AutoGraphQL: A Bring-Your-Own-Schema Framework for Adapting Open-Weight LLMs to GraphQL APIs

Keya Battu
IIT Kharagpur
keyabattu@kgpian.iitkgp.ac.in

Parth Mehta
DPS, Bengaluru
parthmehta78996@gmail.com

Harikrishnan Balagopal
IBM, Bengaluru
harikrishnan.balagopal@ibm.com

Manish Kesarwani
IBM, Bengaluru
manishkesarwani@in.ibm.com

Abstract

GraphQL has emerged as a widely adopted interface for accessing heterogeneous enterprise data. However, developing Natural Language to GraphQL (NL2GraphQL) systems remains challenging due to the need to rapidly adapt them to proprietary GraphQL schemas.

We present **AutoGraphQL**, a Bring-Your-Own-Schema framework for developing and evaluating NL2GraphQL systems. Given a GraphQL schema, AutoGraphQL synthesizes executable GraphQL queries, generates aligned natural language utterances, and validates them through schema and execution checks to automatically construct high-quality NL2GraphQL datasets. The framework additionally supports human-in-the-loop curation and an end-to-end workflow where the generated datasets are used to fine-tune and benchmark models.

Using AutoGraphQL, we construct a corpus of 17,621 NL2GraphQL instances spanning 173 GraphQL schemas and seven major query categories. Empirically, we show that fine-tuning open-weight models on AutoGraphQL-generated datasets substantially improves NL2GraphQL performance. Notably, the best fine-tuned 8B-parameter model achieves **42.57%** benchmark accuracy, approaching Claude Sonnet 4.5 at **44.23%**, while outperforming it on several GraphQL query categories. These results indicate that AutoGraphQL enables compact open-weight models to approach frontier-level performance on schema-grounded GraphQL generation. To support further research and benchmarking, we are releasing the training and test datasets under the MIT License. The dataset is available at https://nl2graphql-bench.github.io/.

1 Introduction

Large language models (LLMs) have advanced natural language interfaces to structured data, driven by benchmark datasets such as CoSQL (Yu et al., 2019), BIRD (Li et al., 2023), and Spider 2.0 (Lei et al., 2024). However, modern enterprise applications increasingly expose data spanning multiple services and data sources. To enable access to heterogeneous enterprise data, GraphQL has emerged as a widely adopted schema-driven interface, powering platforms such as GitHub, Salesforce, and AWS and increasingly supported by commercial database systems (Oracle, 2026).

GraphQL Schema

```
type Student {
  id: ID!
  name: String!
  email: String
  major: String!
  year: Int
  advisor: Faculty
  enrollments: [Enrollment]
}

type Enrollment {
  semester: String!
  grade: String
  course: Course!
}

type Course {
  code: String!
  title: String!
  credits: Int!
  level: String
  department: String
  instructor: Faculty
}

type Faculty {
  id: ID!
  name: String!
  department: String!
  courses: [Course]
}

input StudentFilter {
  major: String
  year: Int
  minGpa: Float
}

type Query {
  students(
    filter: StudentFilter,
    first: Int,
    after: String
  ): [Student]

  instructor(
    id: ID!
  ): Faculty
}
```

NL2GraphQL Example

Natural Language Query

List the first 10 third-year Computer Science students. Include their names, email (optionally), course details, grades, and instructor details.

GraphQL Query

```
query CourseEnrollment(
  $showEmail: Boolean!
) {
  csStudents: students(
    filter: {
      major: "CS",
      year: 3,
    },
    first: 10
  ) {
    name
    email @include(if:
      $showEmail)
    enrollments {
      semester
      grade
      course {
        ...CourseFields
        instructor {
          name
          department
        } } } } }

fragment CourseFields on
      Course {
    code
    title
    credits
    level
}
```

Figure 1: Illustrative NL2GraphQL example

According to Gartner, GraphQL adoption is expected to exceed 60% of enterprises by 2027, up from less than 30% in 2024 (Gartner, 2024). As GraphQL adoption continues to grow, rapidly adapting NL2GraphQL systems to new GraphQL APIs has become an increasingly important challenge.

NL2GraphQL presents unique natural language understanding challenges. GraphQL queries must conform to a schema-defined type system and often involve nested selection structures, filters, aliases, fragments, directives, and mutations (GraphQL, 2025). Consequently, generating GraphQL queries from natural language demands both schema grounding and compositional reasoning. For example, Figure 1 shows an NL2GraphQL example that traverses multiple schema-defined relationships while incorporating filters, aliases, and nested selection, highlighting the challenges of generating valid GraphQL queries from natural language.

Recent research has explored NL2GraphQL through benchmark datasets (Kesarwani et al., 2024), customized prompting strategies (Gupta et al., 2025), and evaluation frameworks (Sonthalia et al., 2025). However, a fundamental challenge remains largely unaddressed: GraphQL schemas are highly application-specific. Existing approaches primarily focus on fixed schemas and provide limited support for adapting NL2GraphQL systems to new domains. As a result, organizations seeking to deploy NL2GraphQL systems must typically create training data, validation infrastructure, and evaluation benchmarks for their own APIs.

To address these challenges, we present **AutoGraphQL**, a Bring-Your-Own-Schema framework for automated NL2GraphQL schema feasibility analysis, data generation, validation, fine-tuning, and benchmarking. Given a GraphQL schema and optionally a target query distribution, AutoGraphQL first determines whether the requested query categories, complexity levels, and coverage targets can be supported by the underlying schema. It then synthesizes executable GraphQL queries spanning diverse constructs, generates aligned natural language utterances, and validates generated instances through schema-aware and execution-based checks. Unlike prior NL2GraphQL resources that provide fixed datasets, AutoGraphQL enables practitioners to generate customized datasets for GraphQL schemas supported by its generators by controlling query categories, complexity, and distributional characteristics. The framework further supports optional human-in-the-loop curation and an end-to-end workflow in which generated datasets can be directly used to fine-tune LLMs and benchmark their performance, enabling rapid adaptation of NL2GraphQL systems to new GraphQL APIs.

As an instantiation of AutoGraphQL, we generate and release a large-scale NL2GraphQL resource comprising 17,621 instances spanning 173 GraphQL schemas and seven major query categories. The resulting corpus includes both training data for model adaptation and benchmark datasets for evaluation. We further provide a theoretical characterization of the GraphQL query generation space, deriving upper bounds for major query categories and enabling principled reasoning about dataset diversity and coverage. Through supervised fine-tuning, we show that AutoGraphQL-generated dataset substantially improve cross-schema NL2GraphQL performance, enabling effective adaptation of compact open-weight models to previously unseen GraphQL APIs.

Our contributions are summarized as follows:

1. We present **AutoGraphQL**, a Bring-Your-Own-Schema framework for automated NL2GraphQL feasibility analysis, data generation, validation, fine-tuning, and benchmarking for user-provided GraphQL schemas.

2. We construct a large-scale NL2GraphQL corpus comprising 17,621 executable instances spanning 173 GraphQL schemas and seven major query categories, including both training and benchmark datasets, and publicly release the training and public benchmark splits.

3. We demonstrate that AutoGraphQL-generated datasets enable effective adaptation of open-weight LLMs. The best fine-tuned 8B model achieves **42.57%** benchmark accuracy, approaching Claude Sonnet 4.5 at **44.23%** while outperforming it on several structurally complex query categories.

4. We provide a theoretical characterization of the NL2GraphQL generation space, deriving upper bounds for major query categories and enabling principled reasoning about dataset diversity, coverage, and generation capacity.

2 AutoGraphQL Framework

Figure 2 presents an overview of AutoGraphQL. In the remainder of this section, we describe the key

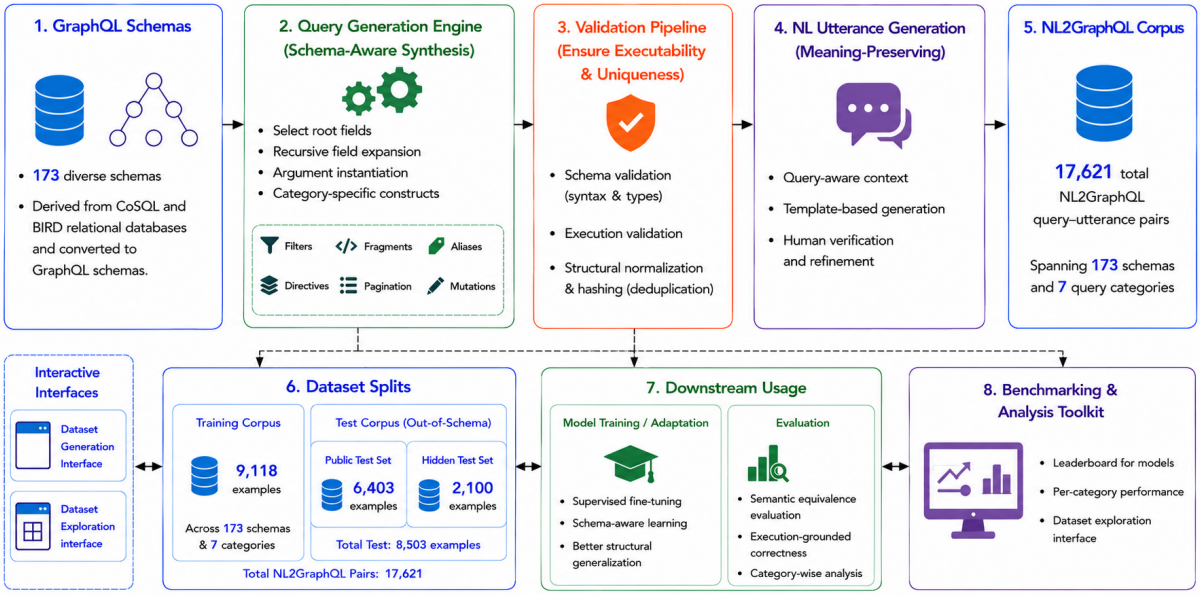


Figure 2: Overview of the proposed AutoGraphQL framework.

components of the framework.

2.1 Schema Feasibility Analysis

Practitioners often require datasets with specific characteristics, such as target query categories, complexity levels, or construct coverage. However, the ability to generate such datasets depends on the capabilities of the underlying GraphQL schema. To address this, AutoGraphQL performs a schema feasibility analysis before dataset generation begins.

Given a GraphQL schema and optional user requirements, AutoGraphQL analyzes schema properties including type hierarchies, field connectivity, nesting opportunities, argument availability, mutation support, fragment applicability, and traversal depth to determine which query categories and complexity levels can be supported. The resulting feasibility report provides estimates of achievable dataset coverage and generation capacity, guiding subsequent dataset generation. Formal definitions and upper-bound analyses for the supported query categories are provided in Appendix A.

2.2 Schema-Aware Query Synthesis

Given an input GraphQL schema, AutoGraphQL constructs an internal schema graph containing type definitions, field relationships, arguments, and operation signatures, and performs query synthesis directly over this representation.

Unlike prior work (Kesarwani et al., 2024), AutoGraphQL supports a broad range of GraphQL constructs, including filters, aliases, pagination, fragments, directives, mutations, and multi-hop traversals. Category-specific generators recursively expand schema fields while enforcing type compatibility and argument constraints. Generation is controlled through user-configurable parameters specifying query categories, nesting depth, traversal complexity, construct coverage, and target distributions, enabling the creation of diverse training and benchmark datasets ranging from simple single-hop queries to highly compositional GraphQL operations. Details of the individual generation algorithms are provided in Appendix B.

2.3 Validation Pipeline

To ensure correctness and quality, all synthesized queries undergo a multi-stage validation process:

Schema Validation. Generated queries are validated against the target schema to ensure compliance with type constraints, argument signatures, field definitions, and nesting rules.

Execution Validation. Validated queries are executed against deployed GraphQL endpoints.

Structural Deduplication. To maximize dataset diversity, queries are transformed into canonical representations and normalized using structure-preserving transformations. Hash-based signatures are then used to eliminate duplicate query patterns.

Human Verification. AutoGraphQL additionally supports optional human-in-the-loop validation through interactive interfaces, allowing users

to review, refine, annotate, and approve generated instances before inclusion in the final dataset. Appendix C presents screenshots of the validation UI.

2.4 Natural Language Utterance Generation

For each validated GraphQL query, a corresponding natural language utterance is generated using LLM-based prompting. Prompts are structured to preserve semantic fidelity to the underlying query by explicitly encoding filters, nested fields, aliases, directives, and argument values. This design ensures that utterance generation serves as a faithful realization of structured query semantics rather than introducing additional structural variability.

It is important to note that utterance generation primarily involves surface realization of an existing GraphQL query. In contrast, the NL2GraphQL task requires inferring schema-constrained query structure from natural language, which is substantially more challenging. Consequently, while LLMs assist in generating NL descriptions, the structural complexity of the resulting dataset is derived entirely from schema-grounded query synthesis.

2.5 NL2GraphQL Corpus

The output of AutoGraphQL is an NL2GraphQL corpus comprising executable triplets of the form `(GraphQL Schema, Natural Language Query, GraphQL Query)`. Each instance associates a natural language utterance with a ground-truth GraphQL query under a specific schema context. The resulting corpus serves as the foundation for both model adaptation and benchmarking.

2.6 Model Fine-Tuning and Evaluation

Beyond dataset generation, AutoGraphQL supports an end-to-end workflow for developing and evaluating NL2GraphQL systems. Generated datasets can be directly incorporated into supervised fine-tuning pipelines such as Granite Build (IBM Research, 2025), enabling rapid adaptation of LLMs to previously unseen GraphQL schemas.

For benchmarking, AutoGraphQL adapts and integrates an evaluation framework based on RGEval (Sonthalia et al., 2025), which evaluates GraphQL queries beyond exact string matching. The evaluation pipeline combines semantic-equivalence, execution-grounded correctness checking, and category-wise performance reporting to provide a comprehensive assessment.

Together, these components transform NL2GraphQL from a static benchmarking task into a practical ecosystem for schema-aware data generation, model adaptation, evaluation, and benchmarking across new GraphQL schemas.

Split	Instances
Training	9,118
Public Benchmark	6,403
Hidden Benchmark	2,100
Total	17,621

Table 1: Corpus composition.

3 Instantiating AutoGraphQL

We instantiate AutoGraphQL on 173 GraphQL schemas to generate and release a large-scale NL2GraphQL corpus. Details of the resulting dataset are presented below.

3.1 Corpus Composition

The corpus contains 17,621 NL2GraphQL instances. Each instance consists of a GraphQL schema, a natural language utterance, and a ground-truth GraphQL query. The GraphQL queries are generated through schema-aware query synthesis, while the corresponding natural language utterances are produced using *GPT-OSS-120B*. Schema and execution-based validation are performed using IBM API Connect for GraphQL, formerly StepZen (IBM, 2026) to ensure that generated queries are both syntactically valid and executable. Additional annotations collected during validation are retained to support analysis and benchmarking.

To support both model development and evaluation, the corpus is partitioned into training and benchmark splits at the schema level: 104 schemas are assigned to training, 49 to the public benchmark, and 20 to the hidden benchmark. The three schema sets are pairwise disjoint, ensuring that evaluation is performed on schemas not observed during fine-tuning. Table 1 summarizes the composition of the released corpus, corresponding to an instance-level allocation of 51.7% training, 36.3% public benchmark, and 11.9% hidden benchmark. The partition is performed by schema rather than by individual instance to preserve out-of-schema evaluation, so these instance counts follow from the selected schemas rather than a fixed target percentage. We publicly release the training and public benchmark splits (15,521 instances); the hidden benchmark (2,100 instances) is withheld for future leaderboard evaluation and is not released.

Category	Train	Test
Aliases	1,542	1,079
Directives	1,545	1,061
External Fragments	1,498	1,070
Filters	1,474	1,031
Inline Fragments	1,530	1,068
Pagination	861	616
Mutations	668	478
Total	9,118	6,403

Table 2: Category-wise distribution

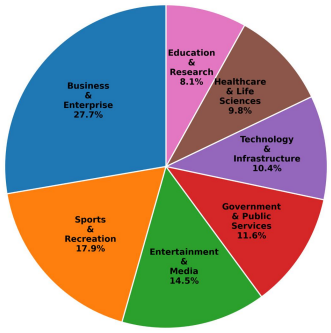


Figure 3: Domain distribution of the GraphQL schemas

3.2 Query Category Distribution

The corpus covers seven major GraphQL query categories: Aliases, Directives, External Fragments, Filters, Inline Fragments, Pagination, and Mutations. Together, these categories capture many of the constructs commonly encountered in production GraphQL APIs (GraphQL, 2025) and span a broad range of structural complexity. Table 2 summarizes the category-wise distribution of the dataset.

3.3 Schema Diversity

The 173 schemas used in this study are derived from the CoSQL and BIRD relational databases and converted into GraphQL schemas (Yu et al., 2019; Li et al., 2023). Figure 3 shows the distribution of schemas across application domains. This diversity in entity types, schema structures, and query patterns provides a realistic testbed for evaluating schema grounding and cross-schema generalization, requiring models to learn transferable reasoning capabilities rather than domain-specific memorization.

3.4 Dataset Quality and Complexity

We characterize the corpus by structural complexity and schema-level balance. Query depth, defined as the maximum number of nested object-field traversals from the root to a leaf, has a median of 4 in both the training and public-test splits , with 54.8% of training instances and 47.1% of the 6,403-query public benchmark at depth 4 or higher, and a maximum depth of 11. The released corpus contains only instances that pass structural deduplication, schema validation, and execution validation.

The 104 training schemas contain 61–103 instances each (median: 88), while the 49 public-test schemas contain 61–187 instances (median: 129). Their Gini coefficients are 0.0451 and 0.1629, and the five largest schemas account for 5.48% and 14.43% of instances, respectively, indicating limited concentration.

4 Experimental Evaluation

4.1 Experimental Setup

We evaluate whether AutoGraphQL-generated datasets can effectively adapt compact open-weight LLMs to previously unseen GraphQL schemas. Our study considers three open-weight models: *IBM Granite-3.1-8B-Instruct*, *Llama-3.1-8B-Instruct*, and *Ministral-8B-Instruct-2410*. We additionally report results for *Claude Sonnet 4.5* as a frontier-model reference.

We use the AutoGraphQL corpus introduced in Section 3, consisting of 9,118 training instances and a held-out public benchmark of 6,403 instances. All open-weight models are evaluated in a 2-shot setting ($k = 2$) before and after fine-tuning on the AutoGraphQL training corpus. A separate hidden benchmark comprising 2,100 instances is reserved for future leaderboard evaluation.

Fine-tuning is performed using the Granite.Build framework (IBM Research, 2025) with LoRA (Hu et al., 2021) under two configurations: **A** ($r = 16$, $\alpha = 32$, learning rate 10^{-4}, 3 epochs) and **B** ($r = 16$, $\alpha = 32$, learning rate 5×10^{-5}, 3 epochs). All experiments use a maximum sequence length of 4096, gradient accumulation of 8, and an effective batch size of 16. Additional implementation details, including a representative training configuration, are provided in Appendix D.

Evaluation Metric: Generated queries are evaluated using the RGEval framework (Sonthalia et al., 2025), which measures semantic equivalence beyond exact string matching and supports aliases, directives, fragments, filters, pagination, and mutations. Aliases are resolved to their underlying field names during AST normalization, so neither the dataset's `aliased<FieldName>` convention nor

Model	Aliases	Directives	External Fragments	Inline Fragments	Filters	Pagination	Mutations	Overall
Granite-3.1-8B-Instruct	9.36	4.71	16.92	16.29	17.94	33.93	33.68	16.57
Llama-3.1-8B-Instruct	**22.34**	**12.91**	24.67	**25.94**	**41.90**	48.54	46.23	**29.22**
Ministral-8B-Instruct-2410	17.15	6.79	**27.57**	23.22	35.40	**50.65**	**47.91**	26.64

Table 3: Performance (%) before fine-tuning.

Model	Setting	Aliases	Directives	External Fragments	Inline Fragments	Filters	Pagination	Mutations	Overall	Improvement
Granite-3.1-8B-Instruct	A	24.28	15.46	20.75	23.60	**71.68**	71.43	86.19	38.90	22.33
	B	25.95	14.89	24.02	22.85	68.48	70.94	**88.70**	39.14	**22.57**
Llama-3.1-8B-Instruct	A	31.97	20.08	**28.50**	28.65	67.60	71.75	87.45	**42.57**	13.35
	B	31.88	15.55	24.30	24.72	54.80	68.99	88.28	38.19	8.97
Ministral-8B-Instruct-2410	A	29.38	**22.15**	27.94	**29.03**	66.44	**71.92**	87.66	42.29	15.65
	B	**33.83**	20.17	27.76	27.53	59.65	71.43	86.82	41.23	14.59

Table 4: Performance (%) after fine-tuning.

the specific alias chosen affects equivalence: the alias score reflects structural correctness on alias-containing examples, not the model's ability to reproduce an alias name. Directives are retained during normalization, since their type, placement, and resolved condition change query semantics, but variable references are first substituted with their concrete values, so variable names do not affect equivalence; RGEval instead checks whether `@include`/`@skip` is applied to the correct element with the correct resolved condition, making directives the strictest of the seven categories. Query graphs are then compared using these normalized fields, arguments, and structural relationships.

4.2 Results

We first evaluate the out-of-the-box NL2GraphQL capabilities of open-weight LLMs. Table 3 reports the baseline 2-shot performance before fine-tuning. Overall accuracy remains modest across all models, ranging from 16.57% to 29.22%. Performance is particularly limited on categories requiring schema grounding and compositional reasoning, such as directives, fragments, and filters. These results suggest that general-purpose open-weight models struggle to generate GraphQL queries.

Table 4 reports performance after fine-tuning. AutoGraphQL-generated supervision consistently improves performance across all models, with gains of up to **22.57** percentage points. The strongest overall result is achieved by *Llama-3.1-8B-Instruct*, which reaches **42.57%** accuracy under Setting A, compared to a baseline of 29.22%. *Granite-3.1-8B-Instruct* exhibits the largest improvement, increasing from 16.57% to 39.14%. The largest gains are observed for filters, pagination, and mutations, where several models improve by more than 30 percentage points relative to their baseline performance.

Category	Claude Sonnet 4.5	Llama-3.1
Aliases	**53.66**	31.97
Directives	11.88	**20.08**
External Fragments	**42.80**	28.50
Inline Fragments	**40.73**	28.65
Filters	62.37	**67.60**
Pagination	42.05	**71.75**
Mutations	69.46	**87.45**
Overall	**44.23**	42.57

Table 5: Frontier vs. adapted open-weight LLM

To assess how closely adapted models approach frontier-model performance, Table 5 compares the strongest fine-tuned model (*Llama-3.1-8B-Instruct*) against Claude Sonnet 4.5. The adapted model achieves competitive overall performance with the frontier model and outperforms it on directives, filters, pagination, and mutations.

4.3 Robustness Across Fine-Tuning Seeds

We repeated fine-tuning of *Llama-3.1-8B-Instruct* under Setting A with three independent random seeds, evaluated on the full public benchmark: 40.57%, 43.87%, and 44.31% (mean 42.92 ± 2.04), each improving substantially on the 29.22% pre-fine-tuning baseline and confirming that the before/after improvement is not a single-run artifact. The three-seed mean of 42.92% is slightly higher than the single-run 42.57% reported in Table 4, narrowing the average gap to Claude Sonnet 4.5 (44.23%) to 1.31 points.

4.4 External Benchmark Evaluation

We evaluate our three open-weight models, before and after fine-tuning, together with Claude Son-

Model	Base (%)	Fine-tuned (%)
Claude Sonnet 4.5	56.36	–
Granite-3.1-8B-Instruct	6.87	49.31
Llama-3.1-8B-Instruct	40.89	50.31
Ministral-8B-Instruct-2410	38.92	51.20

Table 6: External benchmark accuracy (%)

net 4.5, on the GraphQL query generation benchmark of Kesarwani et al. (2024). Fine-tuning again yields substantial gains – Granite-3.1-8B-Instruct improves from 6.87% to 49.31%, Llama-3.1-8B-Instruct from 40.89% to 50.31%, and Ministral-8B-Instruct from 38.92% to 51.20% – corroborating that AutoGraphQL-generated supervision transfers beyond the released benchmark.

4.5 Error Analysis

We inspected evaluation traces to characterize common failure modes. The dominant issue is **schema hallucination**, where models generate nonexistent fields, types, filter inputs, or pagination arguments despite access to the target schema. We also observed **demonstration copying** in the 2-shot setting, with models reproducing schema elements from examples rather than grounding generation in the target schema. Finally, models frequently introduced **unnecessary or incorrect constructs**, including fragments, aliases, and directives, leading to structural mismatches even when individual fields were correct.

5 Related Work

Natural language interfaces to structured data have been extensively studied through text-to-SQL benchmarks such as Seq2SQL (Zhong et al., 2017), CoSQL (Yu et al., 2019), BIRD (Li et al., 2023), and Spider 2.0 (Lei et al., 2024). These resources, together with schema-aware approaches such as RAT-SQL (Wang et al., 2020), have significantly advanced research on semantic parsing, structured generation, and compositional generalization.

In contrast, NL2GraphQL remains relatively underexplored. Existing efforts include GraphQL-specific datasets (Kesarwani et al., 2024), schema-aware prompting approaches (Gupta et al., 2025), evaluation frameworks (Sonthalia et al., 2025), multi-agent generation systems (Liang et al., 2024), and practical query-generation solutions and applications (geobde, 2023; Rocket Connect, 2023; Arize-ai, 2024; Vo et al., 2025). While these works have advanced NL2GraphQL generation and evaluation, they primarily operate on fixed schemas and provide limited support for adapting NL2GraphQL systems to new GraphQL APIs.

A preliminary version of the data generation approach underlying AutoGraphQL was presented in (Battu et al., 2026), which demonstrated schema-aware synthetic query generation on a manually validated benchmark of 1,845 instances spanning 20 real-world schemas. The present paper substantially extends this direction into a unified framework spanning schema feasibility analysis, category-specific query synthesis, validation, and natural language generation; a corpus of 17,621 instances across 173 schemas and seven query categories; supervised fine-tuning of open-weight models; and a theoretical characterization of the generation space.

Unlike prior work that focuses on individual aspects of NL2GraphQL, and fixed datasets, our work addresses the broader challenge of adapting NL2GraphQL systems to arbitrary GraphQL schemas through a unified framework for schema-aware data generation, and benchmarking.

6 Conclusion

In this paper, we presented AutoGraphQL, a Bring-Your-Own-Schema framework for developing and evaluating NL2GraphQL systems. Given a GraphQL schema, AutoGraphQL automatically generates NL2GraphQL datasets through schema-aware query synthesis, natural language generation, and validation, while providing an end-to-end workflow for model adaptation and benchmarking.

As an instantiation of AutoGraphQL, we generated a large-scale NL2GraphQL resource comprising 17,621 instances spanning 173 GraphQL schemas and seven major query categories, publicly releasing the training and public benchmark splits. We further provided a theoretical characterization of the NL2GraphQL generation space and demonstrated that AutoGraphQL-generated supervision substantially improves cross-schema NL2GraphQL performance, enabling compact open-weight models to approach frontier-model accuracy on previously unseen GraphQL schemas.

We hope that AutoGraphQL will facilitate the development of robust and reproducible NL2GraphQL systems and provide a practical pathway for deploying natural language interfaces over proprietary GraphQL APIs.

Limitations

While our framework enables controlled and diverse NL2GraphQL dataset generation and benchmarking, several limitations remain. First, execution-based validation is performed using a single GraphQL engine, IBM API Connect for GraphQL (formerly StepZen) (IBM, 2026). Extending validation across multiple engines (e.g., Apollo and Hasura) is an important direction for improving the generalizability of the generated datasets. Second, natural language utterances are generated using GPT-OSS-120B, which may introduce model-specific biases; exploring multi-model utterance generation strategies remains an important direction for future work. Third, the 173 schemas used in this study are mechanically converted from the CoSQL and BIRD relational benchmarks (Section 3) and consequently lack GraphQL-native modeling constructs such as interface types, union types, enums, `@deprecated` directives, and Relay-style connections; the corpus should therefore not be read as representative of the full complexity of proprietary enterprise GraphQL APIs, and applying AutoGraphQL to native GraphQL schemas that use these constructs is a direct way to close this gap. Fourth, the per-category error analysis shows that gains from fine-tuning are uneven: structurally regular categories such as filters, pagination, and mutations improve by 30–55 points, while compositional categories such as external fragments, aliases, and directives remain the dominant source of residual error. The framework does not yet automatically map these prediction errors to targeted data regeneration and retraining; integrating error mining, category-aware data augmentation, and subsequent model updating into a closed feedback loop remains an important direction for future work.

References

Arize-ai. 2024. text-to-graphql-mcp. GitHub repository.

Keya Battu, Parth Mehta, and Manish Kesarwani. 2026. Synthetic data generation for schema-aware query interfaces: Benchmarking nl2graphql systems. In *Proceedings of the Workshop on Synthetic Data Generation and Management for Building AI Systems*.

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

geobde. 2023. graphqlexplorer. GitHub repository.

GraphQL. 2025. Graphql. `https://graphql.org/`.

Nitin Gupta, Manish Kesarwani, Sambit Ghosh, Sameep Mehta, Carlos Eberhardt, and Dan Debrunner. 2025. Schema and natural language aware in-context learning for improved graphql query generation. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 3: Industry Track)*, pages 1009–1015.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

IBM. 2026. IBM API Connect for GraphQL as a Service Documentation. `https://www.ibm.com/docs/en/api-connect-graphql/saas`. Accessed: 2026-06-17.

IBM Research. 2025. Granite build. `https://github.com/ibm-granite/granite.build`. Accessed: 2026-06-17.

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.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, Victor Zhong, Caiming Xiong, Ruoxi Sun, Qian Liu, Sida Wang, and Tao Yu. 2024. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. *arXiv preprint arXiv:2411.07763*.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C.C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Yuanyuan Liang, Tingyu Xie, Gan Peng, Zihao Huang, Yunshi Lan, and Weining Qian. 2024. Nat-nl2gql: A novel multi-agent framework for translating natural language to graph query language. *arXiv preprint arXiv:2412.10434*.

Oracle. 2026. *Oracle Database Support for GraphQL Developer's Guide*. Oracle.

Rocket Connect. 2023. gqlpt. GitHub repository.

Vansika Sonthalia, Manish Kesarwani, and Sameep Mehta. 2025. Robust evaluation of llm-generated graphql queries for web services. In *IEEE International Conference on Web Services (ICWS)*.

Ngoc Phuoc An Vo, Manish Kesarwani, Ruchi Mahindru, and Chandrasekhar Narayanaswami. 2025. Finops agent – a use-case for it infrastructure and cost optimization. *Preprint*, arXiv:2510.25914.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. In *Proceedings of the 58th annual meeting of the association for computational linguistics*, pages 7567–7578.

Tao Yu, Rui Zhang, He Yang Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, Yi Chern Tan, Tianze Shi, Zihan Li, Youxuan Jiang, Michihiro Yasunaga, Sungrok Shim, Tao Chen, Alexander Fabbri, Zifan Li, Luyao Chen, Yuwen Zhang, Shreya Dixit, and 5 others. 2019. Cosql: A conversational text-to-sql challenge towards cross-domain natural language interfaces to databases. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 1962–1979.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1985–1995.

A Theoretical Upper Bounds on Generatable Queries

This section gives tight, implementation-accurate upper bounds on the number of syntactically distinct queries our generator can produce for a fixed schema. We count *normalized* query forms (canonical field order, fixed variable names); two queries are distinct iff their normalized syntactic forms differ.

A.1 Notation and setup

Let S be a GraphQL schema and $\mathcal{R}$ its root query fields. For any object type t denote

- $\mathcal{F}(t)$: selectable fields of t, with $a(t) = |\mathcal{F}(t)|$;
- $\mathcal{S}(t) \subseteq \mathcal{F}(t)$: scalar (and enum) fields, with $s(t) = |\mathcal{S}(t)|$;
- $r_s(t)$: required (non-null) scalar fields, and $o_s(t)$: optional (nullable) scalar fields, so $s(t) = r_s(t) + o_s(t)$;
- $\mathcal{O}(t) = \mathcal{F}(t) \setminus \mathcal{S}(t)$: object fields.
- $\mathcal{R}_s(t) \subseteq \mathcal{S}(t)$: the required scalar fields of t, so $|\mathcal{R}_s(t)| = r_s(t)$.

For input object types u write $\mathcal{R}(u)$ and $\mathcal{O}(u)$ for required and optional input fields with sizes $r(u)$ and $o(u)$. Let m be the per-type selection cap (`max_fields`) and $d_{\max}$ the maximum nesting depth; remaining depth is d with $0 \le d \le d_{\max}$. Let W be an upper bound on the number of representative value patterns the generator uses per input scalar (empirically $W > 1$). Define:

- $S_{\text{args}}(\mu)$ and $S_{\text{resp}}(\mu)$: the number of argument and response shapes the mutation generator uses for mutation field μ (implementation: required args fixed, S_{args} typically 1; response shapes are subsets of scalars capped at 3);
- $n_{\max}(t, d)$: an upper bound on the number of directive-eligible sites the generator may annotate in any selection for type t at depth d;
- $K_{p(r)} \le 2$: number of pagination patterns used by the generator for a paginated root r (implementation: one pagination style and at most two patterns).

A.2 Selection-set recurrence

Let $B(t, d)$ be the number of distinct, normalized, non-empty selection sets for type t with remaining depth d. The generator always includes required scalars and only varies optional scalars; thus the leaf (depth 0) count is

$$B(t,0) = \begin{cases} 2^{\,o_s(t)}, & r_s(t) > 0, \\ 2^{\,o_s(t)} - 1, & r_s(t) = 0, \end{cases}$$

(i.e., required scalars are present; if there are no required scalars the empty scalar selection is excluded). For $d \ge 1$, required scalars remain mandatory, exactly as at $d = 0$; the combinatorial choice is only over the remaining fields $C(t) := \mathcal{F}(t) \setminus \mathcal{R}_s(t)$ (the optional scalars and object fields), with $c(t) := |C(t)| = a(t) - r_s(t)$. We exactly enumerate non-empty selections up to the cap m:

$$B(t,d) = \sum_{k=k_0(t)}^{\min(m-r_s(t),\,c(t))} \; \sum_{\substack{T \subseteq C(t) \\ |T|=k}} \prod_{f \in T} h_f(d)$$

where $k_0(t) = 0$ if $r_s(t) > 0$ and $k_0(t) = 1$ if $r_s(t) = 0$ (excluding the fully empty selection, as at $d = 0$), and

$$h_f(d) = \begin{cases} 1, & f \in \mathcal{S}(t) \setminus \mathcal{R}_s(t) \text{ (optional scalar)}, \\ B(\text{type}(f),\, d-1), & f \in \mathcal{O}(t) \text{ (object field)}. \end{cases}$$

This reduces to the $d = 0$ case above whenever $\mathcal{O}(t) = \emptyset$, since $C(t)$ then consists only of the $o_s(t)$ optional scalars and the sum over k telescopes to $2^{o_s(t)}$ (or $2^{o_s(t)} - 1$ when $r_s(t) = 0$) once the cap does not bind — so the two cases are consistent rather than independently stipulated. When a type has no selectable scalars at a level, the generator's nesting logic selects a default object field when $d \geq 1$, so non-empty selections remain possible for $d \geq 1$.

For convenient estimation define $M_{\max} = \max_{t,d} \max_{f \in C(t)} h_f(d)$; then

$$\boxed{B(t, d) \leq (1 + M_{\max})^{a(t)}.}$$

(We state this without the "-1" correction used at $d = 0$: that correction is only valid when $r_s(t) = 0$, and can otherwise make the bound too tight, e.g. failing when $M_{\max} = 0$ and $r_s(t) > 0$. The looser bound above holds unconditionally.)

A.3 Category-specific bounds

Basic queries. Each root $r \in \mathcal{R}$ with return type t_r admits at most $B(t_r, d_{\max})$ normalized selection bodies, hence

$$\boxed{UB_{\text{basic}} = \sum_{r \in \mathcal{R}} B(t_r, d_{\max}).}$$

Filter queries. For an input object u with $r(u)$ required and $o(u)$ optional fields and per-field pattern bound W, the generator always includes all required fields and chooses any subset of optional fields (excluding the empty subset when $r(u) = 0$). Therefore

$$\boxed{V(u) = \begin{cases} \displaystyle\sum_{j=0}^{o(u)} \binom{o(u)}{j} W^{\,r(u)+j}, & r(u) > 0, \\ \displaystyle\sum_{j=1}^{o(u)} \binom{o(u)}{j} W^{\,j}, & r(u) = 0, \end{cases}}$$

and the schema-level filter bound is

$$\boxed{UB_{\text{filter}} = \sum_{r \in \mathcal{R}} \mathbb{I}[r \text{ has filter}] \cdot V(u_r) \cdot B(t_r, d_{\max}).}$$

Fragment queries (external & inline). The generator creates at most one fragment body per query, so each fragment style individually adds no extra combinatorics beyond the selection builder. External and inline fragments are distinct, separately generated and separately reported categories (Table 2), so we bound them as two separate terms rather than one shared bound:

$$\boxed{UB_{\text{extfrag}} = UB_{\text{infrag}} = \sum_{r \in \mathcal{R}} B(t_r, d_{\max}).}$$

Directive queries. For any selection the generator may annotate any directive-eligible site with one of {none, `@include`, `@skip`}. Let $n_{\max}(t, d_{\max})$ upper bound eligible sites in selections for type t. The directive variants produced per selection are therefore at most $3^{\,n_{\max}(t, d_{\max})}$, so

$$\boxed{UB_{\text{dir}} = \sum_{r \in \mathcal{R}} B(t_r, d_{\max}) \cdot 3^{\,n_{\max}(t_r, d_{\max})}.}$$

Alias queries. When the aliasing mode is used the generator deterministically aliases every chosen subfield using a fixed pattern (e.g., `aliased<FieldName>`). Alias names are therefore fixed given the selection body and do not multiply combinatorial choices:

$$\boxed{UB_{\text{alias}} = \sum_{r \in \mathcal{R}} B(t_r, d_{\max}).}$$

Mutation queries. Each mutation field μ is generated with a fixed (required) argument shape and a family of scalar response shapes. Let $S_{\text{args}}(\mu)$ and $S_{\text{resp}}(\mu)$ be the generator's argument/response shape counts for μ, and let I and D denote, respectively, the schema's insert (create) and delete mutation fields targeted by the generator, so that $I \cup D$ is the full set of mutation fields covered by this bound. Then

$$\boxed{UB_{\text{mut}} = \sum_{\mu \in I \cup D} S_{\text{args}}(\mu)\, S_{\text{resp}}(\mu).}$$

Pagination queries. Pagination is applied only to root fields; for each paginated root r the generator uses a single pagination style and at most two argument patterns. Let $K_{p(r)} \leq 2$ be that count. Then

$$\boxed{UB_{\text{page}} = \sum_{r \in \mathcal{R}_{\text{page}}} B(t_r, d_{\max}) \cdot K_{p(r)}.}$$

A.4 Overall bound and remark

A conservative overall upper bound on syntactically distinct queries produced by the generator is the sum of the per-category bounds above:

$$\begin{aligned} UB_{\text{total}} = {} & UB_{\text{basic}} + UB_{\text{filter}} + UB_{\text{extfrag}} \\ & + UB_{\text{infrag}} + UB_{\text{dir}} + UB_{\text{alias}} \\ & + UB_{\text{mut}} + UB_{\text{page}} \end{aligned}$$

Note that UB_{basic}, UB_{extfrag}, UB_{infrag}, and UB_{alias} coincide by construction, since fragment and alias generation deterministically transform a selected query body without introducing additional combinatorial choices; each still yields syntactically distinct query forms (plain, externally fragment-wrapped, inline fragment-wrapped, or alias-annotated), so summing them is not double-counting a single query category. These bounds characterize the maximum generation capacity supported by a given schema and configuration; they are not intended to match the released corpus size, since AutoGraphQL samples a target number of instances per category rather than exhaustively enumerating the space. Accordingly, UB_{total} should be read as a feasibility and capacity indicator rather than a corpus-coverage estimate.

B Generation Framework Details

This appendix provides additional implementation details for the schema-aware generation framework described in Section 2. The framework follows a query-first design in which executable GraphQL queries are synthesized directly from schema metadata before corresponding natural language utterances are generated.

B.1 Dataset Generation Algorithm

Algorithm 1 summarizes the overall dataset generation process. Given an input schema and configuration parameters, the framework repeatedly samples query categories, synthesizes candidate queries, validates their correctness, and retains only structurally unique executable instances.

Algorithm 1 GraphQL Query Synthesis Overview

Input: schema S, configuration C, target size N
Output: dataset D

$D \leftarrow \emptyset$, Seen $\leftarrow \emptyset$
while $|D| < N$ **do**
  cat $\leftarrow$ sampleCategory(C)
  gen $\leftarrow$ getGeneratorFor(cat)
  $(q, vars) \leftarrow$ gen.generate(S)
  sig $\leftarrow$ NormalizeAndHash(q)
  if sig $\in$ Seen **then**
    continue
  end if
  if not validateSyntax(q, S) **then**
    continue
  end if
  if C.enableExecution AND **not** validate($q, vars$) **then**
    continue
  end if
  Seen $\leftarrow$ Seen $\cup$ sig
  $D \leftarrow D \cup (q, vars, cat)$
end while
return D

B.2 Recursive Field Expansion

The framework constructs GraphQL selection sets through recursive traversal of schema types. Expansion depth is bounded by configurable parameters, enabling explicit control over structural complexity and compositional difficulty.

Algorithm 2 BuildFields(type, depth, max_depth)

Input: GraphQL object type *type*, current selection depth *depth*, maximum depth *max_depth*
Output: Selection set

if depth $>$ max_depth **then**
  return ""
end if
fields $\leftarrow$ sample(getFields(type))
for f in fields **do**
  if isObject(f.type) **then**
    recursively expand nested fields
  else
    add scalar field
  end if
end for
return selectionSet

B.3 Category-Specific Query Generation

Different GraphQL constructs are synthesized using specialized generators built on top of the shared

recursive expansion framework. Depending on the selected category, additional transformations are applied to introduce filters, aliases, fragments, directives, pagination operators, or mutations while preserving schema correctness.

Algorithm 3 GenerateQueryByCategory

Input: schema S, category C
Output: executable GraphQL query

```
root ← pickRootField(S, C)
body ← BuildFields(returnType(root), 0, max_depth)
apply category-specific transformations
assemble final query
return query
```

C Interactive Interfaces

Figure 4 presents screenshots of the interactive interfaces provided by the NL2GraphQL toolkit. The Dataset Generation Interface enables users to create customized datasets by selecting schemas, query categories, and complexity constraints. The Dataset Exploration Interface supports browsing, filtering, validation, and export of generated query–utterance pairs.

```yaml
tuning_config:
  use_lora: true
  chat_template_name: tulu

  lora_rank: 16
  lora_alpha: 32
  lora_dropout: 0.05

  learning_rate: 1e-4
  num_train_epochs: 3

  max_seq_length: 4096

  per_device_train_batch_size: 1
  gradient_accumulation_steps: 8

  gradient_checkpointing: true
  use_flash_attn: true

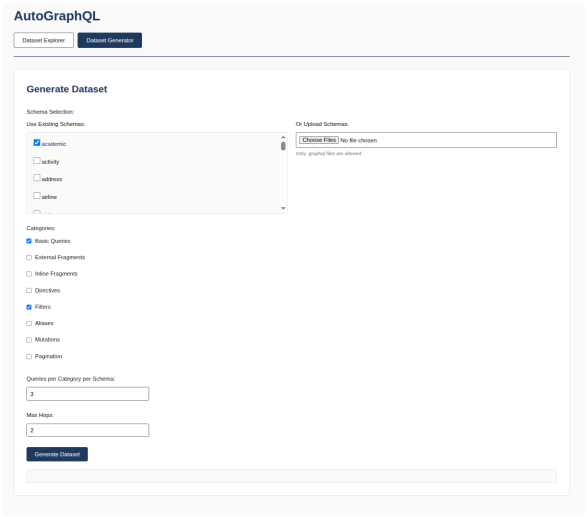
  warmup_ratio: 0.03
  lr_scheduler_type: cosine

compute_config:
  num_gpus_per_node: 2
  num_nodes: 1
```

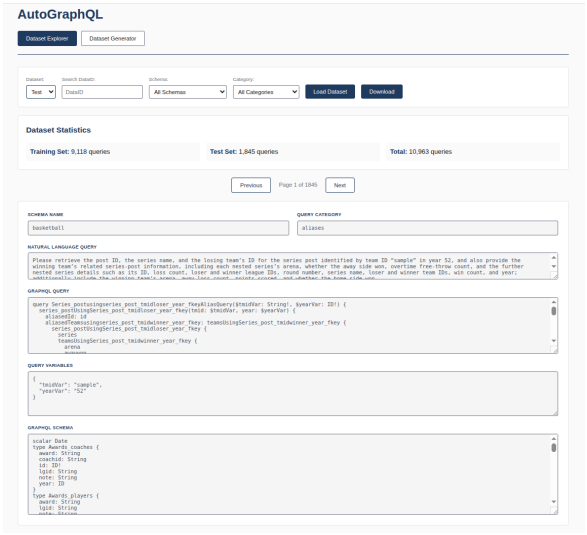
Listing 1: Representative LoRA training configuration corresponding to Setting A

D Fine-Tuning Configuration

To facilitate reproducibility, Listing 1 shows a representative training configuration used for LoRA adaptation of Granite-3.1-8B-Instruct under Variant A. The configuration specifies the LoRA parameters, optimization settings, sequence length, and compute resources used throughout our experiments.



(a) Dataset Generation Interface



(b) Dataset Exploration Interface

Figure 4: Interactive interfaces for reviewing NL2GraphQL instances.