Robust Evaluation of LLM-Generated GraphQL Queries for Web Services

Vansika Sonthalia
SRM Institute of Science and Technology, India
sonthaliavansika@gmail.com

Manish Kesarwani, Sameep Mehta
IBM Research, India
manishkesarwani@in.ibm.com, *sameepmehta@in.ibm.com*

***Abstract*—GraphQL offers a flexible alternative to REST APIs, enabling precise data retrieval across multiple services—a critical requirement in today's service-oriented architectures. However, constructing complex GraphQL queries remains challenging, and even Large Language Models (LLMs) often generate suboptimal queries due to limited schema awareness. Recent advancements, such as specialized prompt engineering, schema-aware in-context learning, and dedicated datasets, have aimed to improve query generation. However, evaluating the quality of generated queries remains challenging: GraphQL's inherent flexibility allows semantically equivalent queries to differ syntactically, complicating both automatic and manual evaluation.**

In this work, we introduce Robust GraphQL Evaluation (RGEval), the first benchmarking pipeline designed to systematically assess the quality of LLM-generated GraphQL queries. RGEval effectively handles schema complexities and structural variations, ensuring accurate evaluations while significantly improving efficiency—reducing evaluation time from hours to minutes. With Gartner projecting that over 60% of enterprises will adopt GraphQL in production by 2027, RGEval provides a critical solution for benchmarking LLM-generated queries, fostering trust in AI-driven web service consumption.

I. Introduction

GraphQL [1] is a modern API query language that provides a flexible and efficient alternative to traditional REST APIs. It enables precise data retrieval through a single query, avoiding over-fetching and under-fetching, which in turn optimizes network performance. Its schema-based approach, with strong typing and self-documentation, simplifies API integration and supports the management of heterogeneous data sources within complex, distributed systems. By enabling precise, dynamic, and client-driven composition of data across services, this design facilitates efficient service orchestration and scalable interactions, addressing core research challenges in web services and service-oriented architectures.

Recent forecasts indicate that by 2027 over 60% of enterprises will deploy GraphQL in production, a significant rise from less than 30% in 2024 [2]. This trend underscores GraphQL's growing importance in environments such as e-commerce, financial systems, and healthcare information systems, where integrating diverse data sources and optimizing API interactions are crucial. The advancements in API technology and distributed system design exemplified by GraphQL align well with current research efforts focused on next-generation web service integration and dynamic service orchestration.

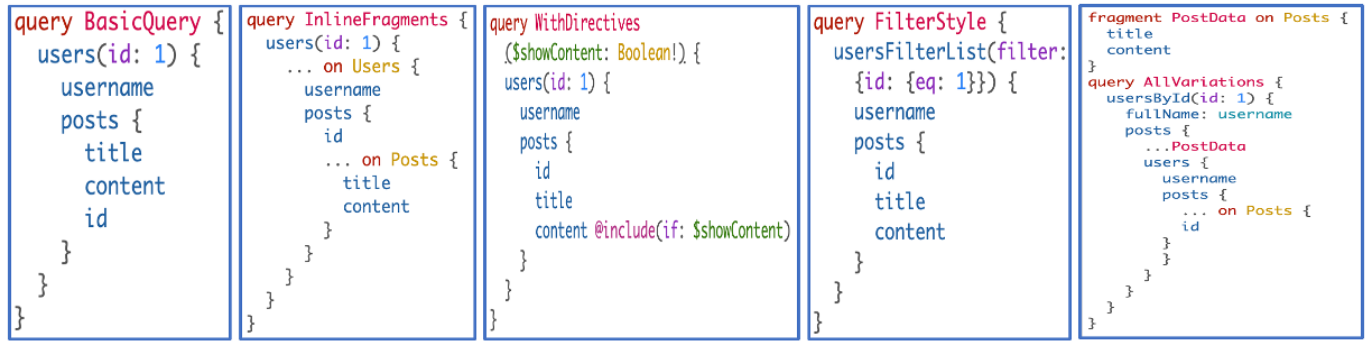
```
type Posts {
  content: String!
  id: Int!
  title: String
  user_id: Int
  users: Users # Corresponding User details
}

type Users {
  id: Int!
  username: String!
  posts: [Posts] # Corresponding posts
}

input IntFilter {
  eq: Int
  gt: Int
}

input StringFilter {
  eq: String
  like: String
}

input UserFilter {
  id: IntFilter
  username: StringFilter
}

type Query {
  posts(id: Int!): Posts
  users(id: Int!): Users
  usersFilterList(filter: UserFilter): [Users]
  usersById(id: Int!): Users # Redundant Field
}
```

Listing 1: GraphQL Schema for Blogging Service

Listing 1 shows a sample GraphQL schema for a blogging service with the following types: **Posts**, **Users** and **Query**. The `Posts` type represents a blog post with `content`, `id`, an optional `title`, and a `user_id` referencing the author. The `users` field in `Posts` establishes a relationship with the `Users` type, which contains an `id`, a required `username`, and a list of `posts` authored by the user. The schema also includes filtering capabilities through input type `UserFilter`, enabling filtering of users based on `id` and `username`. The `Query` type defines multiple entry points for data retrieval: fetching a post or user by `id`, filtering users

(a) LLM Generated Query 1 (b) LLM Generated Query 2 (c) LLM Generated Query 3

Fig. 1: Equivalent GraphQL Query operations (LLM generated)



(a) Basic Query (b) Inline Fragment (c) With Directives (d) Custom Filter (e) Mixed Variations

Fig. 2: Other Equivalent GraphQL Query operations

via `usersFilterList`, and an alternative (redundant) field, `usersById`, for retrieving users by `id`.

GraphQL Query Operation Generation

Constructing complex GraphQL queries, formally called query operations, remains challenging, leading to the exploration of LLM-based solutions where a natural language (NL) query is transformed into a GraphQL query via structured prompts. This approach provides a flexible mechanism for users to retrieve data from web services, enabling seamless interaction with APIs without requiring deep expertise in GraphQL syntax. However, as demonstrated in [3], state-of-the-art LLMs struggle with GraphQL query generation due to limited exposure to GraphQL data during training. Recent advancements in specialized prompt engineering and schema-aware in-context learning [4] have helped improve LLM-driven query generation. Additionally, the dedicated dataset [3] enables fine-tuning and benchmarking, a crucial step in ensuring trust in LLM deployment for production environments.

However, a key challenge in evaluation arises from GraphQL's syntactic flexibility—multiple queries can retrieve identical data using different constructs. The choice of prompt and phrasing of the NL query can lead LLMs to generate functionally equivalent but structurally distinct (maybe suboptimal) queries. For example, Figure 1 illustrates three GraphQL query variants generated by `Llama-3-405b-instruct`, all returning the same data. Queries in Figures 1a and 1b resulted from the same prompt but with different paraphrases of the same NL query, while Figure 1c stemmed from slight modifications in prompt instructions. Additionally, Figure 2 presents other possible variations of the same query, highlighting the diversity of syntactic forms. Due to space constraints, a sample prompt is provided in Appendix A, while the rest of the prompts are available in the technical report [5]. Given this inherent variability, it is essential to develop an effective mechanism to assess whether a generated query is equivalent to the ground truth (GT) query in benchmark datasets.

Need for Robust GraphQL Query Validation

Traditional evaluation approaches for GraphQL queries typically rely on two methods [3]: (a) string matching and (b) execute-and-match. However, as evident from Figure 1, string matching is insufficient for establishing query equivalence because GraphQL queries can be formulated using a variety of constructs and stylistic differences, resulting in distinct string representations even when they retrieve equivalent data.

In contrast, the execute-and-match approach evaluates equivalence by executing both the generated and ground truth queries and comparing their responses. However, this method also encounters several challenges. For instance, alias usage can lead to differences in projected field names, affecting the output comparison despite the underlying data being equivalent. Moreover, query execution may yield an empty result set or over-fetch data, further complicating validation. Even when the query results match, it does not necessarily imply that the queries themselves are semantically equivalent—identical

outputs can arise due to specific characteristics of the dataset, such as data sparsity or redundancies in stored values.

This approach additionally requires a carefully configured test environment where all benchmark schemas are deployed on an active GraphQL server connected to live data sources. When large datasets are involved, extra mechanisms are needed to ensure that the results returned by the ground truth query align with those from the generated query. The non-deterministic order of returned results—especially when data is sourced from relational backends—often necessitates brute-force comparison strategies with a computational complexity of $\mathcal{O}(n^2)$, where n represents the number of data elements. Furthermore, the execute-and-match approach is inherently unsuitable for GraphQL mutations, as executing them would alter the dataset, making repeated evaluations inconsistent and unreliable. Finally, the inherent nested structure of GraphQL queries introduces further complexity to the validation process. For example, a query retrieving a user's posts may redundantly re-nest the user within each post (`Users` $\rightarrow$ `Posts` $\rightarrow$ `Users`), creating structural cycles. Such nested constructs complicate direct equivalence checks, as semantically identical queries may fetch different amounts of data due to redundant or repeated field nesting, even if the high-level intent remains the same.

Therefore, there is a need for an automated, efficient, and robust GraphQL query evaluation pipeline that, for any benchmark such as [3], can reliably assess the semantic equivalence between an LLM-generated query and its corresponding ground truth query.

In this paper, we introduce Robust GraphQL Evaluation (RGEval), the first automated pipeline for verifying the validity and equivalence of LLM-generated GraphQL queries against a ground truth benchmark. Compared to state-of-the-art approaches, RGEval dramatically reduces evaluation time from hours to minutes while expanding the range of GraphQL query operations that can be reliably assessed. Additionally, it features a configurable over-fetching threshold, enabling adaptive evaluation criteria that account for acceptable levels of over-fetching in LLM-generated queries.

Contributions

The key scientific contributions of this paper are as follows:

1) A formal introduction of the GraphQL query equivalence problem for reliable benchmarking.
2) The design and implementation of the first Robust GraphQL Evaluation (RGEval) pipeline for efficient and accurate query equivalence assessment.
3) Expanded coverage of comparable GraphQL query operations, improving validation robustness.
4) An order of magnitude reduction in evaluation runtime, accelerating the process from several hours to just a few minutes.

By providing a structured approach to benchmarking GraphQL query generation, RGEval strengthens the reliability of AI-driven API interactions, enhancing the robustness of web service integration in modern enterprise architectures.

The rest of the paper is organized as follows: Section II provides a brief background on GraphQL along with the problem statement. The proposed robust GraphQL evaluation pipeline is detailed in Section III. Experimental details are presented in Section IV. Related work is reviewed in Section V, and finally, conclusions and future research directions are discussed in Section VI.

II. Preliminaries and Problem Framework

GraphQL [1] has emerged as a powerful alternative to traditional REST APIs, offering greater flexibility in data retrieval for modern web services. Its schema-driven approach enables precise queries, reducing unnecessary data transfer and improving network efficiency. Unlike REST, where fixed endpoints return predefined responses, GraphQL allows clients to dynamically specify the exact data they need, facilitating seamless integration across distributed systems.

Within service-oriented architectures, GraphQL serves as a unifying layer for aggregating data from multiple backend services. When a query is received, the server processes it through multiple stages, including parsing, validation, and execution. As part of this process, the query is converted into an Abstract Syntax Tree (AST)—a structured representation that enables efficient validation against the schema. This ensures that queries are correctly formatted and reference only valid fields before execution. By enforcing schema constraints and optimizing execution, GraphQL provides a reliable foundation for scalable web service interactions, particularly in applications requiring real-time updates and complex data compositions [6].

GraphQL's flexibility allows for multiple ways to express the same data request, leading to structural variations in queries. This poses challenges when evaluating LLM-generated queries against a ground truth. Below, we highlight key GraphQL constructs that contribute to these variations, illustrated in Figures 1.

1) **Fragments**: A fragment is a reusable selection of fields that can be included in multiple queries. This reduces redundancy and improves maintainability. Figure 1a uses `UserDetails` and `PostDetails` fragments, while Figure 1b employs `UserProfile` and `PostDetails` to modularize repeated selections. Although the structures differ, the retrieved data remains identical.
2) **Aliases**: GraphQL allows renaming fields using aliases to improve readability or avoid conflicts. For example, Figure 1c shows `user_profile`, `user_id`, and `user_name` as aliases for `users`, `id`, and `username`. While functionally identical, aliases change the query's structure, impacting direct string comparisons.
3) **Inline Fragments**: Unlike named fragments, inline fragments define type-specific selections directly within a query. They are particularly useful when querying union types or interfaces. Figure 2b demonstrates this, altering syntax while preserving intent.
4) **Directives**: Directives dynamically control query execution based on runtime conditions. Common di-

rectives like `@include(if: $condition)` and `@skip(if: $condition)` (Figure 2c) allow conditional field retrieval, leading to structural variations depending on variable values.

5) **Filters and Arguments**: GraphQL queries can filter results using arguments or explicit filter objects. Figure 2d illustrates how the same data can be retrieved using an explicit filter on `id` field.
6) **Redundant Endpoints**: In GraphQL redundant endpoints refer to multiple schema-defined fields under the type Query, that offer alternative access paths to the same underlying data. For example, consider the GraphQL schema shown in Listing 1. Here, the `users`, `usersById`, and `usersFilterList` fields can all be used to retrieve the same user information, albeit through different names or argument styles as shown in Figures 2c, 2d, and 2e. While such redundancy is valid in the schema and often intentional for developer flexibility, it introduces challenges in equivalence analysis.
7) **Cyclic Dependency**: Due to GraphQL's hierarchical and nested structure, queries—particularly those involving recursive types—can introduce cycles where entities reference each other in loops. For the schema shown in Listing 1, the `Posts` type includes a `users` field linking to the `Users` type, while the `Users` type includes a `posts` field that refers back to `Posts`. This bidirectional relationship introduces a cycle: `Users` →`Posts`→ `Users`. Although such cycles are valid within the schema and often reflect real-world relationships, they generally do not alter the core semantics of a query and can inflate structural comparisons by introducing redundant traversal paths. Identifying and eliminating these schema-induced cycles is essential for accurate and meaningful equivalence analysis.

Moreover, Figure 2e combines several of the above constructs, resulting in a new structurally distinct query that still returns the same data. These examples highlight why evaluating GraphQL query correctness using string matching is unreliable: even minor syntactic variations—such as the use of aliases, fragments, inline fragments, or different argument styles—can lead to mismatches despite functional equivalence.

The alternative, execute-and-match, also has limitations. Queries can return null or empty results due to factors such as out-of-range filter predicates, making it difficult to determine whether a mismatch stems from an incorrect query or simply the absence of matching data. Additionally, queries that fetch extra data may fail a strict match, even though they are semantically correct. Even when two queries produce identical results, those outputs may stem from specific characteristics of the dataset—such as data sparsity or redundancies in stored values—while the queries themselves remain semantically distinct.

A. Problem Statement

Given a GraphQL schema $\mathcal{S}$, a set of ground truth queries $\mathcal{Q}_{\text{GT}}$, and their corresponding LLM-generated queries $\mathcal{Q}_{\text{LLM}}$, the goal is to determine whether each LLM-generated query is functionally equivalent to its ground truth counterpart. To accommodate acceptable differences in the number of fetched fields, we introduce an overfetching budget parameter $\delta \in [0, \infty)$, representing the maximum allowable number of additional fields fetched by $\mathcal{Q}_{\text{LLM}}$ beyond those in $\mathcal{Q}_{\text{GT}}$.

The equivalence evaluation function is defined as:

$$\mathcal{E}(\mathcal{S}, \mathcal{Q}_{\text{LLM}}, \mathcal{Q}_{\text{GT}}, \delta) \rightarrow \{0, 1\}$$

where $\mathcal{E} = 1$ if $\mathcal{Q}_{\text{LLM}}$ retrieves all required fields from $\mathcal{Q}_{\text{GT}}$ and the number of additional fields fetched does not exceed δ. Otherwise, $\mathcal{E} = 0$. Notably, setting $\delta = 0$ enforces exact equivalence, ensuring that $\mathcal{Q}_{\text{LLM}}$ retrieves exactly the same fields as $\mathcal{Q}_{\text{GT}}$ without any overfetching. The objective is to develop an evaluation pipeline that systematically determines $\mathcal{E}$ for any given test dataset and δ.

To operationalize this, we design an evaluation pipeline that takes as input a test set provided in a CSV file, containing the GraphQL schema, ground truth queries, and LLM-generated queries. The pipeline systematically assesses query equivalence by leveraging structural analysis, redundancy detection, and schema-aware validation. Specifically, it produces the following outputs:

- **Detailed per-query analysis** – Identifies discrepancies such as structural mismatches, redundant selections, and errors in the generated queries.
- **Aggregate statistics** – Provides a summary of query accuracy across all test cases, offering insights into overall model performance.
- **Graph-based visualizations** – Facilitates interpretability by illustrating structural differences between the expected and generated queries.

By integrating schema-aware analysis, this pipeline offers a systematic and scalable approach to benchmarking GraphQL query generation, addressing the inherent flexibility of GraphQL while ensuring reliable equivalence evaluation.

Assumptions: This work focuses exclusively on processing GraphQL query operations and does not extend support to GraphQL mutations. The comparison process assumes that type names are not considered. Furthermore, queries involving pagination features or "like" filters are beyond the scope of this analysis and are not handled in the current implementation. These limitations will be explored in future work.

III. Our Contribution: Robust GraphQL Evaluation Pipeline

In this section, we present our proposed Robust GraphQL Evaluation Pipeline, designed to systematically assess the accuracy of LLM-generated GraphQL queries against their ground truth counterparts. The pipeline ensures schema compliance, semantic correctness, and structural equivalence through an eight-stage process, summarized in Figure 3.

The pipeline begins with a preprocessing step that extracts queries from LLM responses using regular expressions. The extracted query is then parsed into an Abstract Syntax

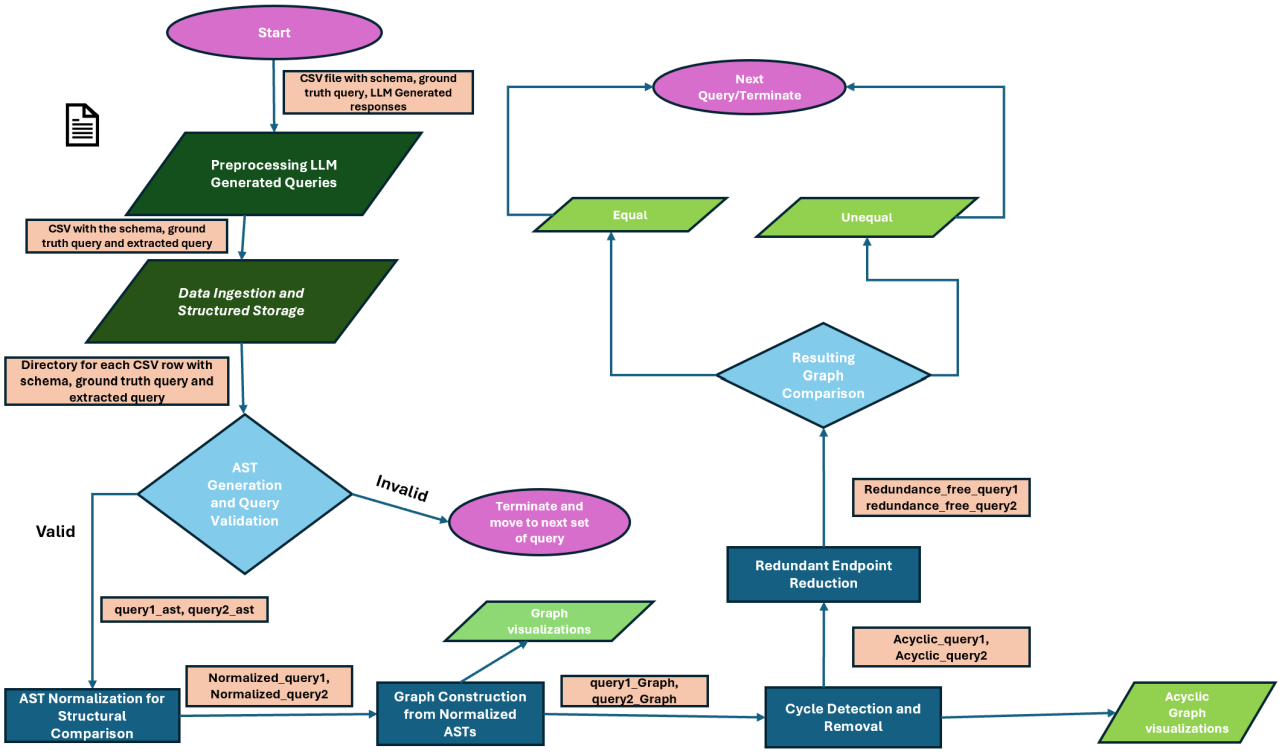


Fig. 3: RGEval Framework

Tree (AST) and validated against the provided schema. To standardize comparisons, the AST undergoes transformations such as fragment flattening, alias removal, and field type normalization.

Next, the refined AST is converted into a graph representation, where nodes correspond to fields or arguments, and edges capture relationships. Cycles in the graph are detected and removed to simplify analysis. Removing such schema-induced cycles helps isolate the intended query logic from incidental structural recursion. Additionally, redundant endpoints—fields that refer to the same return type but use alternate names (e.g., `usersById` vs `users`)—are identified through schema-based type comparison and replaced to improve alignment. Finally, transitive closure is applied to the resulting graphs of the LLM-generated and ground truth queries to assess semantic equivalence. This graph-based strategy enables data-agnostic comparison that goes beyond shallow field-level matching, accounting for nesting, aliasing, and indirect access paths.

For clarity, we refer to the *Ground Truth Query* as **Query 1** and the *LLM-Generated Query* as **Query 2** in the following sections. The subsequent subsections provide a detailed breakdown of each module. As a running example, we consider the GraphQL queries shown in Figure 2a and 2e as **Query 1** and **Query 2**, respectively.

A. *Preprocessing LLM-Generated Queries*

The evaluation pipeline takes as input a CSV file containing LLM-generated responses, ground truth queries, and the corresponding schema. A key challenge in processing LLM outputs is the lack of a standardized format, as queries are often embedded within extraneous text. To extract GraphQL queries from unstructured text, we employ a regex-based extraction method.

B. *Data Ingestion and Structured Storage*

Next, the pipeline ingests the processed CSV file containing triplets: the schema, ground truth query, and the corresponding LLM-generated output for each test case. To enable detailed, test case–level analysis and systematic evaluation, each test case is stored in a structured, timestamped directory. This storage captures all query-level diagnostics generated throughout the pipeline, including the original and normalized ASTs, intermediate graph representations, and evaluation logs. Additionally, storing queries separately enables parallel processing, enhancing performance for large-scale evaluations.

C. *AST Generation and Query Validation*

To ensure meaningful analysis, each LLM-generated query is first parsed and validated against the GraphQL schema using the `graphql-core` [7] Python library. This step verifies both syntactic correctness and schema compliance. Each query is then parsed into an Abstract Syntax Tree (AST) using the library's `parse` function, yielding a structured representation of its components. A partial AST for **Query 2** is shown in Listing 2, with the complete version available in the technical report [5]. The resulting ASTs are stored in two separate

files—query1_ast for the ground truth and query2_ast for the LLM-generated query.

```
{
  "kind": "document",
  "definitions": [
    {
      "kind":
      ...
      "selection_set": {
        "kind": "selection_set",
        "selections": []
      }
    }
  ]
}
```

Listing 2: AST snippet for GraphQL Query 2

D. AST Normalization for Structural Comparison

To ensure that query comparisons focus on logical structure rather than syntactic variations, the pipeline applies a series of transformations to standardize the ASTs of both the LLM-generated and ground truth queries. These transformations eliminate elements such as aliases and fragments that do not impact the query's semantics. Both query1_ast and query2_ast undergo this normalization process to produce a simplified representation for meaningful comparison. Algorithm 1 provides a high-level summary of this process, which is elaborated upon in the following discussion.

1) **Alias Resolution:** To ensure structural uniformity, the alias resolution step recursively traverses the AST, replacing aliased field names with their original counterparts. For example, in the case of fullName: name, the alias fullName is removed, ensuring name is used consistently across queries.

2) **Fragment Expansion:** This step identifies fragment spreads within the AST, retrieves the corresponding fragment definitions, and expands them in place. The process continues recursively for nested fragments, ensuring that the final AST contains fully expanded queries with all fragment references replaced by their explicit field definitions. Inline fragments are similarly expanded to eliminate structural discrepancies.

3) **Field Pruning:** Certain metadata fields, such as descriptions and comments, do not contribute to query execution but introduce extraneous differences. These fields are systematically removed to ensure that the final ASTs capture only the essential query structure.

By applying these transformations, the pipeline ensures that the comparison focuses solely on query logic, thereby enhancing the robustness of downstream validation and equivalence assessment. The output of this step is saved as normalized_query1_ast and normalized_query2_ast, containing the normalized ASTs used for structural comparison.

E. Graph Construction from Normalized ASTs

Transforming customized ASTs into graph representations is essential for structural comparisons of GraphQL queries. Unlike direct string-based comparisons, which are affected by superficial syntactic variations (e.g. whitespace, or field ordering differences), graph-based representations capture the underlying logical structure of queries. This ensures that equivalence assessments are based on semantic content rather than textual formatting.

Algorithm 1 *AST Normlization for structural comparison*

Require: Raw ASTs of Query1 and Query2
Ensure: Normalized ASTs
1: **Function** normalize_asts(query1_ast, query2_ast):
1) **Alias Resolution:**
a) $query1\_ast \leftarrow remove\_aliases(query1\_ast)$
b) $query2\_ast \leftarrow remove\_aliases(query2\_ast)$
2) **Fragment Expansion:**
a) $query1\_ast \leftarrow expand\_fragments(query1\_ast)$
b) $query2\_ast \leftarrow expand\_fragments(query2\_ast)$
3) **Field Pruning:**
a) $query1\_ast \leftarrow prune\_fields(query1\_ast)$
b) $query2\_ast \leftarrow prune\_fields(query2\_ast)$
4) **return** $(query1\_ast, query2\_ast)$

Graph-based comparisons leverage efficient algorithms, such as adjacency matrices and transitive closure, to facilitate accurate and scalable evaluations, even for complex queries. The process begins by recursively traversing the AST. For each node, relevant properties are extracted:

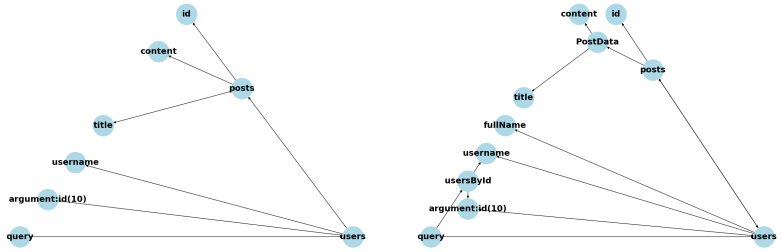
- **Field Nodes**: Each field in the query is represented as a node in the graph.
- **Argument Nodes**: If a field contains arguments, each argument is treated as a separate node, with an edge linking the field node to its argument nodes.
- **Nested Fields**: If a node contains child fields (i.e., selections within a field), these are recursively processed as child nodes with edges linking them to the parent field.

This recursive traversal ensures that all relationships between fields, arguments, and nested selections are explicitly captured. Once the graph is constructed, it is serialized into a standard format (e.g., JSON) for further processing and comparison. The resulting graphs are stored as graph1 and graph2. Figure 4 presents the initial query graphs for both Query 1 and Query 2, while Figure 5a illustrates the graph representation of Query 2 after normalization.

F. Cycle Detection and Removal

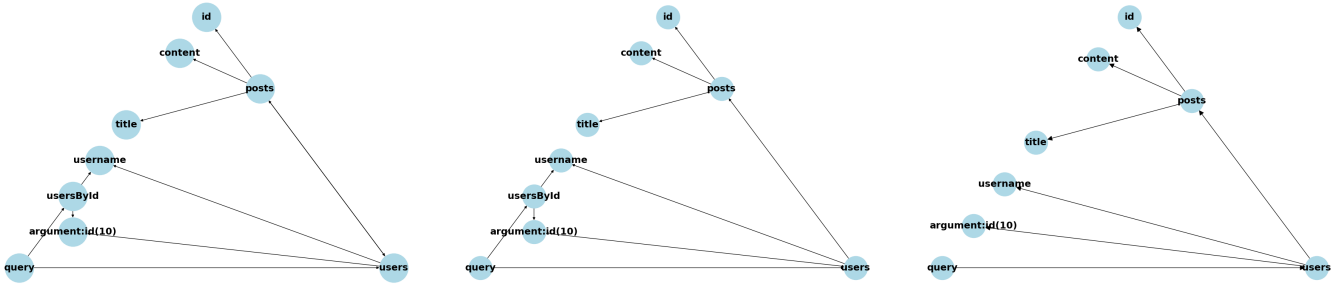
A key challenge in GraphQL query analysis is handling cyclic structures. While cycles are valid under the schema, they typically arise from recursive schema relations and reflect structural topology rather than user intent. These cycles do not contribute to the core query semantics but can distort structural equivalence comparisons by introducing redundant relationships and inflating graph complexity. To ensure meaningful analysis, the pipeline detects and removes these schema-induced cycles, improving the precision of downstream equivalence assessments.

Algorithm 2 provides a high-level overview of this stage. The cycle removal process consists of the following steps:



(a) Query 1 (b) Query 2

Fig. 4: Graph Visualization of the Input Query 1 and Query 2



(a) Query 2 after normalization (b) Query 2 after cycle removal (c) Query 2 after redundancy removal

Fig. 5: Visualization of processed LLM generated GraphQL query (Query 2)

1) **Graph Construction**: An adjacency list is generated from the query graph's edges.
2) **Cycle Detection**: A depth-first search (DFS) algorithm is used to identify cycles.
3) **Cycle Removal**: Identified cycles are pruned based on predefined schema rules, ensuring that only essential relationships remain. The schema rules are provided as a list of edge pairs that can be safely removed if they form part of a structural loop.

After processing, the cleaned graphs are stored as `acyclicGraph1` and `acyclicGraph2`. The resulting acyclic structure of **Query 2** is visualized in Figure 5b.

Algorithm 2 ***Cycle Detection and Removal***

Require: Graph representation of queries
Ensure: Acyclic graphs
1: **Function** remove_cycles(graph1, graph2):
1) Construct adjacency list from input graphs.
2) Detect cycles using depth-first search (DFS):
   a) $cycles_1 \leftarrow detect\_cycles(graph1)$
   b) $cycles_2 \leftarrow detect\_cycles(graph2)$
3) Remove redundant cycles based on schema rules:
   a) $acyclicGraph1 \leftarrow prune\_cycles(graph1, cycles_1)$
   b) $acyclicGraph2 \leftarrow prune\_cycles(graph2, cycles_2)$
4) Save cleaned graphs:
   a) save_graph(acyclicGraph1, "acyclicGraph1.json")
   b) save_graph(acyclicGraph2, "acyclicGraph2.json")
5) **return** (acyclicGraph1, acyclicGraph2)

G. *Redundant Endpoint Reduction*

LLM-generated queries might choose any of the interchangeable endpoints to access the same data, leading to semantically correct but structurally different queries. To ensure accurate equivalence analysis, the pipeline identifies and standardizes such redundant endpoints.

This process begins with schema analysis to detect interchangeable fields under the `Query` type that retrieve identical data (e.g., `users`, `usersById`, `usersFilterList`). One of these is selected as the canonical form—typically the primary or most general-purpose field.

Next, the cleaned, acyclic query graphs of both the ground truth and LLM-generated queries are traversed to locate and replace redundant endpoints with their canonical equivalents. For example, `usersById(id:1)` is substituted with `users(id:1)` to enforce structural alignment. By unifying these access paths, the pipeline avoids false mismatches caused by superficial differences in naming or argument structure, improving robustness in semantic comparisons.

Algorithm 3 summarizes this endpoint reduction process. A visualization of the optimized query graph after redundant endpoint removal is shown in Figure 5c.

H. *Resulting Graph Comparison*

The final stage of the pipeline involves comparing the optimized query graphs (`graph1` and `graph2`) to determine whether they are structurally equivalent. Despite syntactic differences, two queries can be considered equivalent if they

Algorithm 3 ***Redundant Endpoint Reduction***

Require: Cleaned, acyclic query graphs $graph1$ and $graph2$
Ensure: Optimized query graph $graph2$
1: **Function** optimize_query(graph1, graph2):
1) Identify equivalent endpoints:
a) $equivalent\_fields \leftarrow find\_equiv\_endpoints(graph1, graph2)$
2) **if** $equivalent\_fields \neq \emptyset$ **then**
a) $graph2 \leftarrow rep\_fields(graph2, equiv\_fields)$
3) Save and return optimized query graph:
a) save_graph(graph2, "optimized_query2.json")
b) **return** graph2

request the same fields and maintain the same logical relationships between entities. This comparison is performed by analyzing the transitive closures of the graphs, which capture all field reachability paths from the query root, abstracting over nesting or ordering differences. To ensure a rigorous comparison, the process first converts each query graph into an adjacency matrix representation. The adjacency matrix encodes direct connections between nodes, forming the basis for computing the transitive closure. The transitive closure of a graph extends this representation to include all possible indirect paths between nodes, revealing structural similarities or differences.

The Floyd-Warshall algorithm is employed to compute the transitive closure. This algorithm iteratively checks whether a path exists between two nodes, progressively updating the adjacency matrix to reflect all possible connections. The algorithm is applied to both query graphs while ensuring that only paths originating from the designated root node are considered, thereby preserving query-specific semantics.

Once the transitive closures for both query graphs are obtained, a direct element-wise comparison is performed. If the resulting matrices are identical, the queries are deemed structurally equivalent. However, in practice, LLM-generated queries $\mathcal{Q}_{\text{LLM}}$ may retrieve additional fields beyond those present in the ground truth query $\mathcal{Q}_{\text{GT}}$. To account for such permissible variations, we introduce an overfetching budget parameter δ.

Given two transitive closure matrices T_{LLM} and T_{GT}, we define the difference matrix:

$$\Delta_T = T_{\text{LLM}} - T_{\text{GT}}$$

where each element $\Delta_T(i, j)$ represents the additional relationships introduced by $\mathcal{Q}_{\text{LLM}}$ that are absent in $\mathcal{Q}_{\text{GT}}$.

The equivalence function $\mathcal{E}$ evaluates whether $\mathcal{Q}_{\text{LLM}}$ remains within the permissible overfetching threshold:

$$\mathcal{E}(\mathcal{S}, \mathcal{Q}_{\text{LLM}}, \mathcal{Q}_{\text{GT}}, \delta) = \begin{cases} 1, & \text{if } ||\Delta_T||_1 \leq \delta \\ 0, & \text{otherwise} \end{cases}$$

where $||\Delta_T||_1$ denotes the sum of all additional relationships in $\mathcal{Q}_{\text{LLM}}$. If $\delta = 0$, strict structural equivalence is enforced, ensuring that $\mathcal{Q}_{\text{LLM}}$ retrieves exactly the same fields as $\mathcal{Q}_{\text{GT}}$. By allowing a nonzero δ, we introduce flexibility in the comparison process while preserving the core query intent.

Algorithm 4 formally outlines the steps involved in the comparison process.

Algorithm 4 ***Compare Query Graphs Using Transitive Closure***

Require: Graph representations of `query1` and redundancy-free `query2`, overfetching budget δ
Ensure: Boolean result (`True` if equivalent, `False` otherwise)
1: **Function** compare_graphs(graph1, graph2, δ):
1) Compute adjacency matrices:
a) $adj\_matrix1 \leftarrow create\_adjacency\_matrix(graph1)$
b) $adj\_matrix2 \leftarrow create\_adjacency\_matrix(graph2)$
2) Compute transitive closures:
a) $transitive\_closure1 \leftarrow compute\_transitive\_closure(adj\_matrix1)$
b) $transitive\_closure2 \leftarrow compute\_transitive\_closure(adj\_matrix2)$
3) Compute the difference matrix:
a) $\Delta_T \leftarrow transitive\_closure1 - transitive\_closure2$
4) Compute overfetching count:
a) $overfetching \leftarrow ||\Delta_T||_1$
5) Compare transitive closures with δ threshold:
a) **if** $overfetching \leq \delta$ **then**
i) **return** True
b) **else**
i) **return** False

IV. Experiments

A. Evaluation on Schema-Based Queries

We evaluated the RGEval pipeline on a system running Windows 11 Home (version 10.0.26100 Build 26100) with an 11th Gen Intel Core i5-1135G7 processor and 8 GB of RAM.

For this experiment, we manually designed three relational database schemas—two for PostgreSQL 17 and one for MySQL 9.1—and utilized StepZen CLI (version 0.48.0) to generate their corresponding GraphQL schemas. To assess RGEval's capability in recognizing semantic equivalence, we **manually** constructed a set of **52 query pairs**, each consisting of a baseline query and its logically equivalent counterpart. These queries incorporated a range of GraphQL constructs, including **fragments, inline fragments, aliases, variables, nested fragments, redundant fields**, and **directives** such as `@include` and `@skip`, either in isolation or in combination.

The RGEval pipeline correctly classified all query pairs as equivalent, demonstrating its robustness in detecting semantic equivalence across structurally diverse schemas. This evaluation underscores the system's ability to generalize across varying schema designs while maintaining accuracy in query comparison.

B. Comparison with Execute-and-Match

Beyond the schema-based evaluation, we generated queries for **1164 test cases** from the EMNLP study [3] using

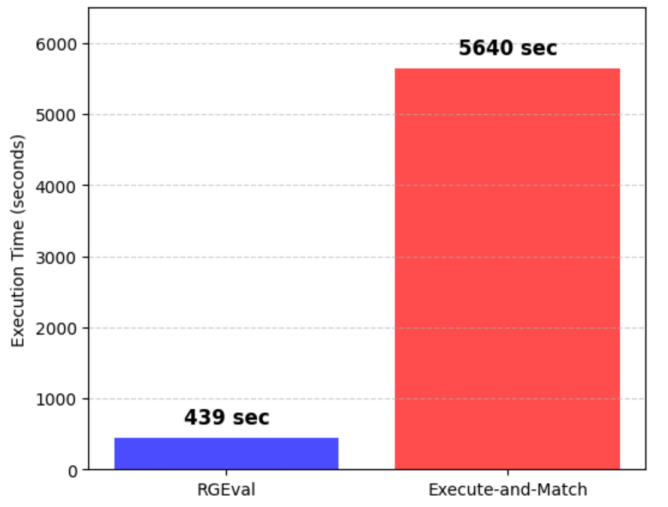


Fig. 6: Runtime Comparison: RGEval vs. Execute-and-Match

`llama-3-405b-instruct`. These queries were then processed through our pipeline. Among them, 995 queries were successfully validated, with 761 classified as equivalent and 234 marked as unequal. For the remaining 169 cases, the generated queries did not conform to the schema. The total execution time for this process was **7 minutes and 19 seconds**.

In contrast, the Execute-and-Match approach required approximately **94 minutes** (**1 hour and 34 minutes**) to process the same test cases. This demonstrates a substantial efficiency gain—RGEval reduces the execution time from *hours* taken by the existing pipeline to just *minutes*, making it a far more practical and scalable solution for large-scale query evaluations. The performance comparison is visualized in Figure 6, highlighting the significant speedup achieved by RGEval.

V. Related Work

GraphQL has garnered considerable interest in both academic and industrial domains. While prior efforts have explored leveraging LLMs for GraphQL query generation [8]–[11] and enhancing their query generation capabilities [3], [4], to the best of our knowledge, there is no formal study focused on determining the equivalence of GraphQL queries. Therefore, in this section, we provide a brief overview of relevant literature on GraphQL.

Comparative studies between REST and GraphQL APIs highlight several advantages of GraphQL. For example, [12] demonstrates that GraphQL reduces client-server interactions and optimizes JSON payload sizes, while [13] shows that GraphQL queries offer easier implementation. Additional works, such as [14], [15], further examine GraphQL's benefits over REST.

Beyond these advantages, recent research has explored GraphQL query testing [16] and schema mapping investigations [17]. GraphQL is also widely adopted in real-world applications [18] and industry use cases [19], with its potential for seamless data access and integration across heterogeneous sources highlighted in [20].

However, as demonstrated in [3], state-of-the-art LLMs struggle with GraphQL query generation due to limited exposure to GraphQL data during training. Recent advancements in specialized prompt engineering and schema-aware in-context learning [4] have improved LLM-driven query generation. Additionally, dedicated datasets [3] enable finetuning and benchmarking, a crucial step in ensuring trust in LLM deployment for production environments.

From an industry standpoint, GraphQL adoption is projected to grow significantly. A recent Gartner report predicts that by 2027, over 60% of enterprises will deploy GraphQL in production, a sharp rise from less than 30% in 2024 [2].

VI. Conclusion and Future Work

In this work, we present the first comprehensive pipeline for evaluating the semantic equivalence of LLM-generated GraphQL queries against ground truth references. The pipeline integrates multiple rigorous mechanisms to address the challenges posed by structurally divergent but semantically equivalent queries—differences that may arise due to GraphQL-specific constructs such as fragments, aliases, redundant endpoints, and cyclic dependencies. Our evaluation framework performs in-depth analysis, including structural normalization, redundancy resolution, and graph-based representations, to deliver both fine-grained per-query assessments and aggregate performance insights. The results underscore the difficulty of ensuring that LLM-generated queries faithfully capture the intended semantics, while also revealing the limitations of traditional string-matching and execution-based evaluation approaches.

While our current framework offers a structured and scalable evaluation methodology, several avenues remain open for future exploration. First, our study focuses exclusively on query operations, and extending the pipeline to support GraphQL mutations would enhance its applicability. Additionally, handling more complex query constructs, such as those involving pagination and "like" filters, remains an important direction for future work.

With this study, we aim to drive further research into reliable GraphQL query evaluation, particularly for AI-driven web services. Ensuring the correctness of LLM-generated queries is crucial for seamless API integration in modern architectures. By enhancing automated query generation and validation, our work contributes to building more trustworthy and efficient AI-powered web service interactions.

***Index Terms*—GraphQL Evaluation, LLM Query Generation, Generative AI in Services**

References

[1] (2025) Graphql. [Online]. Available: https://graphql.org/

[2] (2024) Gartner report - graphql. [Online]. Available: https://www.apollographql.com/resources/gartner-when-to-use-graphql-to-accelerate-api-delivery

[3] M. Kesarwani, S. Ghosh, N. Gupta, S. Chakraborty, R. Sindhgatta, S. Mehta, C. Eberhardt, and D. Debrunner, "Graphql query generation: A large training and benchmarking dataset," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, 2024, pp. 1595–1607.

[4] N. Gupta, M. Kesarwani, S. Ghosh, S. Mehta, C. Eberhardt, and D. Debrunner, "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)*, Apr. 2025, pp. 1009–1015.
[5] (2025) Techreport: Benchmarking graphql query generation for web services. [Online]. Available: https://github.com/vsonthalia/ICWS2025
[6] (2015) Graphql: A data query language. [Online]. Available: https://engineering.fb.com/2015/09/14/core-infra/graphql-a-data-query-language/
[7] (2025) Graphql-core. [Online]. Available: https://pypi.org/project/graphql-core/
[8] Y. V. Levin. (2023) A developer's journey to the ai and graphql galaxy. [Online]. Available: https://medium.com/@yonatanvlevin/a-developers-journey-to-the-ai-and-graphql-galaxy-3e8e7fd41928
[9] (2023) Graphql explorer. [Online]. Available: https://github.com/geobde/graphqlexplorer
[10] (2023) Gqlpt. [Online]. Available: https://github.com/rocket-connect/gqlpt
[11] (2023) Weaviate gorilla part 1 graphql apis. [Online]. Available: https://weaviate.io/blog/weaviate-gorilla-part-1
[12] G. Brito, T. Mombach, and M. T. Valente, "Migrating to graphql: A practical assessment," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 140–150.
[13] G. Brito and M. T. Valente, "Rest vs graphql: A controlled experiment," in *2020 IEEE International Conference on Software Architecture (ICSA)*, 2020, pp. 81–91.
[14] M. Seabra, M. F. Nazário, and G. Pinto, "Rest or graphql? a performance comparative study," in *Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse*, 2019, pp. 123–132.
[15] M. Mikuła and M. Dzieńkowski, "Comparison of rest and graphql web technology performance," *Journal of Computer Sciences Institute*, vol. 16, pp. 309–316, 2020.
[16] A. Belhadi, M. Zhang, and A. Arcuri, "Random testing and evolutionary testing for fuzzing graphql apis," *ACM Transactions on the Web*, vol. 18, no. 1, pp. 1–41, 2024.
[17] A. Quiña-Mera, P. Fernandez, J. M. García, and A. Ruiz-Cortés, "Graphql: a systematic mapping study," *ACM Computing Surveys*, vol. 55, no. 10, pp. 1–35, 2023.
[18] (2024) Companies using graphql. [Online]. Available: https://theirstack.com/en/technology/graphql
[19] (2024) Stepzen. [Online]. Available: https://stepzen.com
[20] H. Li, O. Hartig, R. Armiento, and P. Lambrix, "Ontology-based graphql server generation for data access and data integration," *Semantic Web*, no. Preprint, pp. 1–37, 2024.

APPENDIX

Here, we present a sample prompt used with the LLMs for GraphQL query operation generation.

Prompt: You are an expert in GraphQL. Given the following GraphQL schema and a natural language (NL) request, generate the corresponding GraphQL query.

GraphQL Schema:

```
type Posts {
  content: String!
  id: Int!
  title: String
  user_id: Int
  users: Users
}

type Users {
  id: Int!
  posts: [Posts]
}

input IntFilter {
  eq: Int
  ne: Int
  gt: Int
  lt: Int
}

input StringFilter {
  eq: String
  like: String
}

input UsersFilter {
  id: IntFilter
  username: StringFilter
}

type Query {
  posts(id: Int!): Posts

  users(id: Int!): Users

  usersFilterList(filter: UsersFilter): [Users]

  usersById(id: Int!): Users
}
```

Natural Language Request: Fetch the profile of the user having ID 1, showing their username and all their blog posts. Each post entry should display its unique identifier, headline, and main text body.

Expected GraphQL Query Format:

- Use appropriate queries from the schema.
- Structure the query correctly.
- Ensure field selection follows the request.
- Format the query to use Fragments and alias to improve readability.

Generated GraphQL Query: