Harvesting Production GraphQL Queries to Detect Schema Faults

Louise Zetterlund[†], Deepika Tiwari[*], Martin Monperrus[*], and Benoit Baudry[*]

[†]Redeye AB, Sweden
[*]KTH Royal Institute of Technology, Sweden

***Abstract*—GraphQL is a new paradigm to design web APIs. Despite its growing popularity, there are few techniques to verify the implementation of a GraphQL API. We present a new testing approach based on GraphQL queries that are logged while users interact with an application in production. Our core motivation is that production queries capture real usages of the application, and are known to trigger behavior that may not be tested by developers. For each logged query, a test is generated to assert the validity of the GraphQL response with respect to the schema. We implement our approach in a tool called AutoGraphQL, and evaluate it on two real-world case studies that are diverse in their domain and technology stack: an open-source e-commerce application implemented in Python called Saleor, and an industrial case study which is a PHP-based finance website called Frontapp. AutoGraphQL successfully generates test cases for the two applications. The generated tests cover 26.9% of the Saleor schema, including parts of the API not exercised by the original test suite, as well as 48.7% of the Frontapp schema, detecting 8 schema faults, thanks to production queries.**

***Index Terms*—GraphQL, production monitoring, automated test generation, test oracle, API testing, schema**

I. Introduction

Web APIs consist of programmable endpoints to interact with software systems. They can be implemented in different ways, including the well known REST [1] and SOAP [2], [3] paradigms. GraphQL is a new way to define web APIs invented by Facebook in 2015 [4]. A GraphQL API implementation consists of a schema that specifies the data structures and operations exposed by the API, as well as a server that implements the logic to handle API requests, resolving them into actual data. The clients of a GraphQL API, typically a browser or an app, send requests that specify the data they want to retrieve. The performance [5], [6] and flexibility [7] of GraphQL have contributed to its rapid adoption in the industry [8], [9].

While GraphQL offers significant benefits to develop web APIs, correctly implementing it remains a challenge. Specifically, a bug in the server may cause a GraphQL query to be resolved into data that is incompatible with the properties defined in the schema. We designate this kind of fault as a schema fault. Let us consider the example of *Saleor*, an e-commerce platform that exposes a GraphQL API. A user reported an error when trying to create a new product without assigning it to a category[1]. This issue was identified as a schema fault and fixed by the maintainers, because the API implementation contradicted the *Saleor* GraphQL schema. Schema faults are the focus of testing techniques for other systems specified using schemas, such as databases [10], [11], or OpenAPI REST APIs [12]–[14]. However, there has been little work to detect schema faults in GraphQL APIs. Only one approach, by Karlsson *et al.* [15] targets them, by generating GraphQL queries randomly and using them as inputs in property-based tests with the goal of exercising more of the GraphQL schema.

[1]https://github.com/mirumee/saleor/issues/5589

In this work, we propose to harvest GraphQL queries from an application in production, and use them as inputs for test generation. Our motivation is that test cases generated from production GraphQL queries assess the behavior of the application with respect to real API usages. Moreover, it has been shown that production data can lead to valuable test cases that invoke behavior untested by developer-written tests [16], [17]. We implement our technique in a tool called AutoGraphQL, which operates in two phases. The first phase involves monitoring an application in production and logging every unique GraphQL query. In the second phase, AutoGraphQL generates one test for each query logged in the monitoring phase. Each generated test includes the required oracles to assess whether the format of the response is consistent with the GraphQL schema.

We evaluate AutoGraphQL on one open-source and one closed-source case study. Our open-source case study is an e-commerce platform called *Saleor*. Our closed-source industrial case study, *Frontapp*, is the primary website of Redeye AB, an equity research and investment banking company based in Stockholm, Sweden. AutoGraphQL harvests 334 and $24,049$ unique GraphQL queries in production, for Saleor and Frontapp, respectively. Our tool successfully generates one test for each logged query. The generated tests exercise 26.9% of the GraphQL schema in Saleor, including parts of the schema not exercised by the original, developer-written test suite. The tests generated by AutoGraphQL for Frontapp exercise 48.7% of the schema and detect 8 schema faults.

Our evaluation of AutoGraphQL with two diverse case studies demonstrates that it can successfully generate tests with GraphQL queries harvested from production. The generated tests complement developer-written tests by triggering untested behavior, and are able to discover schema faults in the implementation of the GraphQL API. To sum up, our contributions are as follows:

- A novel technique to harvest GraphQL queries from production and use them to generate test cases with oracles tailored for the detection of schema faults
- The evaluation of our technique with one industrial and one open-source case study, which demonstrates that the generated tests trigger untested behavior, and discover schema faults
- An open-source implementation of our methodology in a tool called AutoGraphQL, as well as a publicly available dataset for reproducibility at https://github.com/castor-software/autographql/

The rest of this paper is organized as follows: section II discusses GraphQL, section III introduces AutoGraphQL, and section IV describes the methodology we use to evaluate it. We present the results from this evaluation in section V, and discuss some aspects of AutoGraphQL in section VI. section VII includes related work and section VIII concludes the paper.

II. Background

This section introduces GraphQL, a specification for web APIs, as well as its implementation.

A. GraphQL APIs

GraphQL is a new paradigm to build web APIs. A GraphQL API consists of one schema that defines the data structures that are available through the API, and a set of requests, or 'queries', that can be made against the schema. The implementation of the API is composed of so-called 'resolvers' which map the information requested by the queries to actual data from the underlying database or storage of the application.

GraphQL requests are typically triggered from clients such as the frontend of an application, and are handled by a server at the backend. Unlike REST APIs [7], [18], the requests are not centered around resources [1]. Instead, they are structured around operations. There are two types of requests in GraphQL: queries and mutations, defined as follows. A request that only fetches data, such as the details of a product on a website, is called a `Query`; a request that changes, or "mutates" data, such as adding or updating a shipping address, is called a `Mutation`. Both kinds of requests are sent to a GraphQL endpoint exposed by the application backend, where they are resolved. The corresponding responses are sent back to the frontend, typically as JSON.

B. GraphQL Schemas

A GraphQL schema serves as a contract between the frontend and the backend of the application [19]. A schema is specified with the strongly-typed Schema Definition Language (SDL), which is defined in the GraphQL specification[2]. It includes declarations of object, enum, interface, and union types. The types themselves are composed of fields that define their properties. These fields may be a scalar, such as `Int`, `String`, or `ID`, other object types defined in the schema, or an array thereof. An exclamation mark `!` represents non-nullable fields. In addition to the type declarations, the schema also defines the `Query` and `Mutation` operations that can be performed on them.

```
interface Node {                    type Teaser {
  id: ID!                             title: String!
}                                     subTitle: String
                                      publishedOnSite: Boolean
enum VideoTypeEnum {                  url: String!
  ANIMATION                           duration: Float
  LIVE_ACTION                       }
  SCREENCAST
}

type Video implements Node {        type Query {
  id: ID!                             video(id: ID!): Video
  title: String!                      teasers(first: Int!):
  url: String!                               [Teaser]
  videoType: VideoTypeEnum          }
  teaser: Teaser
}
```

Listing 1: An excerpt from the GraphQL schema of Frontapp

Listing 1 is an excerpt from the GraphQL schema of Frontapp, the primary website of a company called Redeye AB[3], employing one of the authors. The schema defines an `interface` called `Node` which has a non-nullable `id` of type `ID`. The schema also defines two object types. The `Video` type represents a video published on Frontapp. It implements `Node` and it has non-nullable `String` fields that specify its `title` and its `url`. The field `videoType` expresses the kind of a video as one of the values enlisted in the enumeration type `VideoTypeEnum`. A video may also have a `teaser` of type `Teaser`, which itself has non-nullable `String` fields for its `title` and `url`, and a nullable `subTitle`. A teaser also has a `duration` expressed as a `Float`. The `Boolean` field `publishedOnSite` determines if the teaser has been published on Frontapp.

`Query` is a special GraphQL type that defines the entry-points of all GraphQL queries that fetch data. This excerpt of the Frontapp schema defines two entry-points, `video` and `teasers`. A `Video` object can be fetched through a non-nullable `id` argument, through the `video` entry-point. The `teasers` entry-point returns a list of the first *n* `Teaser`-type objects, based on the value of *n* provided as the non-nullable `Int` argument to the variable `first`.

GraphQL allows the requesting entity to explicitly specify, in a single declarative GraphQL query [20], the data or fields required in the response. Listing 2 shows a query made against the schema defined in Listing 1. The query is given an explicit name, called its *operation name*, which is `GetTeasers`. This query is generated by the interactions of end-users as they browse through the videos published on the Frontapp website. This interaction would trigger the `teasers` entry-point and fetch a list of objects of type `Teaser`. The query requests for the `title`, `subTitle`, and `url` of the first `2` teasers. The

[2]https://spec.graphql.org

[3]https://www.redeye.se

```
query GetTeasers {
  teasers(first: 2) {
    title
    subTitle
    url
    __typename
  }
}
```

Listing 2: A production GraphQL query made against the schema defined in Listing 1

```
{
  "data": {
    "teasers": [
      {
        "title": "Finance 101",
        "subTitle": "The basics of finance",
        "url": "https://youtu.be/dQw4w9WgXcQ",
        "__typename": "Teaser"
      },
      {
        "title": "Development 101",
        "subTitle": null,
        "url": "https://youtu.be/jNQXAC9IVRw",
        "__typename": "Teaser"
      }
    ]
  }
}
```

Listing 3: The response for the query in Listing 2

meta-field __typename, wherever used in a query, specifies the type of the object at that point in the query.

C. GraphQL Resolvers

Resolvers are functions to map each field requested in incoming queries with actual data in the application. The resolvers are not written in GraphQL, they may be implemented in any programming language supported by the underlying GraphQL engine, including Java, JavaScript, PHP, Python, and others[4]. Therefore, GraphQL API implementations can evolve [7] while providing stable APIs.

Listing 4 shows a resolver for Frontapp, written in PHP. The resolver fetches the first n Teaser objects, from the teaserRepository, which is the component that interacts with the Frontapp database. The resolver then prepares the response, with values for all the fields requested by the query in Listing 2 fetched from the database. The response is a list of teaser objects with their title, subTitle, and url. We present the response in Listing 3. It contains only the fields explicitly requested in the query, for the first 2 teasers, including their title, subtitle, and url, and with their __typename being Teaser.

III. AutoGraphQL

This section describes AutoGraphQL, a tool that automatically generates tests for the GraphQL backend of an application. We first discuss schema faults, which are the targets for the tests generated by AutoGraphQL. Then, we present an overview of AutoGraphQL, and describe the phases

```
 1 @@ -0 +2 @@
 2 public function resolveTeasers(int $first) {
 3   $teasers = $this->teaserRepository->findMatching($first);
 4   $data = [];
 5   foreach ($teasers as $teaser) {
 6     $newTeaser = new \stdClass();
 7     $newTeaser->title = $teaser->getTitle();
 8     $newTeaser->subTitle = $teaser->getSubTitle();
 9 +   if ($teaser->isPublishedOnSite()) {
10       $newTeaser->url = $teaser->getUrl();
11 +   }
12     $data[] = ["teaser" => $newTeaser];
13   }
14   return (array_column($data, "teaser"));
15 }
```

Listing 4: A resolver that fetches a list of the first n teasers, with a schema fault that has just been introduced

in which it operates. We conclude this section by discussing the implementation of the tool.

A. Schema Faults

In this work, we aim at detecting faults in the implementation of the GraphQL resolvers, leading them to return data with a format that does not conform to the schema. We call these faults "schema faults". They occur in GraphQL APIs when valid queries get resolved into invalid responses, as a result of incorrect mapping between the fields requested and the actual data storage in the application. This response may be sent to the client without the error being explicitly identified as such (say HTTP 5xx or a JSON error object). Such invalid responses basically break the interface contract between the server and the client as specified in the schema. This kind of fault is common, for example, a user of the e-commerce platform Saleor reported an issue[5], confirmed by a developer, due to a schema fault.

In order to detect schema faults, AutoGraphQL automatically generates test oracles, derived from the schema. These test oracles determine that the data returned by the API is well-formed, with respect to the schema. For example, the Boolean field publishedOnSite for a Teaser is defined as nullable in the schema in Listing 1. If the condition on line 9 is introduced in the resolver presented in Listing 4, the url of the teaser object will be resolved to null if publishedOnSite is false or null. This contradicts the schema which specifies that the url of a teaser cannot be null, and is therefore a bug in the implementation of the resolver. This kind of fault would be detected by AutoGraphQL.

B. Overview of AutoGraphQL

AutoGraphQL generates tests that (i) exercise the GraphQL API implementation, and (ii) assess that the data returned as a resolution to a GraphQL query conforms to the schema. AutoGraphQL operates in two phases, illustrated in Figure 1. The first phase consists in monitoring the application in production, in order to collect the queries that are performed by users, as well as their arguments. This data collection process is performed for a given amount of time, decided by

[4]https://graphql.org/code/#language-support

[5]https://github.com/mirumee/saleor/issues/6750

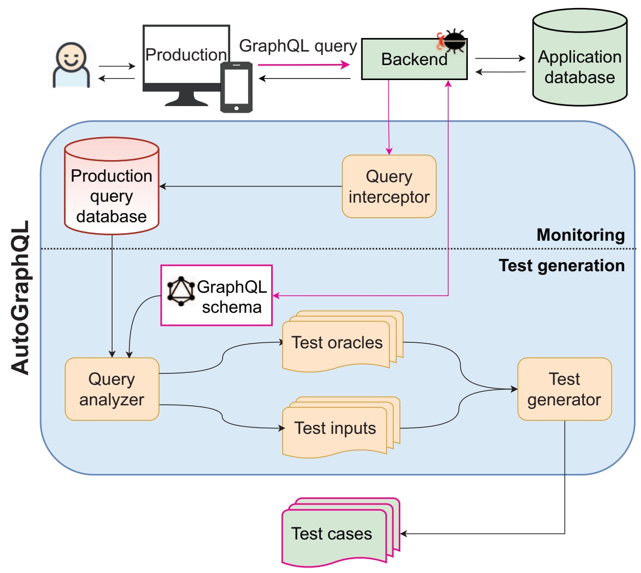

Fig. 1: Overview of AutoGraphQL

developers. The second phase of AutoGraphQL is triggered by developers and consists in analyzing the data that was observed in production to turn them into test cases. AutoGraphQL automatically extracts test inputs from the monitored data, as well as the corresponding oracles from the GraphQL schema. The tests generated by AutoGraphQL use a single GraphQL query as the test input and verify the format of the response to the query. The following two subsections discuss the details of each phase of test generation with AutoGraphQL.

C. Monitoring in Production

Monitoring GraphQL queries in production constitutes the first phase of AutoGraphQL, and is illustrated in the top-half of Figure 1. AutoGraphQL's *query interceptor* monitors the requests sent from the frontend of an application to its backend. It intercepts incoming GraphQL query requests, and the arguments with which they were invoked, and logs them into a database. It also aggregates metadata about these queries, including the number of times a specific query request was invoked or when it was last invoked. The output from this phase is a database of GraphQL queries logged from production.

Listing 5 presents an example of a logged query. The keys `query` and `variables` represent the actual query executed as well as the argument passed to it, respectively. In this case, the query is the same as in Listing 2, and the value of `2` is passed as argument for `first`, to fetch the first `2 teasers`. The entry also includes the operation name for the query (`operation_name`), which is `GetTeasers`. In order to gather statistics about the queries triggered in production, we also save timestamps for when they are first logged (`created_at`) and when they are logged most recently (`updated_at`). Moreover, during our experiments, we observe that a query may frequently be invoked with

```
{
  "query":
    "query GetTeasers($first: Int!) {
      teasers(first: $first) {
        title
        subTitle
        url
        __typename
      }
    }",
  "variables": {
    "first": 2
  },
  "operation_name": "GetTeasers",
  "created_at": "2021-03-03 08:57:46",
  "updated_at": "2021-05-05 16:55:19",
  "times_called": 301016
}
```

Listing 5: A logged GraphQL query from production

the same arguments. We create one entry for each unique combination of query and arguments, and record the frequency of its occurrence with the `times_called` field. The value of $301,016$ means that this combination of query and argument occurred as many times, in production, during the course of our experiment.

D. Test Generation

The second phase of AutoGraphQL, presented in the bottom-half of Figure 1, is triggered by developers whenever they want to generate tests after a period of monitoring. It involves automatically fetching the GraphQL schema of the application from the configured GraphQL endpoint, and using this schema in conjunction with the queries logged in the monitoring phase, in order to produce the inputs and oracles for the generated tests. The output from this phase is the test suite generated from the logged production queries. We now discuss the two components that are responsible for analyzing the logged queries and using them to generate tests.

1) Query Analyzer: The goal of the query analyzer is to use the GraphQL schema of an application and the queries logged in the query database to generate test inputs and their corresponding oracles. Given a logged GraphQL query, such as the one in Listing 5, the analyzer extracts the test input, which is the combination of `query` and its associated `variables`, as well as the `operation_name` given to the query. Next, the query analyzer produces two sets of oracles. We summarize them in Table I, together with their implementation as assertions in the PHPUnit framework[6]. The first set of oracles verify the format of the response, specifically i) its HTTP status code, ii) the validity of the JSON text, and iii) that it does not contain a JSON error object because of an invalid request. The second set of oracles are specific to the schema and i) verify that the response contains all the data requested by the query, and ii) map each object and field requested by the query with the properties defined for it in the schema. These properties include its type, its kind, i.e., whether it is an object, enum, list, or interface, and its nullability.

[6]https://phpunit.de

TABLE I: The oracles generated by AutoGraphQL depending on the response

CATEGORY	ORACLE	IMPLEMENTATION AS PHPUnit ASSERTION
Format	HTTP status code of the response is 200	`assertEquals(200, ...)`
	Response to query is well-formed, valid JSON	`json_decode` doesn't throw exceptions; `assertIsArray(...)`
	Response does not contain a JSON error object	`assertArrayNotHasKey('errors', ...)`
Schema	Response contains all requested fields	`assertArrayHasKey(...)`
	Correct kind of each element in response	`assertIsArray(...) assertContains(...) assertEquals(...)`
	Correct type of each element in response	`assertIsString(...) assertIsBool(...) assertIsNumeric(...)`
		`assertIsInt(...) assertEquals(...)`
	Nullability-contract of each field in response	`assertNotNull(...)`

For example, a subset of the oracles produced for the query in Listing 5 is that the response would contain a field called `title` (`assertArrayHasKey('title', ...)`), which is a non-null (`assertNotNull(...)`) `String` type object (`assertIsString(...)`).

2) Test Generator: The test generator uses the test input and test oracles produced by the query analyzer in order to generate a valid and executable test case that verifies the implementation of the GraphQL API for the application. The output of the test generator is one test case for each logged query. By default, AutoGraphQL generates tests in PHP, using the PHPUnit framework as test driver.

Listing 6 shows the test generated for the query in Listing 5. The test fetches the response to the query (lines 10 to 23) by sending it as an HTTP request to the GraphQL endpoint of an application (lines 25, 26). After verifying its HTTP status code (line 28), the test decodes the response and verifies that it is well-formed JSON (lines 30, 31). Next, the assertion on line 33 ensures that the response does not contain an error due to an invalid query, due to the query trying to fetch data that does not exist, or an exception being raised during query resolution. Lines 36 to 51 contain the assertions produced by the test generator using the oracles derived from the schema. We use the assertion available in PHPUnit to check the validity of collections: `assertIsArray` verifies that `teasers` is a list. Next, `for` each of the items within `teasers`, `assertEquals` verifies that its `__typename` is `Teaser`. `assertArrayHasKey` assertions verify that each of the `teaser` objects in the list has a `title`, a `subTitle`, and a `url`, as requested by the query. The assertions for the `title` and the `url` of a `Teaser` are `assertNotNull` since they are defined as non-nullable, per the schema in Listing 2. Moreover, `assertIsString` verifies that the `title` and `url`, and `if` present, the `subTitle` of a `teaser` are all strings. When this test is executed, assuming the server returns the response shown in Listing 3, 22 assertions are evaluated in total. A failure in any of the assertions in a generated test causes the test to fail, which could be indicative of a schema fault. On the other hand, a generated test that passes can serve as a regression test.

E. Challenges

We now discuss the challenges of test generation with AutoGraphQL.

Query Interception: In order for the query interceptor of AutoGraphQL to monitor and log incoming GraphQL queries, it must be tailored to fit the technology stack of an application. For example, the configuration of the query interceptor used by an application with a backend implemented in Python would differ from the one implemented in Ruby. This is a potentially significant engineering effort.

Testing Database: The execution of the generated test suite requires a running application server with a testing database. This is typically provided by a staging environment. The state of the staging database may have an impact on the execution of the generated tests. Thus an important engineering challenge is to be able to re-initialize a clean staging database before running the AutoGraphQL tests.

F. Implementation

AutoGraphQL is implemented in Python. The query analyzer uses the GraphQLParser of graphql-py[7] to map the elements of a query with the schema and produce test oracles. By default, AutoGraphQL populates a template with the test input and oracles in order to produce tests in the PHPUnit framework. This allows the properties of each node to be expressed as PHPUnit assertions, which serve as oracles in the generated test. Jinja2[8] is the templating language used to render the assertions into PHPUnit test files. We choose PHPUnit for the generated tests, since PHP is a popular server-side language for the implementation of web APIs [21]. AutoGraphQL can generate tests for applications that do not use PHP, or even be extended to support any testing framework, since the generated tests only interact with the HTTP GraphQL endpoint of the application.

IV. Evaluation Methodology

This section describes our two real-world case studies, *Saleor* and *Frontapp*. We also present our experimental setup, including the configuration of AutoGraphQL with the two case studies, their production workloads, as well as the metrics used to evaluate the effectiveness of AutoGraphQL in generating tests for them.

[7]https://github.com/ivelum/graphql-py
[8]https://jinja.palletsprojects.com/en/2.11.x/

```
<?php declare(strict_types=1);

namespace GraphQL;
use PHPUnit\Framework\TestCase;
use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;
use Symfony\Component\HttpFoundation\Request;

class GetTeasersTest extends WebTestCase {
 public function testGraphQL() {
   $client = static::createClient();

   /* Use the details from the logged query */
   $query = <<<’JSON’
   {
     "query": "query GetTeasers($first: Int!) {
       teasers(first: $first) {
         title
         subTitle
         url
         __typename
       }
     }",
     "variables": { "first": 2 },
     "operationName": "GetTeasers"
   }
   JSON;

   /* Make an HTTP request with the query */
   $client->request(’POST’, ’/graphql/’, [], [],
         ["CONTENT_TYPE" => ’application/json’], $query);
   $response = $client->getResponse();

   /* Verify the HTTP status code of the response */
   $this->assertEquals(200, $response->getStatusCode());

   /* Decode the response and verify that it contains valid
         JSON */
   $responseArray = json_decode($response->getContent(),
         true);
   $this->assertIsArray($responseArray, ’Response is not
         valid JSON’);

   /* Verify that the response does not have errors */
   $this->assertArrayNotHasKey(’errors’, $responseArray,
         ’Response contains errors’);
   $responseContent = $responseArray[’data’];

   /* Verify the properties of the response payload per the
         schema */
   $this->assertArrayHasKey(’teasers’, $responseContent);
   if ($responseContent[’teasers’]) {
     $this->assertIsArray($responseContent[’teasers’]);
     for($i = 0; $i < count($responseContent[’teasers’]);
           $i++) {
       if ($responseContent[’teasers’][$i]) {
         $this->assertEquals(’Teaser’ ,
               $responseContent[’teasers’][$i][’__typename’]);
         $this->assertArrayHasKey(’title’,
               $responseContent[’teasers’][$i]);
         $this->assertNotNull(
               $responseContent[’teasers’][$i][’title’]);
         $this->assertIsString(
               $responseContent[’teasers’][$i][’title’]);
         $this->assertArrayHasKey(’subTitle’,
               $responseContent[’teasers’][$i]);
         if ($responseContent[’teasers’][$i][’subTitle’]) {
           $this->assertIsString(
                 $responseContent[’teasers’][$i][’subTitle’]);
         }
         $this->assertArrayHasKey(’url’,
               $responseContent[’teasers’][$i]);
         $this->assertNotNull(
               $responseContent[’teasers’][$i][’url’]);
         $this->assertIsString(
               $responseContent[’teasers’][$i][’url’]);
       }
     }
   }
 }
}
```

Listing 6: A PHPUnit test generated using the logged query in Listing 5, based on the schema in Listing 1

TABLE II: Case studies for the evaluation of AutoGraphQL

PROJECT	LOC	COMMITS	LANGUAGE	DOMAIN
Saleor	691K	17.4K	Python	E-commerce
Frontapp	154K	7K	PHP	Finance

A. Case Studies

We use one open-source and one industrial project as case studies in order to evaluate the effectiveness of AutoGraphQL. We describe these projects and some relevant metrics below.

1) Saleor: Saleor is a widely-used, open-source e-commerce platform, maintained by more than 170 contributors. The project has more than $13K$ stars on GitHub. It is a well-documented and mature project, that can be deployed with Docker. Saleor is crafted with modern technologies, such as Django[9], PostgreSQL, Redis, React, and TypeScript. It offers both a storefront for customers to browse through a catalog of products and make purchases, as well as a dashboard for administrators to manage products, users, and orders. Incoming requests from both of these frontends are handled by a GraphQL server implemented as part of the core component of Saleor. The test suite of Saleor contains automated tests for both the backend and the two frontend components. We choose Saleor as a case study in order to have reproducible experiments with an open-source project, and to demonstrate the versatility of AutoGraphQL in generating tests that target the GraphQL implementation of an application, regardless of the underlying backend technology.

Table II summarizes the key characteristics of the case studies: the number of lines of code and of commits, the language implementing the GraphQL API and the domain of the case study. We use the latest stable release of Saleor, version 2.11, for our experiments. As mentioned in the table, this version contains $691K$ lines of code and the backend is in Python.

2) Frontapp: Frontapp is the primary website of our industrial partner, Redeye AB. Frontapp contains articles, financial analyses, tools, and video streams of events hosted by Redeye. The site is implemented in Symfony[10], a web application framework for PHP projects, and in JavaScript. Its GraphQL API is connected to multiple data sources and receives approximately $64K$ requests daily. Frontapp has been in production for more than 7 years. There is no automated test for the application, and it is tested manually by the QA team before major versions are released.

As presented in Table II, more than 20 Redeye developers have contributed about $7K$ commits to Frontapp. The application contains nearly $154K$ lines of code (LOC), as measured on February 09, 2021, and the GraphQL API is implemented in PHP.

[9]https://www.djangoproject.com/
[10]https://symfony.com/

B. Experiments

This section describes the experimental protocol followed and the metrics used to evaluate AutoGraphQL with Frontapp and Saleor.

1) Query Interceptor Configuration: As described in subsection III-E, the query interceptor of AutoGraphQL is specific to a given software stack. In order to conduct experiments with Saleor, we extend it with an agent, implemented as a GraphQL middleware [22]. This agent allows the query interceptor of AutoGraphQL to access queries that arrive at the GraphQL endpoint of Saleor, `/graphql/`, and log them into the query database. For our experiments with Frontapp, we configure a PHP event listener that triggers the query interceptor, which then logs all queries arriving on the GraphQL endpoint, which is also `/graphql/`.

2) Production Workloads: AutoGraphQL generates tests for queries that are triggered by user actions as part of interactions in production, during the monitoring phase of AutoGraphQL. We define such a production workload for each case study, as follows.

For our experiments with Saleor, we deploy the e-commerce application in a local server in our laboratory. In order to produce a realistic production workload, one of the authors interacted with the components on the frontend to perform typical operations related to e-commerce websites, such as browsing through the catalog of products, searching for specific products from the search bar, viewing the web-page for a product, and making orders. Additionally, the author performed administrative actions such as fetching the list of registered customers and orders, or searching for a specific customer or order. The experiment was carried out over the duration of nearly 3 hours.

The production workload for Frontapp consists of the interactions of actual end-users with the system. We do not ask the end-users to perform any specific operations, and simply log the queries that are generated as they browse through the website, reading articles or using its search feature, etc. We log these queries for a period of 33 days.

3) Metrics for Evaluation: In order to gauge the effectiveness of the tests generated by AutoGraphQL, we adopt the concept of schema coverage introduced by Karlsson *et al.* [15]. Compared to traditional code coverage, schema coverage is a more relevant metric to assess the tests generated by AutoGraphQL since they are intended to directly target the GraphQL schema.

Schema Coverage: We consider a GraphQL schema to be composed of a set of tuples of the form $\{Object_o, Field_n\}$, by combining all *Object o*, defined as a `type` or an `interface` in the schema, with each of its n fields. A query is said to reach a tuple $\{o, f\}$ if it requests for a field f defined in the object o. This is determined statically by analyzing the Abstract Syntax Tree of the query. The schema coverage (SCHEMA_COV) of a test, or the test suite, generated by AutoGraphQL is then defined as the number of tuples in the schema that are reached by the query, or set of queries, invoked by the test(s) (COVERED_TUPLES), divided by the total number of tuples in the schema (SCHEMA_TUPLES). This is presented in Equation 1.

$$\text{SCHEMA\_COV} = \frac{\text{COVERED\_TUPLES}}{\text{SCHEMA\_TUPLES}} \tag{1}$$

A schema coverage of 0% means that the test suite of a project does not invoke any queries that cover a tuple in the schema. On the other hand, a schema coverage of 100% would imply that all the tuples in the schema are covered by the test suite. For example, the query in Listing 2 covers 4 tuples, $\{$*Query, teasers*$\}$, $\{$*Teaser, title*$\}$, $\{$*Teaser, subTitle*$\}$, *and* $\{$*Teaser, url*$\}$, of the 13 tuples of the schema in Listing 1. The schema coverage of the corresponding test in Listing 6 is therefore 30.8%.

Metrics: We collect and report the following metrics for Frontapp and Saleor, based on their schema, logged production queries, test generation, and test execution:

1 TYPES is the number of types defined in the schema.
2 ENTRY_POINTS is the number of entry-points defined in the `Query` type of the schema.
3 UNIQUE_QUERIES is the number of unique combinations of queries and arguments logged during the experiment, and consequently, the number of tests generated.
4 ASSERTIONS_EVALUATED is the total number of assertions evaluated on executing the generated tests.
5 PASSING is the number of generated tests that pass.
6 FAILING is the number of generated tests that do not pass.
7 SCHEMA_FAULTS is the number of bugs found by the generated tests.
8 SCHEMA_TUPLES is the number of tuples obtained from the schema.
9 COVERED_TUPLES is the number of tuples of the schema reached by the generated tests.
10 SCHEMA_COV_GENERATED is the schema coverage achieved with the test suite generated by AutoGraphQL, per Equation 1.

All metrics are integer quantities, except for SCHEMA_COV_GENERATED which is expressed as a percentage.

V. Evaluation Results

This section presents the results obtained during our experiments with the two case studies. We summarize the results for all metrics introduced in subsubsection IV-B3 in Table III.

A. Case Study 1: Saleor

As presented in Table III, the GraphQL schema of Saleor defines 460 TYPES and 69 query ENTRY_POINTS. Based on the production workload defined in subsubsection IV-B2, AutoGraphQL logs 334 UNIQUE_QUERIES, and generates one test for each of them. We successfully execute all of these tests. The generated test suite triggers 43 of the 69 query entry-points in the schema. These tests cover 506 tuples (COVERED_TUPLES) out of the 1884

TABLE III: Results from the evaluation of AutoGraphQL on the two case studies

#	Metric	Saleor	Frontapp
1	TYPES	460	92
2	ENTRY_POINTS	69	23
3	UNIQUE_QUERIES	334	24,049
4	ASSERTIONS_EVALUATED	88,668	8,727,519
5	PASSING	334	23,892
6	FAILING	0	157
7	SCHEMA_FAULTS	0	8
8	SCHEMA_TUPLES	1884	875
9	COVERED_TUPLES	506	426
10	SCHEMA_COV_GENERATED	26.9%	48.7%

SCHEMA_TUPLES in Saleor. This results in a value of 26.9% for SCHEMA_COV_GENERATED.

The execution of the 334 test cases triggers the evaluation of 88, 668 assertions (ASSERTIONS_EVALUATED). The difference between the number of tests and the number of assertions evaluated within the tests is because some assertions are made inside loops to verify the properties of elements that are lists, as illustrated on line 39 of Listing 6. Each of the 88, 668 assertions verifies one expected property about the returned data, per the GraphQL schema. None of the 88, 668 assertion evaluations fail, meaning that the generated test cases based on our selected production workload, do not reveal a schema fault in version 2.11 of Saleor's GraphQL resolvers. This is to be expected given the popularity and maturity of Saleor.

Saleor has a solid test suite written by the developers. Now we want to assess the complementarity of the original tests and the test cases generated by AutoGraphQL. In particular, we consider two metrics: SCHEMA_COV_ORIGINAL is the schema coverage achieved with the GraphQL requests triggered by the original test suite. DISTINCT_TUPLES is the number of schema tuples not covered by the original test suite but covered by the test suite generated by AutoGraphQL. A non-zero value for DISTINCT_TUPLES would imply that AutoGraphQL is able to generate valuable new tests.

The original test suite of Saleor includes 5405 test cases, which trigger 2340 requests, of which 1227 are queries and 1113 are mutations. Table IV shows the number of tuples involved in those tests. This original test suite covers 1429 of the 1884 tuples, resulting in a value of 75.8% for SCHEMA_COV_ORIGINAL. The AutoGraphQL test suite covers 506 tuples, including 483 tuples covered by the original as well as the generated test suite. Most importantly, the tests generated by AutoGraphQL using production queries cover 23 DISTINCT_TUPLES in the Saleor schema that are never covered by the original test suite, including one query entry-point. This confirms the findings of Wang *et al.* [17] and

TABLE IV: Coverage of Saleor schema tuples with original and generated tests

	COVERED_TUPLES
Original test suite	**1429** / 1884
AutoGraphQL-generated test suite	**506** / 1884
Intersection of AutoGraphQL and original test suites	**483** / 1884
Tuples only covered by AutoGraphQL tests (DISTINCT_TUPLES)	**23** / 1884

Tiwari *et al.* [16] that in-house tests can miss behavior that is exercised in production. These unique tuples covered by the generated tests complement the existing test cases, and the generated tests would contribute to the prevention of regression bugs in the resolvers that handle these tuples. We also note that 432 of the 1884 tuples in the schema are covered neither by the original tests, nor by the generated tests, showing that comprehensive schema testing is hard. We will discuss the possible reasons why parts of the schema are not covered by the generated tests in more detail in section VI.

Highlight from the Saleor experiment

With the 334 GraphQL queries harvested during our experiments with Saleor, AutoGraphQL generates 334 test cases. These tests cover 26.9% of the schema, including 23 tuples in the schema that have not been covered by the developers in the original test suite. This reveals that the AutoGraphQL tests complement the original test suite with respect to the capability of detecting schema faults in GraphQL resolvers.

B. Case Study 2: Frontapp

From Table III, we see that the schema of Frontapp defines 92 TYPES and 23 query ENTRY_POINTS. The Frontapp schema does not define `Mutation` operations. The production workload of Frontapp, described in subsubsection IV-B2, observed over a monitoring period of 33 days, results in AutoGraphQL harvesting and storing 24, 049 UNIQUE_QUERIES. The query most frequently invoked during this period was executed 301, 016 times and is presented in Listing 5. Using the logged queries, AutoGraphQL generates 24, 049 PHPUnit tests, all of which are successfully executed. Frontapp has 875 SCHEMA_TUPLES of which 426 are covered by the generated tests (COVERED_TUPLES), resulting in a SCHEMA_COV_GENERATED of 48.7%. The generated tests trigger 19 of the 23 entry-points in the schema. The number of ASSERTIONS_EVALUATED on the execution of the generated tests is 8, 727, 519.

Of the 24, 049 generated tests, 157 fail. The developers at Redeye confirmed that these failures are caused by 8 distinct SCHEMA_FAULTS in Frontapp. The difference between the number of failures and the number of schema faults is the result of some test cases triggering the same query entry-points, and therefore, the same bugs. These schema faults are caused due to incorrect assumptions about the properties of

```
1  @@ -0 +3 @@
2  + if (is_array($source['authorIds']) &&
3  + count($source['authorIds']) > 0) {
4      if (id::isValid($source['authorIds'][0])) {
5        $firstAuthor = $this->personRepository->findById(
6          id::fromString($source['authorIds'][0])
7        );
8        if ($firstAuthor && $firstAuthor->getTitle()) {
9          $authorTitle = $firstAuthor->getTitle();
10       }
11     }
12 + }
```

Listing 7: A schema fault discovered in Frontapp, and its resolution

objects, causing them to contradict the properties defined in the schema. For example, a nullable variable was sent to a resolver that could not handle a null input, or an element was collected in a non-nullable array without being checked for null first, or a resolver returned a different type than was stated in the schema.

Let us now look at an example of a schema fault found by a generated test. Listing 7 shows a bug located within a resolver defined in Frontapp. This bug was caused because the field `authorIds`, defined as non-nullable in the Frontapp schema, actually had the value of `null`, causing an exception to be raised (line 4). It was discovered due to a failing assertion in a test generated by AutoGraphQL, specifically the assertion that checks if the response has an `errors` field. The bug was consequently fixed by Frontapp developers by adding the highlighted check (lines 1 and 2) to ensure that `authorIds` is indeed not null before performing further computations on it.

As mentioned in subsection IV-A, we note that Frontapp does not have automated tests. Thus, the developers at Redeye proved to be interested in the AutoGraphQL test cases, which complement their manual QA activities. Furthermore, it is a possibility to push the AutoGraphQL tests in a repository with continuous integration, and to run them regularly to identify regressions or new schema faults as the application continues to evolve.

Highlight from the Frontapp experiment

AutoGraphQL harvests 24,049 GraphQL queries that are triggered due to interactions of Frontapp end-users in production, and generates as many tests. The generated test suite achieves a schema coverage of 48.7% and discovers 8 schema faults. Those faults have subsequently been fixed by the developers. This validates the capability of AutoGraphQL to automatically generate valuable tests that detect faults, from GraphQL queries observed in production.

VI. Discussion

We now reflect on some interesting aspects of AutoGraphQL.

Test Minimization and Prioritization: Test generation with AutoGraphQL is systematic in that each unique query harvested from production is used as input for the generation of one test. Thus, the number of harvested queries determines the size of the generated test suite, and consequently its execution time. For example, the 334 generated tests for Saleor are executed in 34 seconds, while the 24,049 tests generated for Frontapp take 114 hours to execute. In situations where the execution time is critical, such as in a continuous integration pipeline, it would be useful to minimize the generated test suite, as well as prioritize the execution of the tests [23], [24]. As described in subsection III-C, the query interceptor of AutoGraphQL aggregates meta-data about each query logged during the monitoring phase, including the number of times it was observed, as well as timestamps for when it was first and last invoked. This information may be used by developers to filter a subset of queries to use as inputs for test generation. For example, a developer may generate tests using the queries triggered at least 500 times within the last 3 days, and execute these tests in a prioritized fashion, based on criteria such as their schema coverage.

Mutation Requests: State of the art techniques for GraphQL, including cost analysis [5], [25], formal analysis [26], and GraphQL test generation [27] only support query requests, with the exception of [15] which provides support for the generation of mutation requests. Outside the academic literature, *Schemathesis* is an open-source tool that uses the GraphQL schema to generate property-based tests, it also only supports query requests[11]. However, we note that mutation operations are an equally integral part of GraphQL APIs. For example, the Saleor schema defines 222 mutation entry-points. Thus, there is a clear need for research on mutation requests. This is an interesting and challenging research endeavour because mutation requests have side-effects on the application database, which may result on breaking tests depending on test ordering and breakage of various assumptions on the application state.

GraphQL Schema Evolution and Test Generation: A GraphQL schema and the implementation of the corresponding resolvers may evolve at a different pace. Some parts of the schema may correspond to functionality that is slated to be deprecated, such as the field in a type that is to be replaced by another field. The schema might also specify elements that are yet to be implemented as part of a future release. For example, the developers at Redeye confirm that the Frontapp schema specifies more elements than those which can be handled by the current resolvers. This is due to the fact that Redeye began the process of migrating Frontapp from a REST API to a GraphQL API in 2019 [8]. This impacts the schema coverage achieved with the tests generated by AutoGraphQL, since there are tuples in the schema that are unreachable by design. For the same reason, the AutoGraphQL tests may become outdated and even break when the schema evolves. Therefore, an important direction for research in automated test generation for GraphQL is to understand how the tests may be evolved to address the schema evolution [28]. One approach to evolve these tests could involve repairing the generated test

[11]https://schemathesis.readthedocs.io/en/stable/graphql.html

suite [29].

VII. Related Work

This section discusses closely related work in the areas of test generation for web APIs, for data schemas, and based on production.

A. Test Generation for Web APIs

Currently, only two studies propose automated test generation strategies for GraphQL APIs. Vargas *et al.* [27] mutate GraphQL queries in existing tests in order to amplify them [30]. Karlsson *et al.* [15] produce randomly-generated queries and arguments based on the GraphQL schema, and use them as inputs in property-based tests. AutoGraphQL differs from these approaches because the inputs for test generation are not existing test queries or randomly generated ones, but queries that are harvested from production.

Several studies propose black-box test generation approaches for REST APIs [12]. In addition to the HTTP status code of the response, the oracles are often derived from OpenAPI/Swagger specifications describing the API. The parameters used as test inputs may be derived from the API specification [14], or produced randomly [13], [31]. Recently, deep learning models have been proposed to determine the validity of these inputs [32]. The generated tests can assess the robustness of the API through invalid requests [33], detect regressions across API versions [34], verify the data dependencies among sequences of requests [35], or verify the constraints imposed on their parameters [36]. Metamorphic relations among requests may also serve as the oracle [37]. EvoMaster [38] is a search-based, white-box approach to generate tests for RESTful web services. The technique is based on an evolutionary algorithm which rewards code coverage and fault-finding ability, the latter being determined by HTTP status codes. AutoGraphQL is a black-box approach, that is fundamentally different from these test generation techniques because it is tailored to GraphQL APIs, and uses GraphQL schemas and requests monitored in production.

B. Test Generation for Data Schemas

Traditional databases are also defined with a schema, as is GraphQL. Several studies use database schemas in the context of testing. For example, Khalek and Khurshid [39] use SQL schemas to generate SQL queries, test data, and oracles verifying the result of query execution, with the goal of testing database engines. McMinn *et al.* [40] and Alsharif *et al.* [10] propose search-based approaches that use the schema to generate test data for covering database integrity constraints. QAGen by Binnig *et al.* [41] generates meaningful test inputs based on the schema. XML schemas have also been used to produce XML instances automatically, which may be used as inputs for testing web services [42]–[45]. AutoGraphQL relates to this domain, but in a new technological context, that of web APIs and GraphQL: it uses the GraphQL schema of an application to produce oracles in the generated tests that verify the format of the GraphQL responses.

C. Test Generation Based on Production

A few studies propose test generation strategies using information obtained from production. For example, Oracle Database Replay [46] and Snowtrail [47] capture production queries made against databases, and replay them in order to detect regressions. Marchetto *et al.* [48] use event logs to generate Selenium tests for web applications. Hammoudi *et al.* [29] incrementally repair tests for web applications generated from record-replay tools. Tiwari *et al.* [16] monitor methods of interest in production in order to generate differential unit tests. Thummalapenta *et al.* [49] use execution traces to generate parameterized unit tests. ReCrash by Artzi *et al.* [50] reproduces failures through unit tests generated from runtime observations. AutoGraphQL is the first tool that harvests GraphQL queries from production to use as inputs for the generation of test cases.

VIII. Conclusion

GraphQL is a new way to specify web APIs. Though it continues to gain widespread adoption, few studies propose automated test generation strategies that target GraphQL API implementations. This paper introduces AutoGraphQL, the first tool that leverages production GraphQL queries to automatically generate test cases. The goal of the generated tests is to detect schema faults through oracles that verify that the response to a query conforms with the GraphQL schema.

We present the evaluation of AutoGraphQL on one open-source and one industrial case study, called Saleor and Frontapp: AutoGraphQL successfully generates tests for both projects. The tests generated for Saleor exercise 26.9% of the schema and cover regions in the GraphQL schema that are not covered by its original test suite. The tests generated for Frontapp exercise 48.7% of the schema and reveal 8 distinct schema faults. These experiments demonstrate that AutoGraphQL is capable of generating tests for untested behavior, as well as detecting errors that occur in the production environment.

An important future direction for AutoGraphQL is to analyze how these tests may be incorporated into a continuous integration pipeline. This would require the generated test suite to be minimized, and for the tests to run in a prioritized fashion. It would also be useful to understand how these tests may be evolved as a result of changes made to the API.

IX. Acknowledgements

This work has been partially supported by the Wallenberg Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

[1] R. T. Fielding and R. N. Taylor, *Architectural Styles and the Design of Network-Based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[2] A. Davis and D. Zhang, “A comparative study of soap and dcom,” *Journal of Systems and Software*, vol. 76, no. 2, pp. 157–169, 2005.

[3] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, “Unraveling the web services web: an introduction to soap, wsdl, and uddi,” *IEEE Internet Computing*, vol. 6, no. 2, pp. 86–93, 2002.

[4] L. Byron, “GraphQL: A data query language.” https://engineering.fb.com/2015/09/14/core-data/graphql-a-data-query-language/ (accessed 2021-07-01).

[5] A. Cha, E. Wittern, G. Baudart, J. C. Davis, L. Mandel, and J. A. Laredo, “A principled approach to graphql query cost analysis,” in *Proc. of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, p. 257–268, 2020.

[6] 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*, pp. 123–132, 2019.

[7] G. Brito and M. T. Valente, “Rest vs graphql: A controlled experiment,” in *2020 IEEE International Conference on Software Architecture (ICSA)*, pp. 81–91, 2020.

[8] 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)*, pp. 140–150, IEEE, 2019.

[9] S. L. Vadlamani, B. Emdon, J. Arts, and O. Baysal, “Can graphql replace rest? a study of their efficiency and viability,” in *2021 IEEE/ACM 8th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*, pp. 10–17, IEEE, 2021.

[10] A. Alsharif, G. M. Kapfhammer, and P. McMinn, “Domino: Fast and effective test data generation for relational database schemas,” in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 12–22, IEEE, 2018.

[11] P. McMinn, C. J. Wright, C. Kinneer, C. J. McCurdy, M. Camara, and G. M. Kapfhammer, “Schemaanalyst: Search-based test data generation for relational database schemas,” in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 586–590, IEEE, 2016.

[12] D. Corradini, A. Zampieri, M. Pasqua, and M. Ceccato, “Empirical comparison of black-box test case generation tools for restful apis,” in *21st IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE, 2021.

[13] S. Karlsson, A. Čaušević, and D. Sundmark, “Quickrest: Property-based test generation of openapi-described restful apis,” in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pp. 131–141, IEEE, 2020.

[14] H. Ed-douibi, J. L. Cánovas Izquierdo, and J. Cabot, “Automatic generation of test cases for rest apis: A specification-based approach,” in *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*, pp. 181–190, 2018.

[15] S. Karlsson, A. Čaušević, and D. Sundmark, “Automatic property-based testing of graphql apis,” in *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, pp. 1–10, 2021.

[16] D. Tiwari, L. Zhang, M. Monperrus, and B. Baudry, “Production monitoring to improve test suites,” *IEEE Transactions on Reliability*, pp. 1–17, 2021.

[17] Q. Wang, Y. Brun, and A. Orso, “Behavioral execution comparison: Are tests representative of field behavior?,” in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 321–332, 2017.

[18] P. Erlandsson and J. Remes, “Performance comparison: Between graphql, rest & soap,” Master’s thesis, University of Skövde, 2020.

[19] E. Wittern, A. Cha, J. C. Davis, G. Baudart, and L. Mandel, “An empirical study of graphql schemas,” in *International Conference on Service-Oriented Computing*, pp. 3–19, Springer, 2019.

[20] M. Cederlund, “Performance of frameworks for declarative data fetching: An evaluation of falcor and relay+ graphql,” Master’s thesis, Kungliga Tekniska Högskolan, 2016.

[21] J. Imtiaz, M. Z. Iqbal, *et al.*, “An automated model-based approach to repair test suites of evolving web applications,” *Journal of Systems and Software*, vol. 171, p. 110841, 2021.

[22] N. Burk, “Open Sourcing GraphQL Middleware - Library to Simplify Your Resolvers.” https://www.prisma.io/blog/graphql-middleware-zie3iphithxy (accessed 2021-07-13).

[23] J. Liang, S. Elbaum, and G. Rothermel, “Redefining prioritization: continuous prioritization for continuous integration,” in *Proceedings of the 40th International Conference on Software Engineering*, pp. 688–698, 2018.

[24] J. A. P. Lima and S. R. Vergilio, “Test case prioritization in continuous integration environments: A systematic mapping study,” *Information and Software Technology*, vol. 121, p. 106268, 2020.

[25] G. Mavroudeas, G. Baudart, A. Cha, M. Hirzel, J. A. Laredo, M. Magdon-Ismail, L. Mandel, and E. Wittern, “Learning graphql query costs (extended version),” *arXiv preprint arXiv:2108.11139*, 2021.

[26] O. Hartig and J. Pérez, “Semantics and complexity of graphql,” in *Proceedings of the 2018 World Wide Web Conference*, pp. 1155–1164, 2018.

[27] D. M. Vargas, A. F. Blanco, A. C. Vidaurre, J. P. S. Alcocer, M. M. Torres, A. Bergel, and S. Ducasse, “Deviation testing: A test case generation technique for graphql apis,” in *11th International Workshop on Smalltalk Technologies (IWST)*, pp. 1–9, 2018.

[28] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. Van Deursen, “Mining software repositories to study co-evolution of production & test code,” in *2008 1st international conference on software testing, verification, and validation*, pp. 220–229, IEEE, 2008.

[29] M. Hammoudi, G. Rothermel, and A. Stocco, “Waterfall: An incremental approach for repairing record-replay tests of web applications,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 751–762, 2016.

[30] B. Danglot, O. Vera-Perez, Z. Yu, A. Zaidman, M. Monperrus, and B. Baudry, “A snowballing literature study on test amplification,” *Journal of Systems and Software*, vol. 157, p. 110398, 2019.

[31] V. Atlidakis, P. Godefroid, and M. Polishchuk, “Restler: Stateful rest api fuzzing,” in *ICSE 2019*, November 2019.

[32] A. G. Mirabella, A. Martin-Lopez, S. Segura, L. Valencia-Cabrera, and A. Ruiz-Cortés, “Deep learning-based prediction of test input validity for restful apis,” in *2021 IEEE/ACM Third International Workshop on Deep Learning for Testing and Testing for Deep Learning (DeepTest)*, pp. 9–16, 2021.

[33] N. Laranjeiro, J. Agnelo, and J. Bernardino, “A black box tool for robustness testing of rest services,” *IEEE Access*, vol. 9, pp. 24738–24754, 2021.

[34] P. Godefroid, D. Lehmann, and M. Polishchuk, “Differential regression testing for rest apis,” in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 312–323, 2020.

[35] E. Viglianisi, M. Dallago, and M. Ceccato, “Resttestgen: automated black-box testing of restful apis,” in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pp. 142–152, IEEE, 2020.

[36] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, “Restest: Black-box constraint-based testing of restful web apis,” in *International Conference on Service-Oriented Computing*, pp. 459–475, Springer, 2020.

[37] S. Segura, J. A. Parejo, J. Troya, and A. Ruiz-Cortés, “Metamorphic testing of restful web apis,” *IEEE Transactions on Software Engineering*, vol. 44, no. 11, pp. 1083–1099, 2017.

[38] A. Arcuri, “Restful api automated test case generation with evomaster,” *ACM Trans. Softw. Eng. Methodol.*, vol. 28, Jan. 2019.

[39] S. Abdul Khalek and S. Khurshid, “Automated sql query generation for systematic testing of database engines,” in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE ’10, (New York, NY, USA), p. 329–332, Association for Computing Machinery, 2010.

[40] P. McMinn, C. J. Wright, C. Kinneer, C. J. McCurdy, M. Camara, and G. M. Kapfhammer, “Schemaanalyst: Search-based test data generation for relational database schemas,” in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 586–590, 2016.

[41] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu, “Qagen: generating query-aware test databases,” in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pp. 341–352, 2007.

[42] A. Bertolino, J. Gao, E. Marchetti, and A. Polini, “Automatic test data generation for xml schema-based partition testing,” in *Second International Workshop on Automation of Software Test (AST ’07)*, pp. 4–4, 2007.

[43] J. M. Almendros-Jiménez and A. Becerra-Terón, “Xquery testing from xml schema based random test cases,” in *Database and expert systems applications*, pp. 268–282, Springer, 2015.

[44] D. Petrova-Antonova, K. Kuncheva, and S. Ilieva, “Automatic generation of test data for xml schema-based testing of web services,” in *2015 10th*

International Joint Conference on Software Technologies (ICSOFT), vol. 1, pp. 1–8, IEEE, 2015.

[45] S. C. Lee and J. Offutt, "Generating test cases for xml-based web component interactions using mutation analysis," in *Proceedings 12th International Symposium on Software Reliability Engineering*, pp. 200–209, IEEE, 2001.

[46] Y. Wang, S. Buranawatanachoke, R. Colle, K. Dias, L. Galanis, S. Papadomanolakis, and U. Shaft, "Real application testing with database replay," in *Proceedings of the Second International Workshop on Testing Database Systems*, pp. 1–6, 2009.

[47] J. Yan, Q. Jin, S. Jain, S. D. Viglas, and A. Lee, "Snowtrail: Testing with production queries on a cloud database," in *Proceedings of the Workshop on Testing Database Systems*, pp. 1–6, 2018.

[48] A. Marchetto, P. Tonella, and F. Ricca, "State-based testing of ajax web applications," in *2008 1st International Conference on Software Testing, Verification, and Validation*, pp. 121–130, IEEE, 2008.

[49] S. Thummalapenta, J. De Halleux, N. Tillmann, and S. Wadsworth, "Dygen: Automatic generation of high-coverage tests via mining gigabytes of dynamic traces," in *International Conference on Tests and Proofs*, pp. 77–93, Springer, 2010.

[50] S. Artzi, S. Kim, and M. D. Ernst, "Recrash: Making software failures reproducible by preserving object states," in *European conference on object-oriented programming*, pp. 542–565, Springer, 2008.