs in generating precise GraphQL queries from natural language inputs. We employed two distinct methodologies for dataset creation, integrating both automated tools and manual query generation. This approach ensured the comprehensiveness and quality of the dataset, providing a robust resource for the enhancement and evaluation of LLM capabilities for GraphQL query generation task.

Our team, composed of six researchers across two geographical locations, invested approximately 1,200 person-hours in the creation and validation of the dataset. To ensure careful and responsible curation, we developed a custom web-based user interface for detailed review and annotation of each data element.

Future work will include expanding the dataset to relax some of the assumptions and address the limitations highlighted in this paper. We believe that our dataset will significantly contribute to advancing research in GraphQL query generation and the practical application of LLMs in the real world.

References

2021. Graphql query generator.

2021. Graphql spec.

2023a. Gqlpt.

2023b. Graphql explorer.

2023. Weaviate gorilla part 1 graphql apis.

2024. Companies using graphql.

2024. Gartner report: When to use graphql to accelerate api delivery.

2024. Granite-20b-code-instruct.

2024. Graphql query depth.

2024. Stepzen.

2024. Tpc-h benchmark.

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.

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*, (Preprint):1–37.

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

Antonio Quiña-Mera, Pablo Fernandez, José María García, and Antonio Ruiz-Cortés. 2023. Graphql: a systematic mapping study. *ACM Computing Surveys*, 55(10):1–35.

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*, pages 123–132.

Tao Yu, Rui Zhang, He Yang Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, Yi Chern Tan, Tianze Shi, Zihan Li, et al. 2019. Cosql: A conversational text-to-sql challenge towards cross-domain natural language interfaces to databases. *arXiv preprint arXiv:1909.05378*.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*.

A GraphQL Query Operation Illustration

In this section, we present the enriched version of the example GraphQL schema shown in Figure 1. Following this, we provide concrete examples of GraphQL queries for various categories, including Aliases, Filters, Fragments, Multiple Types, Multiple Endpoints, and Hops.

A.1 Enriched GraphQL Schema

To demonstrate an example of each category, we enriched the example GraphQL schema added four new type nodes. filter on height of the students, and two new fields (endpoints) in type Query – `get_courses` and `get_instructors`. The enriched schema is depicted in Figure 5.

A.2 Hops Example

Figure 6, shows an example of a three-hop GraphQL query operation corresponding to the following NL query on the enriched GraphQL schema – *Fetch all students. For each student, retrieve their name, personal details including address, contact information including email, and emergency contact details including name and phone.* .

A.3 Filter Example

Figure 7, shows an example of a two-filter GraphQL query operation corresponding to the following NL query – *Fetch students whose weight is less than 70 units and height is greater than 150 units. For each student, retrieve their name, weight, and height*

A.4 Alias Example

Figure 8, shows an example of a two-filter GraphQL query operation corresponding to the following NL query – *Fetch two lists of students: one list where students' weight is less than 70 units, and another list where students' height is greater than 150 units. For the first list, retrieve each student's name and weight, and for the second list, retrieve each student's name and height.*

A.5 Multi-Endpoint Example

Figure 9, shows an example of a two-filter GraphQL query operation corresponding to the following NL query – *Retrieve all students, including their ID, name, weight, and height. Also, retrieve all courses, including their ID, name, and the instructor's name and department.*

```
interface Person {
  name: String
}

type Student implements Person {
  student_id: Int!
  weight: Float
  height: Float
  personal_details: [PersonalDetails]
}

type PersonalDetails {
  address: String
  contact: Contact
}

type Contact {
  email: String
  emergency_contact: EmergencyContact
}

type EmergencyContact {
  name: String
  phone: String
}

type Course {
  course_id: Int!
  course_name: String
  instructor: Instructor
  students: [Student]
}

type Instructor implements Person {
  instructor_id: Int!
 department: String
}

input FloatFilter {
  lt: Float
  gt: Float
}

input StudentFilter {
  weight: FloatFilter
  height: FloatFilter
}


type Query {
  studentList: [Student]
  get_students(filter: StudentFilter): [
      Student]
  get_courses(filter: CourseFilter): [
      Course]
  get_instructors(filter:
      InstructorFilter): [Instructor]
}
```

Figure 5: Enriched GraphQL Schema

A.6 Fragment Example

Figure 10, shows an example of a two-filter GraphQL query operation corresponding to the following NL query – *Show details for two students:*

For student ID 1, give me ID and name. And, for student ID 2, give me address along with the ID and name.

A.7 Multi Type Example

Figure 11, shows an example of a two-filter GraphQL query operation corresponding to the following NL query – *Fetch course IDs, names, instructor details (names and departments), and students' names along with their weights.*

```
query HopQuery {
  get_students {
    name
    personal_details {
      address
      contact {
        email
        emergency_contact {
          name
          phone
        }
      }
    }
  }
}
```

Figure 6: Three Hop Example query

```
query FilterQuery {
  get_students(filter: { weight: { lt:
      70 }, height: { gt: 150 } }) {
    name
    weight
    height
  }
}
```

Figure 7: Two Filters query

```
query AliasQuery {
  c1: get_students(filter: { weight: {
      lt: 70 } }) {
    name
    weight
  }
 c2: get_students(filter: { height: { gt
     : 150 } }) {
    name
    height
  }
}
```

Figure 8: Alias with different filter query

```
query MultiEndQuery{
  students: get_students {
    student_id
    name
    weight
    height
  }
  courses: get_courses {
    course_id
    course_name
    instructor {
      name
      department
    }
  }
}
```

Figure 9: Multi-Endpoint Example query

```
Fragment StudentDetails on Student {
  student_id
  name
}

query WithFragments {
  student1: get_students(filter: {
      student_id: 1 }) {
    ...StudentDetails
  }
  student2: get_students(filter: {
      student_id: 2 }) {
    ...StudentDetails
   personal_details {
        address
   }
  }
}
```

Figure 10: Fragment Example query

```
query MultiTypeQuery{
  courses: get_courses {
    course_id
    course_name
    instructor {
      name
      department
    }
   student{
      name
      weight
    }
  }
}
```

Figure 11: Multi Type Example query

B Errors in the Generated GraphQL queries

The generated GraphQL queries exhibited several significant issues that negatively impacted the accuracy of LLM models, including:

1. **Hallucinated Endpoints:** Queries included GraphQL endpoints that were not present in the input schema, leading to responses based on non-existent data.

2. **Hallucinated Additional Predicates:** Erroneous filter predicates were introduced in the queries that did not align with the corresponding natural language (NL) query requirements.

3. **Selecting Incorrect Endpoint:** Although the correct fields were fetched, they were retrieved from the wrong GraphQL endpoint within the schema.

4. **Fetched Fewer Fields:** Inadequate field retrieval led to incomplete responses.

5. **Returned Additional Data:** Given that GraphQL is designed to fetch specific data, retrieving all fields from a type node when only a few are required constitutes a failure case.

6. **Sensitivity with Utterance:** Models displayed performance instability with minor variations in input, indicating a lack of robustness.

C Beyond full fine-tuning

To address the identified issues and enhance model performance beyond full fine-tuning, the following strategies could be implemented:

1. **Schema Filtering and Enrichment:** Considering that a GraphQL schema can be extensive, identifying and utilizing only the relevant parts of the schema based on the input NL query could optimize prompt efficiency. This approach not only conserves token usage, reducing inference costs, but also frees up space to include more in-context examples, thereby enhancing the guidance provided to the LLM during generation.

2. **Finding More Relevant In-context Examples:** Adding a single in-context example has shown to improve query accuracy. Expanding this to include a wider array of relevant examples can enrich the context and improve model generalization across test data.

3. **Dynamic Number of In-context Examples to Saturate Prompts:** The space available for in-context examples varies with the input GraphQL Schema. Therefore, the number of examples can be dynamically adjusted based on the available context length to enrich the context further and minimize the risk of length errors.

4. **Lightweight PEFT Tuning Including Prompt Tuning:** Rather than extensive full finetuning, parameter-efficient tuning approaches, such as prompt tuning using a subset of the training dataset, could refine model responses with minimal computational overhead.

These enhancements aim to mitigate the identified issues and improve the accuracy and reliability of GraphQL query generation by LLMs.

D LLM Prompt

Here, we present a sample prompt used with the LLMs for GraphQL query operation generation. The prompt is structured into three distinct sections:

- **Instruction:** This section defines the task and outlines the syntax expected in the generated GraphQL query.

- **In-Context Samples:** A few examples are included here to provide contextual information that aids the model in understanding the intended output.

- **Input:** This final section incorporates the input GraphQL schema along with the natural language (NL) query.

Sample LLM Prompt

Your task is to write an API request for a custom database schema based on the API reference provided. For guidance on how to correctly format this API request, consult the API reference here: Note: Please only use the API reference to understand the syntax of the request. Make sure your request is compliant with it.
Here are some quick notes about the API syntax:

- Abbreviation of any word shouldn't be used, for examples India can't be considered as IND.
- All queries should follow below format:

~~~
```{
    returnType1: subFunction1Name("
        parameter1": "value1", "
        parameter2": "value2", ...) {
      Object1
      Object2
      Object3
    ....
    }
  returnType2: subFunction2Name("
      parameter3": "value3", ...) {
      Object4
    ....
    }
  returnType3: subFunction3Name(filter:
        {"parameter4": {"operator": "
      value4"}}, ...) {
      Object5
    ....
    }
  }
~~~
```

Training Example 1:
CUSTOM SCHEMA:

```
```
type Course {
  course: String
  course_arrange: [Course_arrange]
  course_id: Int!
  staring_date: String
}
type Course_arrange {
  course: Course
  course_id: Int!
  grade: Int!
  teacher: Teacher
  teacher_id: Int!
}
type Teacher {
  age: String
  course_arrange: [Course_arrange]
  hometown: String
  name: String
  teacher_id: Int!
}

type Query {
  course(course_id: Int!): Course
  courseList: [Course]
  coursePaginatedList(first: Int, after:
      Int): [Course]
  course_arrangeList: [Course_arrange]
  teacher(teacher_id: Int!): Teacher
  teacherList: [Teacher]
}
```
```

COMMAND: "'text Give me course name and id of the all courses, also name and age of all teachers."'
API Request:

```
```
{
  courseList {
    course_id
    course
  }
  teacherList {
    name
    age
  }
}
```
```

Test Example:
CUSTOM SCHEMA:

```
```
type Accounts {
  "Account Number"
  account_no: ID
  "Address of the client"
  address: String
  "City of the client"
  city: String
  "Status of the client"
  client_status: String
  "Sub Status of the client"
  client_sub_status: String
  "Company name of the client"
  company: String
  "Country of the client"
  country: String
  "The domestic revenue of the client.
      It is included in the calculation
      of total revenue for the client."
  domestic_revenue: String
  "Employee Count of the client"
  employee_count: Float
  "The Global Revenue of the client. It
      is included in the calculation of
      total revenue for the client."
  global_revenue: String
  "Industry of the client"
  industry: String
  "Sub Industry of the client"
  sub_industry: String
}
```

```
type Contacts {
  "Street Address"
  address: String
  "City Name"
  city: String
  "Company Name"
  company: String
  "Street Address"
  company_address: String
  "City Name"
  company_city: String
  "Country Name"
  company_country: String
  "Country Name"
  country: String
  "The email address associated with the
       contact."
  email_address: String
  "First Name"
  first_name: String
  "The unique code or identifier
      associated with the job role."
  job_code: String
  "A brief description of the job role,
      providing additional context or
      details about the position."
  job_description: String
  "The official title or designation of
      the job role within the
      organization."
  job_title: String
  "Last Name"
  last_name: String
  "The phone number associated with the
      contact."
  phone_number: String
  "State Name"
  state: String
}

input StringFilter {
  like: String
}

input AccountsFilter {
  industry: StringFilter
  sub_industry: StringFilter
  city: StringFilter
}

input ContactsFilter {
  job_title: StringFilter
  state: StringFilter
  city: StringFilter
}

type Query {
  " Queries for type 'Accounts' "
  accountsList: [Accounts]
  accountsList_Filter(filter:
      AccountsFilter): [Accounts]
  contactsList: [Contacts]
}
```

"‘ COMMAND: "‘text Give me a list of Financial Markets accounts with their revenue.”’

API Request:

E Manual Schema Enrichment

The initial GraphQL schema corresponding to the schema shown in Figure 1 was generated via StepZen and it comprises of two disjoint GraphQL schema shown below:

Schema 1

```
type Student {
  student_id: Int!
  name: String
  weight: Float
  height: Float
}
type Query {
  studentList:[Student]
}
```

Figure 12: GraphQL Schema for the Student Database

Schema 2

```
type Personal_Details {
  student_id: Int!
  address: String
  contact: String
}

type Query {
  personaldetails(student_id: Int!):[
      Personal_Details]
}
```

Figure 13: GraphQL Schema for the Personal Details REST Endpoint

Now, we perform the following manual steps:

1. Extract the ‘name‘ field from the student type and create an interface to encapsulate it.

   ```
   interface Person{
     name: String
   }
   ```

2. Connect the two disjoint schemas by adding a virtual endpoint that fetches the personal details from the secure API and combines it with the student details

```
type Student implements Person {
  student_id: Int!
  weight: Float
  height: Float
  personal_details:[
      Personal_details]
}
```

3. Add a float filter capability in the schema that allows users to apply less-than filter predicates.

```
input FloatFilter {
  lt: Float
}
```

4. Attach the float filter to the weight field of the student

```
input sFilter {
  weight: FloatFilter
}
```

5. Create a new endpoint to enable users to access a filtered list of students based on their weight.

```
type Query {
  studentList:[Student]
  get_students(filter:sFilter):[
      Student]
}
```