Proposal of an Automated Testing Method for GraphQL APIs Using Reinforcement Learning

Kenzaburo Saito, Yasuyuki Tahara, Akihiko Ohsuga and Yuichi Sei
The University of Electro-Communications, Japan
s2330049@gl.cc.uec.ac.jp, {tahara, ohsuga, seiuny}@uec.ac.jp

Keywords: API Testing, Reinforcement Learning, GraphQL.

Abstract: GraphQL is a new query language for APIs that has a different structure from the commonly used REST API, making it difficult to apply conventional automated testing methods. This necessitates new approaches. This study proposes GQL-QL, an automated testing method for GraphQL APIs using reinforcement learning. The proposed method uses Q-learning to explore the test space. It generates requests by selecting API fields and arguments based on the schema and updates Q-values according to the response. By repeating this process and learning from it, efficient black-box testing is achieved. Experiments were conducted on publicly available APIs to evaluate the effectiveness of the proposed method using schema coverage and error response rate as metrics. The results showed that the proposed method outperformed existing methods on both metrics.

1 INTRODUCTION

Graph Query Language (GraphQL) is a query language for APIs released by Meta in 2015 and has attracted attention as an alternative to REST APIs(GraphQL Foundation., 2024). It is used in some of the APIs provided by large services such as Facebook, Shopify, GitHub, and GitLab. Unlike REST APIs, GraphQL has a single endpoint where clients can specify the required fields to retrieve data. This prevents over-fetching or under-fetching of data and allows multiple operations to be performed in a single API call, enabling fast and secure processing. However, GraphQL often deals with nested objects or objects that may involve circular references, making it highly complex. Additionally, since it does not conform to HTTP specifications, it is difficult to use HTTP status codes for error handling.

Web services are often large and complex, resulting in a vast number of potential test cases. End-to-end (E2E) testing alone may not be sufficient for early detection of issues or identifying detailed causes of failures. Therefore, automated API testing plays an important role in development.

Automated testing for REST APIs has gained significant attention. According to Golmohammadi et al.'s survey (Amid Golmohammadi, 2023), more than 90 papers related to REST API testing had been published by 2022. Recent examples of tools that automatically generate test cases for REST APIs include EvoMaster (Andrea Arcuri, 2021), RESTler (Vaggelis Atlidakis, 2019), Morest (Yi Liu, 2022), and ARAT-RL (Myeongsoo Kim, 2023a). Additionally, various approaches exist such as AGORA (Juan C. Alonso, 2023), which automatically generates test oracles, and NLPtoREST (Myeongsoo Kim, 2023b), which enhances tests using natural language processing.

However, research on automated testing for GraphQL APIs is still limited; we have identified only four approaches so far (discussed in Section 2.3). Moreover, since GraphQL APIs differ significantly from REST APIs in their approach, it is challenging to apply existing automated testing methods designed for REST APIs directly.

In this study, we propose GQL-QL—a black-box testing method for GraphQL APIs using reinforcement learning. Black-box testing refers to a technique that tests the system under test (SUT) without using its internal structure or code. This makes it suitable for applying to other types of APIs compared to white-box testing methods that rely on internal structures or code. Additionally, since GraphQL API tests involve a vast exploration space and dynamic responses can be utilized in black-box testing, we believe it is well-suited for exploration using reinforcement learning. Therefore, we base our approach on existing reinforcement learning-based automated testing methods for REST APIs.

Experiments were conducted using actual

Saito, K., Tahara, Y., Ohsuga, A. and Sei, Y.
Proposal of an Automated Testing Method for GraphQL APIs Using Reinforcement Learning.
DOI: 10.5220/0013263100003890

In *Proceedings of the 17th International Conference on Agents and Artificial Intelligence (ICAART 2025) - Volume 3*, pages 1101-1107
ISBN: 978-989-758-737-5; ISSN: 2184-433X

GraphQL APIs to evaluate our method based on two metrics: error response rate and schema coverage. The results showed that our method outperformed existing methods on both metrics.

This paper is structured as follows: Section 2 introduces related work; Section 3 explains the proposed method GQL-QL; Section 4 presents experimental methods and results along with discussions; finally, Section 5 concludes this study with future directions.

2 BACKGROUND

2.1 GraphQL

GraphQL refers to both a query language for APIs and a server-side runtime. In GraphQL, resources are fetched by extracting a tree structure starting from a specific node in a graph structure that represents the application's data model. Clients can request only the necessary resources and specify the structure of the returned results, thus reducing both the size of the results and the number of network calls.

The benefits of migrating to GraphQL have been presented in various studies. In a survey by Brito et al., seven REST APIs were migrated to GraphQL, and it was reported that the size of the returned JSON documents was reduced by up to 99%(Gleison Brito, 2019). Another study showed that implementing API queries with GraphQL required less effort than with REST APIs, especially for complex endpoints with many parameters(Gleison Brito, 2020).

In GraphQL, queries must be written according to the schema. To examine accessible resources, clients can use introspective queries on the API to obtain information such as queries, types, fields, and arguments. Listing 1 shows part of a schema obtained by sending an introspective query to the SpaceX GraphQL API(Apollo Graph Inc., 2023).

Listing 1: Part of the SpaceX GraphQL API schema.

```
"kind": "OBJECT",
"name": "Query",
"description": null,
"fields": [
    {
        "name": "capsule",
        "description": null,
        "args": [
            {
                "name": "id",
                "description": null,
                "type": {
                    "kind": "NON_NULL",
                    "name": null,
                    "ofType": {
                        "kind": "SCALAR",
                        "name": "ID",
                        "ofType": null
                    }
                },
                "defaultValue": null
            }
        ],
        "type": {
            "kind": "OBJECT",
            "name": "Capsule",
            "ofType": null
        },
        "isDeprecated": false,
        "deprecationReason": null
```

In this example, information about the Query object, which contains a set of available queries as fields, and the schema information for the capsule query included in those fields is shown. The capsule query takes an ID-type argument called id, and its field is retrieved as a Capsule object. The fields of the Capsule object are described elsewhere in the schema.

An example query based on this schema is shown in Listing 2.

Listing 2: Example query for SpaceX GraphQL API.

```
query {
  capsule(id: "5e9e2c5bf35918ed873b2664"){
    id
    missions {
      name
    }
    status
    type
  }
}
```

In this example, id is specified as an argument, and id, missions, status, and type are specified as fields from the Capsule object. Since missions is a Capsule-Mission object, name is specified from its fields. In this way, it is possible to use objects without specifying all their fields.

The response to this query would look like Listing 3. As shown here, when performing queries in GraphQL, it is necessary to create requests using appropriate object types and scalar types based on schema information.

Listing 3: Example response from SpaceX GraphQL API.

```
{
  "data": {
    "capsule": {
      "id": "5e9e2c5bf35918ed873b2664",
      "missions": null,
      "status": "retired",
      "type": "Dragon 1.0"
    }
  }
}
```

GraphQL has five default scalars: Int, Float, String, Boolean, and ID. Additionally, users can define custom scalars. Typical examples include dates, emails, and UUIDs.

2.2 Q-Learning

Reinforcement Learning (RL) is a type of machine learning where an agent learns by interacting with its environment(Richard S. Sutton, 2018). The agent selects actions in various states and receives rewards from the environment based on those actions. Through trial and error in these interactions—where actions lead to changes in state—the agent learns behavior that maximizes cumulative reward.

Q-learning is a reinforcement learning algorithm used to estimate an optimal action-value function (Q-function)(Christopher J. C. H. Watkins, 1992). The Q-function represents the sum of immediate rewards obtained by an agent executing action a in state s, plus future cumulative rewards expected when following an optimal policy thereafter. These future rewards are discounted by a discount factor γ. In Q-learning, Q-values for specific actions taken in each state are recorded in a Q-table and updated using Equation (1).

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)] \tag{1}$$

Here α is the learning rate; γ is the discount factor; r is the immediate reward; s' is the state after action a; and $\max_{a'} Q(s',a')$ represents the maximum Q-value for actions a' available in state s'.

In reinforcement learning methods like ε-greedy are often used to balance exploration and exploitation(Richard S. Sutton, 2018). In Q-learning an exploration rate ε is set such that with probability $1-\varepsilon$, actions with higher Q-values based on current knowledge are chosen (exploitation), while with probability ε, random actions are selected (exploration).

In automated API testing requests are sent to the SUT, rewards are obtained based on responses which are then used to update values.

2.3 API Testing

AutoGraphQL proposed by Zetterlund et al.(Louise Zetterlund, 2022) is one existing approach for automated testing of GraphQL APIs. This tool automatically generates test cases using GraphQL queries operated by users in production environments. Through oracles that verify whether responses conform to GraphQL schemas, schema violations can be detected.

Vargas et al.(Daniela Meneses Vargas, 2018) proposed deviation testing for GraphQL APIs. This approach automatically generates tests with small deviations from manually created test cases and detects failures by comparing their execution results.

A property-based method for generating black-box test cases from schemas was proposed by Karlsson et al.(Stefan Karlsson, 2021). This method takes a GraphQL schema as input and randomly generates GraphQL queries and argument values using property-based techniques. It represents one of the first studies aimed at fully automating test case generation for GraphQL APIs.

Additionally, Belhadi et al.(Asma Belhadi, 2024) proposed methods using evolutionary computation alongside random-based methods. If source code for a GraphQL API is available and written in supported programming languages, test cases are generated through evolutionary search. For black-box testing without source code access, random search is used to generate queries. These approaches have been integrated into EvoMaster—an open-source tool for automated testing of APIs(Andrea Arcuri, 2021).

For REST APIs specifically, ARAT-RL uses reinforcement learning for automated testing(Myeongsoo Kim, 2023a). This method efficiently explores vast input spaces in API testing by dynamically analyzing request-response data using Q-learning to prioritize API operations and parameters. Each API operation corresponds to a state; selecting specific parameters or scalar value mapping sources corresponds to actions handled through separate Q-tables. Rewards based on response status codes update Q-values accordingly.

3 METHOD

In this study, we propose GQL-QL, an adaptation of ARAT-RL with improvements to make it suitable for GraphQL APIs.

First, the schema of the SUT is retrieved using an introspective query, and Q-values are assigned to each field and argument for initialization. Next, requests are generated by referencing the Q-values of fields and arguments, prioritizing API operations while selecting fields and arguments. Then, based on the content of the response, the rewards for each value are updated, and requests are generated repeatedly.

The details of each step are described below.

3.1 Initialization

In this section, we determine the structure of the Q-table based on the retrieved schema and assign initial values.

First, we set the learning rate α, discount rate γ, and exploration rate ε, and retrieve the schema of the SUT using an introspective query. The schema obtained here has a graph structure, but to make it easier to trace parameters and their associated fields when generating requests, we reshape the schema data into a tree structure with the name of each API operation as the root node. To prevent circular references in object fields, we set a maximum depth for object fields. If an object reference exceeds this depth, that field is treated as a leaf node.

Next, we initialize the Q-table for parameters and scalar value mapping sources with an empty dictionary. The Q-table for scalar value mapping sources is structured so that each mapping source field belongs to a specific API operation, with each value initialized to 0. The Q-table for parameters is managed in a tree structure based on the reshaped schema data, where arguments and fields are assigned to each API operation as shown in Figure 1. Each node in the parameter tree contains information such as name, Q-value, an array of belonging fields (child nodes), whether it is an array or not, whether it allows NULL values or not, and whether it was selected during request generation. Only leaf nodes contain information about scalar types. Each node in the field tree contains information such as name, Q-value, and an array of belonging fields (child nodes). Finally, we record how many times each parameter name appears in the parameter Q-table and initialize each node's Q-value based on its appearance count. Parameters that appear frequently are likely important and are prioritized for selection.

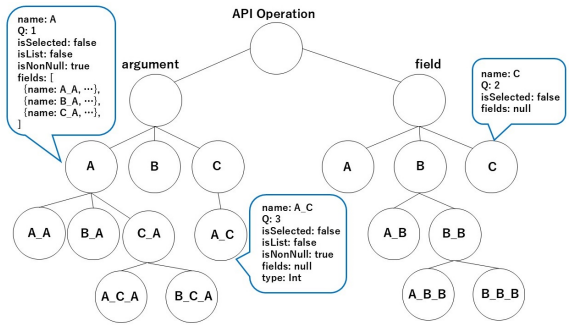

Figure 1: Model of parameter Q-table in GQL-QL.

3.2 Prioritization of API Operations and Parameters

In this section, we select optimal API operations, parameters, and scalar value mapping sources based on Q-values to generate requests.

First, we select which API operation to execute. If a random value is greater than ε, we calculate the average Q-value from the total Q-value and number of nodes for all arguments and fields in each API operation. The API operation with the highest average Q-value is selected; otherwise, a randomly selected API operation is chosen.

Next, parameters are selected based on their Q-values, and scalar values are generated using the selected scalar value mapping source. For parameter nodes at depth d that share the same parent element, a random integer n is chosen between 1 and (number of nodes - d). By limiting the number of selected parameters in this way, we prevent requests from becoming excessively large. If a random value is greater than ε, we select the top n parameters with high Q-values; otherwise, n available parameters are randomly selected. At this time, if non-NULL parameters have not been selected yet, they are additionally selected. If a selected node has children, its depth is updated and similar processing is performed to select parameters. When a leaf node containing scalar type information is selected and if a random value is greater than ε, we refer to the scalar value mapping source's Q-table to use the source with the highest Q-value; otherwise, a random source is chosen to generate scalar values. In this study, we use 'default' and 'random' as mapping sources: 'default' returns preset default values while 'random' returns random values.

In this way, API operations, parameters, and scalar values for arguments are obtained to generate requests.

3.3 Update Q-table

In this section, requests are executed based on what was generated earlier, and then Q-values are updated according to responses.

Since GraphQL cannot accurately use HTTP status codes for error handling purposes, success or failure is determined by whether an 'errors' object exists in the response. In case of success, 1 point is subtracted from the selected scalar value mapping source's Q-value; immediate reward r is assigned as -1; then negative updates are made to selected parameter nodes'Q-values using Equation (1). In case of failure (error), it determines whether it was a valid error based on error messages. If strings such as "ex-

ceeded" or "throttled" appear in messages indicating rate limit errors—unlikely related to detecting failures in SUT—they are treated as invalid errors and skipped. If retry times can be obtained from messages, that amount of time will be waited; otherwise 10 seconds will be waited before generating subsequent requests again. If errors were valid ones instead (i.e., related to failure detection), 1 point will be added back into chosen scalar value mapping source's Q-value while assigning immediate reward r as +1; positive updates would then occur within parameter nodes accordingly using Equation (1). Additionally recorded logs contain error messages during these cases.

In summary: by prioritizing both API operations alongside respective parameters via continuous updates through feedback loops derived from real-time responses—efficient exploration across entire testing space becomes feasible.

4 EVALUATION

4.1 Experiment

To evaluate the proposed method, we tested an actual GraphQL API using GQL-QL. The SpaceX GraphQL API, which is publicly available and does not require authentication, was adopted as the test target. The test environment used Windows 11 with WSL2, and the CPU was an Intel Core i5-10210U. The learning execution time was set to 12 hours, with a learning rate α of 0.1, a discount rate γ of 0.99, and an exploration rate ε of 0.1. For comparison with existing methods, we generated tests using EvoMaster's black-box testing method, as explained in Section 2.3, with the same execution time.

The evaluation metrics used were the error response rate and schema coverage. The error response rate indicates the proportion of responses received that were returned as errors. A higher value suggests a higher ability to detect failures. Schema coverage indicates the proportion of accessible types and field information in the schema that could be accessed during request generation. For calculating coverage, we used the GraphQL-Inspector(THE GUILD, 2019) coverage tool. This tool returns the number of queries, mutations, and subscriptions in the SUT schema and the schema coverage when provided with the SUT's schema file and GraphQL request files. Schema coverage is calculated using the following metrics:

- Types coverage: The proportion of types in the schema that were covered by tests.
- Types fully coverage: The proportion of types whose fields were fully covered by tests.
- Fields coverage: The proportion of fields in the schema that were covered by tests.

4.2 Results

The results for schema coverage at each iteration obtained using GQL-QL are shown in Figures 2, 3, and 4.

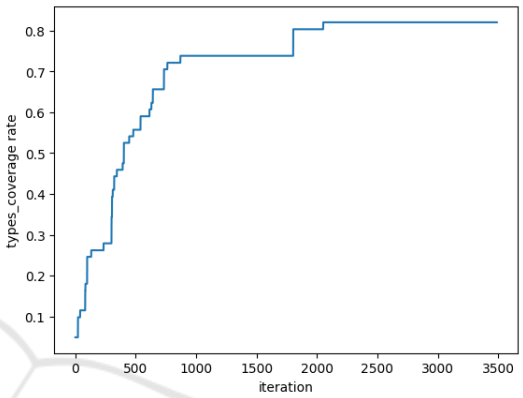

Figure 2: Types coverage rate per iteration.

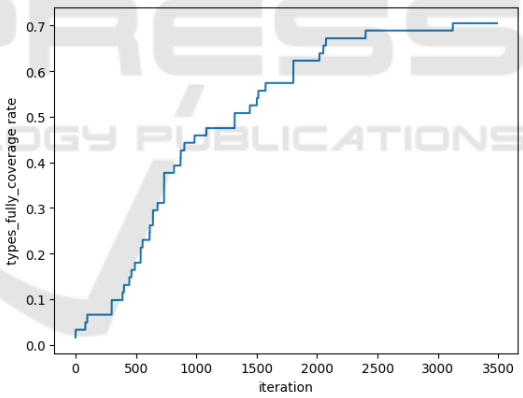

Figure 3: Types fully coverage rate per iteration.

Without relying on specific types or fields, these schema coverages has effectively improved with each iteration. This result indicates that GQL-QL successfully implements an appropriate prioritization and exploration strategy.

The comparison between GQL-QL and EvoMaster's black-box testing method is shown in Table 1.

In both metrics—error response rate and schema coverage—GQL-QL achieved better results than existing methods. In particular, GQL-QL significantly outperformed existing methods in terms of types fully covered. This suggests that Q-learning's prioritization mechanism is functioning well while maintaining a

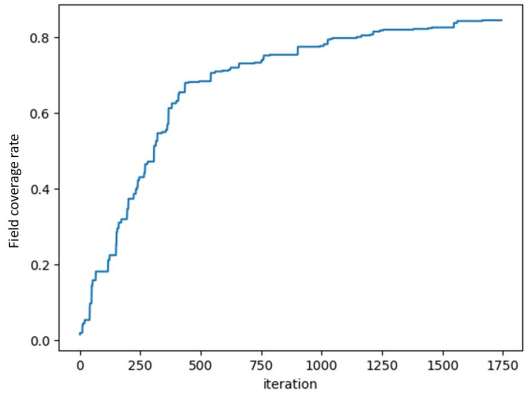

Figure 4: Fields coverage rate per iteration.

Table 1: Comparison of results between EvoMaster and GQL-QL.

	EvoMaster	GQL-QL
Error response rate	13.6%	**41.1%**
Types coverage	80.3%	**82.0%**
Types fully coverage	29.5%	**70.5%**
Fields coverage	67.7%	**84.4%**

balance between exploitation and exploration without repeatedly referencing the same API operations or parameters.

The error response rate per iteration obtained using GQL-QL is shown in Figure 5. The error codes and their occurrences are listed in Table 2.

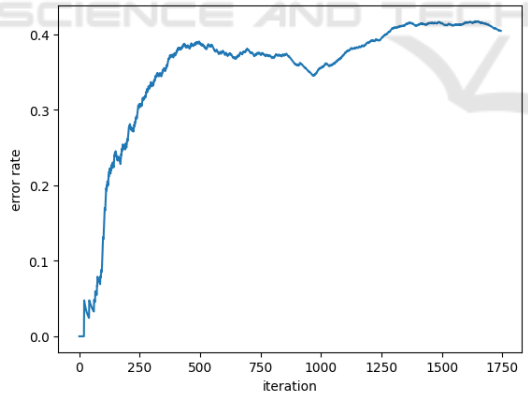

Figure 5: Error response rate per iteration.

Among the recorded errors, some are related to failure detection, such as INTERNAL_SERVER_ERROR, SUBREQUEST_HTTP_ERROR, and SUBREQUEST_SUBREQUEST_MALFORMED_RESPONSE, but there were also invalid errors like PARSING_ERROR and GRAPHQL_VALIDATION_FAILED. These invalid errors likely occurred due to issues with data structure during request generation or missed cases when distinguishing invalid errors. In particular, PARSING_ERROR accounted for most of the errors in this experiment; this error was caused by requests exceeding token limits due to their size, indicating insufficient measures to control request size.

Table 2: Recorded error codes and their occurrences.

Error code	Error number
PARSING_ERROR	602
INTERNAL_SERVER_ERROR	81
GRAPHQL_VALIDATION_FAILED	15
SUBREQUEST_HTTP_ERROR	1
SUBREQUEST_SUBREQUEST_MALFORMED_RESPONSE	1

5 CONCLUSION

In this study, we proposed GQL-QL, a reinforcement learning-based method for automated black-box testing of GraphQL APIs. Using Q-learning, we prioritized the API operations and parameters to be tested and constructed a Q-table in a tree structure to accommodate the GraphQL request format. In comparison experiments with existing methods, GQL-QL achieved better results in both error response rate and schema coverage. However, challenges remained in calculating the error response rate.

Four future challenges are identified: First, expanding the method for generating scalar values. In this study, only default and random values were used, but it will be necessary to enable testing of more diverse patterns, such as intelligently generating values from schema descriptions. Second, improving the reward assignment method. In the proposed method, the same reward was given for all errors. By assigning different rewards based on error codes or error message content, it is believed that the likelihood of uncovering more meaningful errors can be increased. Third, expanding the types of APIs to be tested. In this experiment, a medium-scale API was used, but it is desirable to conduct validation on larger, more practical APIs. Finally, revisiting the evaluation method for failure detection capabilities. In this study, we used the error response rate; however, it included invalid errors. It will be necessary to calculate more reliable evaluation metrics by using a test GraphQL API with embedded faults for validation.

ACKNOWLEDGEMENTS

This work was supported by JSPS KAKENHI Grant Numbers JP22K12157, JP23K28377, JP24H00714. We acknowledge the assistance for the ChatGPT (GPT-4o and 4o mini) was used for proofreading, which was further reviewed and revised by the authors.

REFERENCES

Amid Golmohammadi, Man Zhang, A. A. (2023). Testing restful apis: A survey. In *ACM Transactions on Software Engineering and Methodology, pp.1-41*.

Andrea Arcuri, Juan Pablo Galeotti, B. M. M. Z. (2021). Evomaster: A search-based system test generation tool. In *Journal of Open Source Software*.

Apollo Graph Inc. (2023). Spacex graphql api. https://github.com/apollographql/spacex.

Asma Belhadi, Man Zhang, A. A. (2024). Random testing and evolutionary testing for fuzzing graphql apis. In *ACM Transactions on the Web, pp.1-41*.

Christopher J. C. H. Watkins, P. D. (1992). Q-learning. In *Machine Learning*.

Daniela Meneses Vargas, Alison Fernandez Blanco, A. C. V. J. P. S. A. M. M. T. A. B. S. D. (2018). Deviation testing: A test case generation technique for graphql apis. In *Proceedings of the 11th International Workshop on Smalltalk Technologies (IWST'18), pp.1–9*.

Gleison Brito, M. T. V. (2020). Rest vs graphql: A controlled experiment. In *2020 IEEE International Conference on Software Architecture (ICSA)*.

Gleison Brito, Thais Mombach, M. T. V. (2019). Migrating to graphql: A practical assessment. In *2019 IEEE 26th International Conference on Software Analysis Evolution and Reengineering (SANER), pp.140-150*.

GraphQL Foundation. (2024). Graphql—a query language for your api. https://graphql.org/.

Juan C. Alonso, Sergio Segura, A. R.-C. (2023). Agora: Automated generation of test oracles for rest apis. In *ISSTA 2023: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp.1018-1030*.

Louise Zetterlund, Deepika Tiwari, M. M. B. B. (2022). Harvesting production graphql queries to detect schema faults. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*.

Myeongsoo Kim, Saurabh Sinha, A. O. (2023a). Adaptive rest api testing with reinforcement learning. In *38th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp.446-458*.

Myeongsoo Kim, Davide Corradini, S. S. A. O. M. P. R. T.-B. M. C. (2023b). Enhancing rest api testing with nlp techniques. In *ISSTA 2023: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp.1232-1243*.

Richard S. Sutton, A. G. B. (2018). *Reinforcement Learning: An Introduction*. Bradford Books, 2nd edition.

Stefan Karlsson, Adnan Čaušević, D. S. (2021). Automatic property-based testing of graphql apis. In *IEEE/ACM International Conference on Automation of Software Test (AST)*.

THE GUILD (2019). Graphql inspector. https://the-guild.dev/graphql/inspector.

Vaggelis Atlidakis, Patrice Godefroid, M. P. (2019). Restler: Stateful rest api fuzzing. In *Proceedings of the 41st International Conference on Software Engineering, pp.748-758*.

Yi Liu, Yuekang Li, G. D. Y. L. R. W. R. W.-D. J. S. X. M. B. (2022). Morest: Model-based restful api testing with execution feedback. In *44th International Conference on Software Engineering (ICSE 2022)*.