BenGQL: An Extensible Benchmarking Framework for Automated GraphQL Testing

Abenezer Angamo*
Independent Researcher
Addis Ababa, Ethiopia
abenezer.angamo@gmail.com

Marcello Maugeri*
University of Catania
Catania, Italy
marcello.maugeri@phd.unict.it

***Abstract*—GraphQL APIs provide a unified endpoint for retrieving and uploading data in a web application. Due to its efficient data-fetching strategy, which allows for the retrieval of only the required data, GraphQL is gaining popularity. Its software nature necessitates robust testing, both functional and non-functional. As automated testing tools, including load testers and fuzzers, are developed to assess GraphQL APIs, they lack a common set of case studies for rigorous evaluation and comparison.**

To address this gap, we present BENGQL, a benchmarking framework containing 23 representative open-source GraphQL server applications, spanning different underlying engines and schema complexities. BENGQL provides an extensible infrastructure for running testing tools against these case studies, enabling developers and researchers to: (i) execute testing tools against the same case studies, (ii) analyse and compare results using custom analysis modules, and (iii) extend the benchmark with new case studies, tools, or metrics.

The ultimate goal of BENGQL is to foster more rigorous, reproducible research in automated GraphQL testing by providing both the case studies and the infrastructure for running experiments. As a consequence, we release the source code at https://github.com/marcellomaugeri/BenGQL, inviting other researchers to contribute. A video demonstration is also available at https://youtu.be/wZ06Xxa_Koo.

***Index Terms*—GraphQL, Benchmark, Automated Testing**

I. INTRODUCTION

GraphQL exposes a single endpoint through which clients post a JSON-like *query* specifying the exact data graph they require. The server validates the request against a *schema* – a catalogue of object types, fields, and relationships – and returns a response. Unlike traditional REST APIs, a GraphQL API centralises interaction, enabling clients to avoid over- and under-fetching by composing arbitrarily deep field selections. While this expressiveness streamlines development, it also increases the *attack surface* spanning schema validation, resolver logic, and the underlying data stores, making rigorous automated testing essential.

Automated tools have already started to tame this surface. EVOMASTER [1], [2] employs both white-box and black-box search-based fuzzing to uncover functional faults. GRAPHQLER [3], [4] adopts a black-box approach by inferring a dependency graph during testing, thereby exposing authentication and authorisation flaws, access-control bypasses, and resource-misuse sequences that lead to Denial-of-Service (DoS) attacks. SCHEMATHESIS [5] employs property-based testing to derive fuzzers from GraphQL schemas, detecting schema violations, validation bypasses, and information disclosure vulnerabilities. By contrast, WENDIGO [6] leverages Deep Reinforcement Learning to identify time-consuming queries that could lead to a DoS attack. ARTILLERY [7] provides semi-automated load testing, replaying user-defined traffic patterns to stress endpoints under realistic workloads.

Given the increasing adoption, which suggests that more than 60% by 2027 of enterprises will use GraphQL in production [8], it is fair to assume that new and more advanced automated testing tools will be developed to assess different properties. However, without a well-established benchmark, researchers cannot objectively compare tool effectiveness, practitioners cannot make informed tool selections, and empirical results remain incomparable across studies, hindering reproducible research and objective progress assessment.

This issue has been partially addressed in similar domains by FUZZBENCH [9] for coverage-guided fuzzing and PROFUZZBENCH [10] for stateful protocols. In the web API field, WFD [11], previously known as EMB [12], provides a corpus of web applications for testing research. However, WFD focuses primarily on the execution of JVM-based applications (written in Java and Kotlin), and does not provide infrastructure to orchestrate experiments across multiple testing tools. Moreover, it includes only five GraphQL case studies.

To address this gap, we introduce BENGQL, an open-source and extensible benchmark framework. BENGQL currently packages *23* representative open-source GraphQL server applications, spanning multiple engines, authentication models, and schema complexities. The framework enables the following: (i) integration of case studies in an isolated, containerised environment,(ii) execution of experiments using automated testing tools, and (iii) analysis of raw results using customisable analysis scripts.

By providing a unified benchmarking system of GraphQL applications, BENGQL facilitates rigorous and replicable experimentation in the rapidly evolving GraphQL ecosystem.

II. DESIGN

Figure 1 depicts the BENGQL workflow. On the left are the two input sets: *Case Studies*, namely the GraphQL server

*Both authors contributed equally.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00376

applications, and *Tools*, the automated testing tools to be evaluated and/or compared.

The *Experiment Driver* pairs every chosen case study with every chosen tool and, for each pair, launches a *Trial*. A trial brings up the case study's containerised stack (typically a web server plus backing database) and launches a dedicated container running the tool, which targets the case study's GraphQL endpoint. Trials in the same experiment run in parallel, bounded by a user-defined concurrency limit that prevents resource exhaustion.

When all trials are complete, each deposits its *Raw Results* in a common results directory, ready to be analysed. Then, the *Analysis Module* transforms these data into *Processed Results* ready for visualisation and/or statistical comparison.

The *Experiment Driver* comprises around 400 lines of POSIX-compliant *Bash* and runs unmodified on both Linux and macOS. The next subsections outline the implementation details of the remaining components.

A. Case Study Integration

To ensure portability, isolation, and reproducibility, each case study comes with a *docker-compose.yml* file that instantiates the entire run-time stack. A GraphQL application typically requires one or more databases; thus, the compose file generally initiates one or more data-store containers in conjunction with the server.

To add a new case study, researchers must provide a *Dockerfile* that follows some practical rules: (i) inherits from a slim base image to minimise disk usage; (ii) download the case-study – typically via *git* – to a specific commit or release to ensure repeatable experiments; and (iii) install only the dependencies required to run the server.

When authentication is required, an optional `auth.sh` script performs the login, prints the resulting HTTP header, and forwards it to the respective tool. Lastly, every case study must include a lightweight *healthcheck*: a minimal GraphQL query that verifies the endpoint remains operational throughout the trial.

B. Tool Integration

To integrate a new tool, researchers must containerise it as a service in its own *docker-compose.yml*, which may include any required dependencies (e.g., databases). The only mandatory requirement is that the tool container must mount the *results* directory as a volume to write outputs. The tool connects to case studies exclusively via HTTP, using a URL provided at runtime.

To accommodate different tool interfaces, tool integrators create a small Bash connector script that specifies the tool's launch command using four placeholder tokens:

- *TARGET_URL*: the GraphQL endpoint URL with dynamically allocated port;
- *OUTPUT_DIR_PATH*: where the tool writes results (pattern: *results/$exp_name/$tool/$case_study*);
- *AUTH_HEADER*: authentication header acquired at runtime from `auth.sh`, if needed;
- *TIME_BUDGET*: execution time limit for tools that do not self-terminate or when the user wants to constrain execution time.

The *Experiment Driver* interpolates these token templates before execution, ensuring each tool receives the correct arguments for each trial.

C. Analysis Pipeline

Once every trial has finished, i.e. after an experiment has completed, the raw outputs will be stored in a predictable directory tree that is rooted at results/.

Since the *Analysis Module* is loosely coupled to the framework, researchers can extend it by simply adding post-processing scripts to the *analysis* folder. These scripts can be written in any language (Python, Bash, etc.) as they are executed directly by the researcher.

Each script takes the experiment directory as input. This is used to inspect the internal folders and files, extract the relevant data and emit user-defined metrics. For example, the effectiveness of an API fuzzer may be evaluated based on the number of unique endpoints it covers, whereas the performance of a load tester may be assessed based on mean latency.

III. Benchmark Selection

To populate BenGQL with realistic yet self-contained case studies, we began by examining the *36* GraphQL APIs used in Asma *et al.*'s study on white- and black-box testing [1], which includes the case studies later collected in WFD.

Unfortunately, several candidates proved unsuitable: some expose only a public cloud endpoint with no local deployment artefact, while others depend on third-party services that violate the isolation we require. Consequently, we refined the pool by applying four filter criteria:

- **C1** the application must run locally, either from source or a public Docker image;
- **C2** it must execute without contacting external web services or APIs – a local database is admissible because it remains within the compose stack;
- **C3** the repository should show evidence of maintenance within the past three years;
- **C4** the GraphQL schema must be non-trivial, excluding toy schemas with only a handful of fields.

Applying **C1–C4** reduced the original set to *13* projects. To broaden language and engine coverage, we drew the top 25 repositories under GitHub's *graphql* topic[1], re-applied the same filter criteria, and retained an additional *10* projects, yielding *23* case studies overall.

Figure 2 visualises how many candidates outlived each criterion.

Table I details the final *23* case studies, listing engine, implementation language, and code size. Overall, the set covers six languages – TypeScript dominates (10 projects), followed by Java (4), JavaScript (4), Python (2), Kotlin (2), and Ruby

[1]https://github.com/topics/graphql

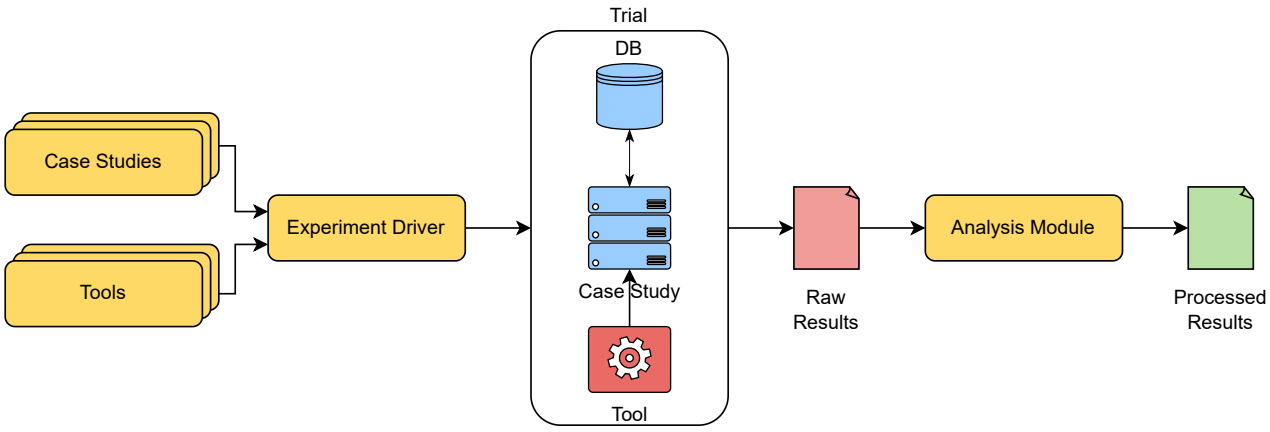

Fig. 1. BENGQL workflow.

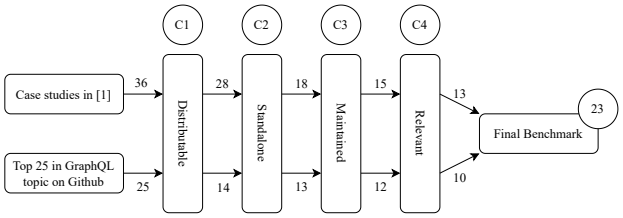

Fig. 2. Visualisation of the selection process.

(1) – and fifteen distinct GraphQL engines or frameworks, inspected manually from the codebase. The size of codebases ranges from a few hundred lines to over ten million lines. In total, the benchmark aggregates approximately *16,3* million lines of code.

IV. DEMONSTRATION

To showcase an end-to-end workflow, the first public release of BENGQL bundles three reference tools: APOLLO ROVER[2] for schema retrieval, and two black-box fuzzers (EVOMASTER and GRAPHQLER). Trials are launched via the `run_experiments.sh` script – i.e., the *Experiment Driver* – which expands the argument-template tokens defined in Section II-B.

To demonstrate BENGQL's evaluation capabilities, we compare EVOMASTER and GRAPHQLER using the coverage metrics from the GRAPHQLER pre-print [3] on the four case studies present in both our benchmark and their evaluation. Following the GRAPHQLER evaluation methodology, we measure both *positive coverage* and *negative coverage*:

- *Positive coverage (+)* represents the percentage of endpoints that successfully return data without errors when queried with valid inputs.
- *Negative coverage (-)* represents the percentage of endpoints that produce errors despite receiving valid inputs, indicating potential bugs or implementation issues.

[2]https://www.apollographql.com/docs/rover

EVOMASTER is executed in *black-box* mode with purely random request generation (no source-code instrumentation or evolutionary algorithms). The snippet below demonstrates how to run this comparison experiment using the *Experiment Driver*:

```
./run_experiments.sh \
  --exp_name exp \
  --tools EvoMaster,GraphQLer \
  --case_studies countries,ehri-rest,\
    rick-and-morty-api,react-finland \
  --time_budget 3600
```

The *Experiment Driver* automatically fills in the endpoint URL, output directory, and time budget for each tool. To analyse the results, we developed a simple Bash script as an analysis module to retrieve data from the respective output directories. Table II presents the positive and negative coverage results for both tools across the four case studies.

V. DISCUSSION AND CONCLUSION

We introduced BENGQL, an open-source, extensible benchmark for evaluating automated testing techniques on GraphQL applications. The first release bundles 23 self-contained case studies that cut across engines, frameworks, and schema sizes. Docker Compose plus Bash make the entire benchmark portable. In fact, although case studies run on the *x64* architecture, BENGQL still runs on Apple Silicon via QEMU seamlessly.

However, even with six programming languages represented, our corpus cannot mirror every industrial GraphQL deployment. We mitigate this gap through several strategies: (i) the case-study integration process is lightweight and documented, lowering the barrier for external contributions; (ii) we provide examples that serve as starting points for new integrations; and (iii) we include continuous integration pipelines to validate contributed case studies automatically. The project welcomes community contributions under the *Apache-2.0 li-*

TABLE I
OVERVIEW OF BENGQL CASE STUDIES.
TOP BLOCK: PROJECTS FROM [1], BOTTOM BLOCK: ADDITIONS FROM GITHUB'S TOP-25 *graphql* REPOSITORIES.

Case Study	Engine/Framework	Language	LOC	URL
CatalysisHubBackend	Flask-GraphQL	Python	26 k	https://github.com/SUNCAT-Center/CatalysisHubBackend
countries	Yoga	Typescript	865	https://github.com/trevorblades/countries
ehri-rest	GraphQL Java	Java	546 k	https://github.com/EHRI/ehri-rest
fruits-api	Apollo	Javascript	13 k	https://github.com/Franqsanz/fruits-api
Gitlab-CE	GraphQL Ruby	Ruby	10 m	https://gitlab.com/gitlab-org/gitlab
sierra	GraphQL Java	Java	1 m	https://github.com/hivdb/sierra
rick-and-morty-api	apollo-server-express	Javascript	8 k	https://github.com/afuh/rick-and-morty-api
react-ecommerce	Nestjs-GraphQL	Typescript	5 k	https://github.com/react-shop/react-ecommerce
graphql-ncs	graphql-java-tools	Kotlin	596	https://github.com/WebFuzzing/EMB/
graphql-scs	graphql-java-tools	Kotlin	582	https://github.com/WebFuzzing/EMB/
spring-petclinic-graphql	Spring for GraphQL	Java	40 k	https://github.com/spring-petclinic/spring-petclinic-graphql
ReactFinland	express-graphql	Typescript	44 k	https://github.com/ReactFinland/graphql-api
timbuctoo	GraphQL-Java	Java	86 k	https://github.com/HuygensING/timbuctoo
Gatsby	graphql-js	Javascript	919 k	https://github.com/gatsbyjs/gatsby
payload	graphql-js	Typescript	689 k	https://github.com/payloadcms/payload
twenty	nestjs/apollo	Typescript	705 k	https://github.com/twentyhq/twenty/
directus	graphql-ws	Typescript	435 k	https://github.com/directus/directus
hey	graphql-codegen	Typescript	52 k	https://github.com/heyverse/hey
rxdb	express-graphql	Typescript	185 k	https://github.com/pubkey/rxdb
saleor	Graphene	Python	647 k	https://github.com/saleor/saleor
parse-server	Apollo	Javascript	166 k	https://github.com/parse-community/parse-server
redwoodjs-graphql	Yoga	Typescript	386 k	https://github.com/redwoodjs/graphql
amplication	nestjs/apollo	Typescript	413 k	https://github.com/amplication/amplication

TABLE II
POSITIVE AND NEGATIVE COVERAGE COMPARISON BETWEEN GRAPHQLER AND EVOMASTER AFTER 1 HOUR ON FOUR COMMON CASE STUDIES. (+) DENOTES POSITIVE COVERAGE, (-) DENOTES NEGATIVE COVERAGE.

Case Study	GraphQLer (+)	GraphQLer (-)	EvoMaster (+)	EvoMaster (-)
countries	6/6 (100.00%)	3/6 (50.00%)	6/6 (100.00%)	6/6 (100.00%)
ehri-rest	19/19 (100.00%)	9/19 (47.37%)	19/19 (100.00%)	0/19 (0.00%)
react-finland	13/13 (100.00%)	6/13 (46.15%)	7/13 (53.85%)	9/13 (69.23%)
rick-and-morty-api	6/9 (66.67%)	6/9 (66.67%)	9/9 (100.00%)	0/9 (0.00%)
TOTAL	**44/47 (93.62%)**	**24/47 (51.06%)**	**41/47 (87.23%)**	**15/47 (31.91%)**

cense, with active maintainer support for integrating new case studies, tools, and analysis modules.

Ultimately, we believe that BENGQL will accelerate research into GraphQL testing and provide tool developers with a fair and reproducible benchmark.

ACKNOWLEDGEMENT

We want to thank the *GARR Consortium* for providing us with the test infrastructure. This work acknowledges support from: Italian PNRR 2022 SERICS Spoke 6, Task 1.2, Project "SCAI — Supply Chain Attack Avoidance".

REFERENCES

[1] A. Belhadi, M. Zhang, and A. Arcuri, "Random testing and evolutionary testing for fuzzing graphql apis," *ACM Transactions on the Web*, 2024.

[2] A. Arcuri, M. Zhang, S. Seran, J. P. Galeotti, A. Golmohammadi, O. Duman, A. Aldasoro, and H. Ghianni, "Tool report: Evomaster—black and white box search-based fuzzing for rest, graphql and rpc apis," *Automated Software Engineering*, 2025.

[3] O. Tsai, H. Zhu, J. Xiao, J. Li, N. Cheung, and L. Huang, "GraphQLer," Nov. 2024. [Online]. Available: https://github.com/omar2535/GraphQLer

[4] O. Tsai, J. Li, T. T. Cheung, L. Huang, H. Zhu, J. Xiao, I. Sharafaldin, and M. A. Tayebi, "Graphqler: Enhancing graphql security with context-aware api testing," 2025. [Online]. Available: https://arxiv.org/abs/2504.13358

[5] Z. Hatfield-Dodds and D. Dygalo, "Deriving semantics-aware fuzzers from web api schemas," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022.

[6] S. McFadden, M. Maugeri, C. Hicks, V. Mavroudis, and F. Pierazzi, "Wendigo: Deep reinforcement learning for denial-of-service query discovery in graphql," in *2024 IEEE Security and Privacy Workshops (SPW)*, 2024.

[7] artillery.io, "Artillery," May 2025. [Online]. Available: https://github.com/artilleryio/artillery

[8] S. Pillai, T. Egiazarov, and M. Bhat, "When to use graphql to accelerate api delivery," Gartner, Tech. Rep., 2024.

[9] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "Fuzzbench: an open fuzzer benchmarking platform and service," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[10] R. Natella and V.-T. Pham, "Profuzzbench: a benchmark for stateful protocol fuzzing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021.

[11] O. Sahin, M. Zhang, and A. Arcuri, "Wfc/wfd: Web fuzzing commons, dataset and guidelines to support experimentation in rest api fuzzing," 2025. [Online]. Available: https://arxiv.org/abs/2509.01612

[12] A. Arcuri, M. Zhang, A. Golmohammadi, A. Belhadi, J. P. Galeotti, B. Marculescu, and S. Seran, "Emb: A curated corpus of web/enterprise applications and library support for software testing research," in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2023.