

GHTraffic: A Dataset for Reproducible Research in Service-Oriented Computing

Thilini Bhagya, Jens Dietrich, Hans Guesgen
Massey University

Palmerston North, New Zealand

Email: {t.bhagya, j.b.dietrich, h.w.guesgen}@massey.ac.nz

Steve Versteeg

CA Technologies

Melbourne, Australia

Email: steve.versteeg@ca.com

Abstract—We present GHTraffic, a dataset of significant size comprising HTTP transactions extracted from GitHub data and augmented with synthetic transaction data. The dataset facilitates reproducible research on many aspects of service-oriented computing. This paper discusses use cases for such a dataset and extracts a set of requirements from these use cases. We then discuss the design of GHTraffic, and the methods and tool used to construct it. We conclude our contribution with some selective metrics that characterise GHTraffic.

Keywords—HTTP; dataset; Web services; REST; benchmarking; reproducibility; service-oriented computing; service virtualisation; API; GitHub;

I. INTRODUCTION

Service-Oriented Computing (SOC) is a popular approach to facilitate the development of large, modular applications using diverse technologies. There is a range of technologies that have been used in SOC, with early attempts to establish standards around the SOAP [1] and WSDL [2] protocols. In recent years, *RESTful services* [3], a more lightweight approach closely aligned with the Hypertext Transfer Protocol (HTTP) [4], have become mainstream.

When using HTTP-based services, different parts of the application cooperate by sending and responding to HTTP requests, typically in order to access and manipulate resources. The ubiquitousness of the HTTP means that clients and servers can be easily implemented in a wide range of languages and deployed on many platforms. While this is useful in itself to architect and design large applications, this approach is now increasingly used to facilitate the development of product ecosystems around successful services. Examples include the APIs that can be used to access the services of Google, Facebook, Amazon, and Netflix.

This has created new challenges for both the research and the engineering community. Of particular interest are scalability, reliability, and security of (systems using and providing) services.

Like other fields of computing research, studies of SOC should aim for reproducibility [5], [6]. There is a wider push for reproducibility in computing research, with some disciplines now including research artefact evaluation as part of the standard peer-review process [7]. One way to facilitate the reproducibility and also the dissemination of research is the provision of standardised datasets. By using carefully

sourced and/or constructed datasets, research results become (1) easier to reproduce (2) comparable (i.e., results from different studies can be compared), and (3) generalisable (i.e., we can assume with a certain amount of confidence that results from a study can be applied to other data/systems that were not studied).

The purpose of this paper is to provide such a dataset, GHTraffic. We extract a base dataset from a successful, large-scale service, GitHub, by reverse-engineering API interactions from existing repository snapshots. We then enrich the dataset to include API interactions that cannot be recovered from snapshots, namely (non-state-changing) queries. This results in a large, rich, and diverse dataset. We argue that this can be used for a wide range of studies, including performance benchmarking and service virtualisation.

The rest of the paper is organised as follows. Use cases and requirements are discussed in detail in Section II, followed by an overview of related work in Section III. The construction of the dataset is discussed in Section IV. Section V and VI present the results of some measurements on the dataset and provide basic instructions how to obtain and use GHTraffic. We discuss threats to validity in Section VII and Section VIII concludes our contribution.

II. USE CASES AND REQUIREMENTS

A. Performance Benchmarking

Modern enterprise applications usually cooperate with a variety of software services such as Web servers, application servers, databases, proxies, and Web service clients to perform their functionalities. These services need to be tested in order to ensure that they are able to deal with large data and transaction volumes. In particular, performance benchmarking can provide useful indications about how services behave under different load conditions. A typical benchmarking tool generates synthetic workloads or replays recorded real-world network traffic in order to simulate realistic workloads and measures performance-related metrics, such as latency and throughput.

A dataset that is large, complex, and extracted from actual network traffic facilitates the benchmarking of such systems with non-trivial, realistic workloads.

B. Functional Testing

A standard dataset can also be employed for functional testing. For instance, it could be used to test a generic REST framework with a CRUD back-end provided by a (non-SQL) database. This would take advantage of the fact that such a dataset encodes a certain semantics, usually a combination of the standard HTTP semantics (for instance, the idempotency of certain methods) plus additional, application-specific rules and constraints. In other words, a suitable dataset can provide an *oracle* of correct system and service behaviour. As an example, consider an HTTP GET request to a named resource. This request should result in a response with 200 status code if there was an earlier successful POST transaction for the resource and no successful DELETE transaction between the POST and the GET, and 404 otherwise. A suitable dataset should contain transaction sequences to reflect such behavioural patterns.

C. Service Virtualisation

Service Virtualisation (SV) [8] is an approach to build a semantic model of a service based on recorded traffic. For instance, SV will try to simulate the behaviour of an actual service by generating responses using models inferred from recorded transactions. This inference is usually done by means of supervised machine learning. The main application is to test systems that heavily rely on external *black-box* services in isolation. This corresponds to (automatically created) mock objects popular in application testing [9].

A suitable standardised dataset could be used to test SV. It would provide an oracle of actual service behaviour to be used in order to assess the quality of inferred behaviour.

D. Requirements

From the use cases above, we extract the following set of requirements to guide the construction of GHTraffic.

- R1 **Large, yet manageable:** a good dataset should be of significant size to facilitate the use cases outlined and obtain results that are generalisable. However, this often conflicts with usability as experiments on large datasets are more difficult to set up and time-consuming. This can be addressed by providing several editions of different sizes.
- R2 **Ease of use:** a good dataset should be presented in a format that is easy to process and preferably includes scripts to facilitate the processing and analysis of data, and a schema that (formally) describes the format used to represent data.
- R3 **Reproducible, independent, and derived from principles:** a good dataset should not be produced ad-hoc, but extracted from real-world data or synthesised using a well-defined process unbiased by its use for one particular experiment.
- R4 **Current:** a good dataset should reflect the state-of-the-art use of HTTP-based services. While this is difficult

to assess in general, we argue that by extracting the dataset from the traffic of one of the most successful active Web services known for its excellent scalability and robustness, this can be achieved.

- R5 **Precise and following standards:** a good dataset should contain transactions that comply with the syntax and semantics of HTTP, and the service(s) used.
- R6 **Diverse:** a good dataset should support a wide set of HTTP features, such as various HTTP methods and status codes. In particular, it should go beyond the exclusive use of POST and GET requests which is a characteristic of older-generation Web applications designed for browser-based clients.

III. RELATED WORK

SPECweb2009 [10] is a standardised Web server benchmark produced by the Standard Performance Evaluation Corporation (SPEC). It is designed to evaluate a Web server ability to serve static and dynamic page requests. The benchmark comprises four distinct HTTP workloads to simulate three common types of consumer activities. The first workload is based on online banking, the second one is based on an e-commerce application, and the third one uses a scenario where support patches for computer applications are downloaded. All these workloads were developed by analysing log files of several popular Internet servers. The benchmark uses one or more client systems to generate HTTP workload for the server according to the specified workload characteristics. Each client sends HTTP requests to the server and then validates the response received. However, SPECweb2009 uses only HTTP 1.1 GET and POST requests and all of these requests are expected to result in responses with a 200 status code. In particular, server errors are communicated back to clients by generating error pages that return 200.

TPC Benchmark W (TPC-W) [11] from the Transaction Processing Council is a notable open-source Web benchmark specifically targeted at measuring the performance of e-commerce systems. TPC-W simulates the principal transaction types of a retail store that sells products over the Internet. The workload of this benchmark specifies emulated browsers that generate Web interactions which represent typical browsing, searching, and ordering activities. It creates different GET and POST requests for specific documents and collects performance data. All these requests are expected to result in responses with 200 status code.

Rice University Bidding System (RUBiS) [12] is another open-source Web benchmark. It is based on an online auction site, modelled after eBay. This benchmark implements the core functionality of an auction site, in particular, selling, browsing, and bidding. The benchmark workload relies on a number of browser emulators that mimic the basic network interactions of real Web browsers. Read and

write interactions are implemented using HTTP GET and POST requests.

DARPA dataset [13] by the MIT Lincoln Laboratory is a widely used evaluation dataset in intrusion detection research. There were three major releases. Each release contains tcpdump files carrying a wide variety of normal and malicious Web traffic simulated in a military network environment. These network packet dumps can be used as a direct input to packet filtering engines like Wireshark to extract sub-datasets which contain only HTTP request/response message traces as relevant to our work. In particular, an HTTP dataset which comprises 25,000 transactions can be obtained from DARPA 2000 tcpdump files. All these transactions used HTTP 1.0 GET requests and returned 200.

CSIC 2010 [14] by the Information Security Institute of Spanish Research National Council is another publicly available dataset, designed for the purpose of testing intrusion detection systems. It contains normal and anomalous HTTP 1.1 POST and GET requests targeting an e-commerce Web application. However, the dataset does not contain response data.

Another example of an HTTP message traces dataset is described in the work of Versteeg et al. [15], [16]. The authors used a relatively small dataset to study **Opaque SV**. This dataset consists of 1,825 request/response message traces collected through the Twitter REST API¹. It contains both POST and GET requests which return 200. However, the dataset does not cover all the HTTP methods such as PUT and DELETE. Besides, it is not publicly available for research purposes due to Twitter’s terms of service.

Table I summarises related benchmarks and datasets showing their request types, response codes, and transaction count. It is apparent that all these datasets only use a small fraction of the HTTP in terms of methods and status codes. They are somehow biased towards performance testing for older Web server where (static) pages are retrieved and in some cases created. They do not reflect the richness of modern Web APIs that take advantage of a much larger part of the HTTP.

Standard datasets have been widely used to support research in many other areas of computer science. For instance, the programming language and software engineering communities use datasets such as **DaCapo** [17] and **Qualitas Corpus/XCorpus** [18], [19] for benchmarking and empirical studies on source code. **Sourcerer** [20] is an infrastructure for large-scale collection and analysis of open source code. The Sourcerer database is populated with more than 2000 real-world open source projects taken from Sourceforge, Apache, and Java.net.

The machine learning community uses several standardised datasets. This includes **UCI Machine Learning Repository** [21] by the Center for Machine Learning and In-

Table I
OVERVIEW OF HTTP BENCHMARKS AND DATASETS

Name	HTTP Method	Response Code	Count
TPC-W	GET, POST	200	13,500,000
RUBiS	GET, POST	200	4,030,000
DARPA 2000	GET	200	25,000
CSIC 2010	GET, POST	-	36,000
Opaque SV	GET, POST	200	1,825

telligent Systems at the University of California, Irvine. It provides a collection of benchmark datasets which can be used for the empirical analysis of learning algorithms. Another example is **Kaggle**².

IV. METHODOLOGY

A. Input Data Selection

Over the past few years, GitHub³ has emerged as the dominant platform for collaborative software engineering. It contains a rich set of features to manage code-related artefacts, including commits, pull requests, and issues.

There are several clients provided by GitHub that can be used to access its services, including the Web front-end and the desktop app. Many developers also use the standard git command line interface (CLI). In order to facilitate the development of a rich product ecosystem to access its services, GitHub also provides a REST API⁴. This allows third parties to integrate GitHub services into their products. Examples include mobile clients as well as IDE and build tool integrations (plugins).

The GitHub REST API provides a rich set of services to create, read, update, and delete resources related to the core GitHub functionality. It employs a large subset of HTTP features for this purpose and is therefore semantically richer than the datasets discussed on Section III. Unfortunately, GitHub does not provide direct access to the recorded API interactions, so this information cannot be directly used for dataset construction.

An interesting use of the GitHub REST API for research purposes is GHTorrent [22]. This project uses the API to harvest information from repositories and stores that information by creating snapshots. These snapshots can then be downloaded and imported into a local MongoDB or MySQL database and queried. As of Feb. 02 2018, GHTorrent offers more than fifteen terabytes of downloadable snapshots. These snapshots have already been used in empirical studies, examples include Gousios et al. work on the pull-based software development model [23] and Vasilescu et al. work on the use of crowd-sourced knowledge in software development [24].

¹<https://developer.twitter.com/en/docs> [accessed Feb. 02 2018]

²<https://www.kaggle.com/datasets/> [accessed Feb. 02 2018]

³<https://github.com/> [accessed Feb. 02 2018]

⁴<https://developer.github.com/v3/> [accessed Feb. 02 2018]

While GHTorrent provides a static view on the state of GitHub at certain points in time, we are interested in a more dynamic view of how interactions of clients with the repository have created this state. The basic idea is to reverse-engineer the respective API interactions (i.e., HTTP transactions) by cross-referencing GHTorrent data with GitHub API functions. This has some obvious limitations. Firstly, we do not know whether all of these records were created via the REST API. They could have been created or altered using a different, or older version of the API, or via GitHub internal systems that bypass the API. We do not consider this as a significant limitation. As far as the data inferred transactions are concerned, this will only have an impact on the user-agent header. Secondly, the static data of the snapshots means that certain API interactions are not visible. This includes all read access (i.e., GET requests), requests that fail (e.g., a DELETE request resulting in a 404 response code will have no effect on the database), and shadowed requests (e.g., a successful PUT request followed by a successful DELETE request). To deal with those unobservable requests, we decided to augment the dataset with synthetic data.

B. Scope

GHTorrent collects a large amount of data on the terabyte scale. To make the data volume more manageable (R1), we decided to focus on a particular subset of GHTorrent, the issue tracking system. The issue tracking system itself references other entities⁵ of the overall data model. The respective model is depicted in Figure 1. It is a refined version of the relational schema used in GHTorrent⁶.

Issues reference multiple other entities such as comments, milestones, labels, and users. While it is important to model some of them to facilitate our use cases, we decided to limit this to user, milestone, and label data. In particular, while issue comments look like integral parts of the issue tracking system, they are modelled in a relational style as one-to-many relationships via back-references. This means that comments reference the issue they are associated with, but issues do not directly reference comments⁷.

The design of GHTraffic is driven by the use cases and the requirements derived from them. We wanted to construct a dataset that is large and diverse, and uses the features seen in modern Web services. This can be achieved by restricting the dataset to user, milestone, and label. Adding issue comments and other related data does increase the size further but does not add new features to the dataset. On the other hand, the increased size makes the dataset less manageable. As we will

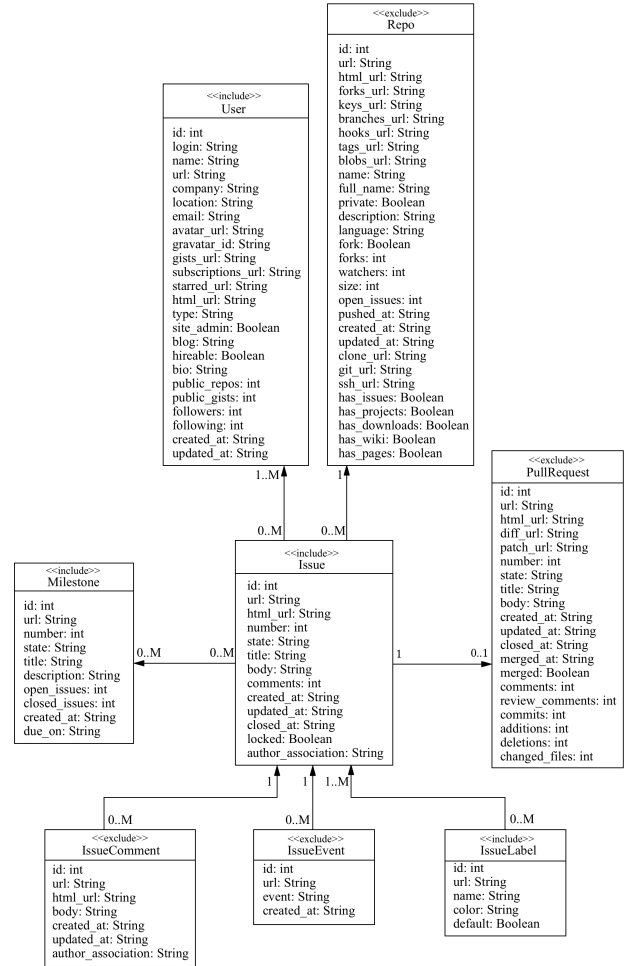


Figure 1. GitHub's data schema (UML 2.0). The stereotypes indicate which entities were included in the construction of GHTraffic dataset.

demonstrate in Section V, the dataset is already sufficiently large.

Data represented in different entities is usually inlined in data returned by API calls. This means that if issue information is returned via the API, the JSON representation of the issue contains information about the issue and a summary of the users, labels, and milestones associated with it. Part of this information are URLs that can be used to query the full information for the respective entity. We treat these URLs as external, un-resolved references in the sense that our dataset does contain transactions to create, modify, delete or query these resources. Note that the GitHub API already uses references to external resources for which resolution cannot be guaranteed, an example for this is the `gravatar_id` attribute pointing to a picture of the user provided by the `gravatar`⁸ service.

⁵Entity is used in this paragraph in the context of entity-relationship data modelling [25], as opposed to the use of entity in the context of HTTP as defined by [4, Sect. 7]

⁶<http://ghtorrent.org/relational.html> [accessed Feb. 02 2018]

⁷The JSON representation of an issue contains a field `comments`, but this contains only the number of comments for the respective issue. This number can then be used to construct comments queries.

⁸<https://pt.gravatar.com/> [accessed Feb. 02 2018]

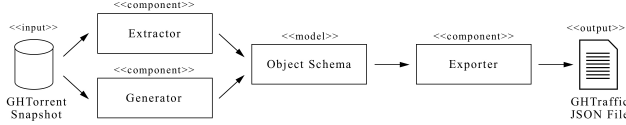


Figure 2. The processing pipeline

The GHTraffic dataset is based on the Aug. 04 2015 GHTorrent snapshot⁹. This is the largest release of issues MongoDB database dumps as of Feb. 02 2018.

C. Processing Pipeline

An abstract overview of the infrastructure used to create the GHTraffic dataset is shown in Figure 2. GHTorrent snapshots are accessed by two core components, the *Extractor* and the *Generator*, the purpose of both is to create HTTP transactions. While the extractor builds transactions directly from snapshot data, the generator infers synthetic transactions. In order to achieve this, it still needs access to the snapshot data. The reason for this is to get access to resource identifiers to be used in order to generate URLs. For instance, the generator creates queries, i.e., GET requests to query issues. If the respective resource names (i.e., issues ids) were generated randomly, almost all of those requests would fail with a 404. This is not very realistic: in practice, most GET requests would try to access existing resources and succeed. In order to model this, the generator needs to access the GHTorrent snapshot.

The transactions generated by both the extractor and the generator instantiate a simple model depicted in Figure 3. This model is implemented in Java, i.e., each transaction has a transient in-memory representation as a Java object when the dataset is created. At the centre of this model are HTTP transactions, basically request/response pairs.

At the end of the pipeline is an *Exporter* component that processes the transactions represented as Java objects and persists them by encoding/serialising using JSON. The structure of the JSON files produced is defined by JSON schemas [26]. Note that there are separate schemas for each HTTP method.

The implementation of the components discussed have some abstractions to facilitate alternative extraction, inference, and data representations. The overall processing model is lazy and stream-like, i.e., only a small number of records remain in memory at any time in order to make processing scalable.

Processing can be customised by employing data filters (predicates). Only records matching certain criteria are processed. The main use case for this is filtering by URL and

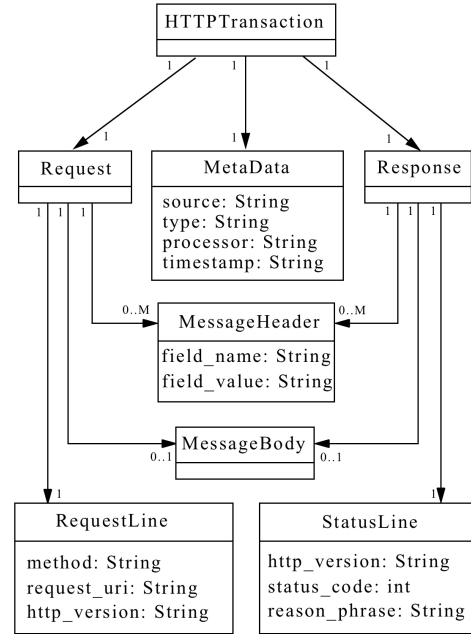


Figure 3. GHTraffic schema

here in particular by the project. This allows us to build different editions of the dataset with certain target sizes. While there is a potentially easier way of doing this by just restricting the number of records being processed and included, using filters has an inherent advantage. GitHub data is fragmented by project and by filtering it accordingly, we are able to extract transactions that manipulate the same resources, reflecting the same issue being created and updated. This was, we can obtain *coherent* subsets of the overall dataset that still reflect the service semantics derived from issue tracking workflows.

D. Extraction

Each data record has `created_at`, `updated_at`, and `closed_at` timestamps which enable us to trace lifecycle events of the issue. Using this data, the GHTraffic scripts produce transaction records. An overview of the process is shown in Figure 4. For instance, a POST transaction record is created in order to represent the creation of an issue at the time stipulated in the `created_at` attribute. The value is converted to the standard date format used by the HTTP [4, Sect. 3.3] and set as the value of `Date` header.

Both the request and the response used headers as specified in the GitHub API documentation. This is a mix of standard HTTP headers and API-specific headers with names starting with “x-”. In case the header values cannot be inferred from snapshot data, we use synthetic data. For example, we generate random token strings and use them as values for the `Authorization` headers. There is also a list of user agent strings to assign ran-

⁹The respective dump is available from <http://ghtorrent-downloads.ewi.tudelft.nl/mongo-full/issues-dump.2015-08-04.tar.gz> [accessed Feb. 02 2018]. The download size is 6,128 MB which results in a 48.29 GB database with 21,077,018 records after restoring.

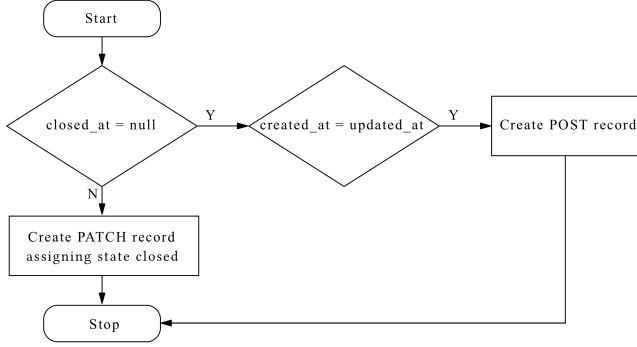


Figure 4. Extractor algorithm to process records

domly as the value of the User-Agent headers. Further, for the request body, the script extracts the values of the title, body, assignee, milestone, labels parameters from the snapshot record and encodes it in JSON as stipulated in the API. The response creation process is analogous. Most of the values for the JSON-encoded response body are filled out with data directly taken from the snapshot. Besides, the GHTraffic script assigns the `created_at` value to the `updated_at` field. Further, it explicitly specifies `closed_at: null, closed_by: null, state: open` and `locked: false`.

Every time a GitHub user updates an existing issue its `updated_at` timestamp gets renewed with the date and time of the update. Marking an issue as closed is a special type of update, as an issue is not deleted, but its status is changed to `close`. In order to extract PATCH transactions used to close issues, the script queries issues whose `closed_at` value is not null and only those are processed by the extractor. The request and response messages are formed by following the GitHub Issues API documentation. Particularly, the `closed_at` value is converted to the standard HTTP date format and set as the value of `Date` response header and the value of `closed_at`, assign to the `updated_at` field to set `closed_at` and `updated_at` columns' values same.

Besides, an update might be a changing the title of an issue, changing its description, specifying users to assign the issue to, etc. However, we could not extract exactly what input data was used for editing an issue, therefore, we did not generate such transaction types.

E. Synthesising Queries

Only successful POST and PATCH transactions can be constructed by reverse-engineering the GHTorrent snapshot. In order to generate additional transaction records such as queries and delete requests, we had to resort to using synthetic data. The aim of generating synthetic data is to mimic transactions concerning several other HTTP request methods that are covered by the API and requests that fail, which indicated by an error HTTP status code.

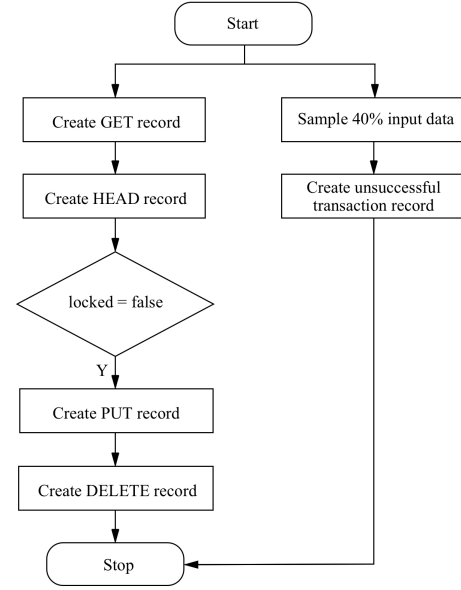


Figure 5. Algorithm to generate synthetic data

Figure 5 shows the process of synthetic data generation. The script generated GET and HEAD transactions for each record in the snapshot. The process is analogous to the process described in Section IV-C. However, the `Date` response header is set to the system date and time at which the request is formed. Similarly, PUT transactions for locking an issue are generated for records with `locked` value set to `false` and followed by DELETE transactions for unlocking the respective issues using the format described in the GitHub API.

Furthermore, the script produces unsuccessful transactions for all those HTTP methods by specifying requests:

- without authorisation token
- with badly formatted URL
- without request body
- with invalidly formatted JSON body

All the respective transactions have an error status code as defined in the API and are generated from a sample of 40% random records from the snapshot¹⁰. More specifically, a message explaining the error is added to the response body as specified in the GitHub API. For this purpose, we performed experiments on a toy project repository for creating synthetic data that closely resemble real-world representation as we found that certain aspects of the GitHub Issue API are undocumented. Additionally, we generated a small number of GET requests that returned 500 status code, in order to represent system failures.

¹⁰The generator component needs to use at least 40% of GHTorrent snapshot records in order to extract an adequate amount of unsuccessful transactions on particular projects

F. Data Representation and Meta-Data

The target format of the GHTraffic dataset is described by the UML class diagram as shown in Figure 3. HTTPTransaction is the base element of the model. A transaction contains a single Request and Response. Each message could have any specific number of MessageHeaders. Additionally, a MessageBody is used to represent data associated with a request/response. MetaData is used to provide some additional information about the transaction record. The source attribute is set to GHTorrent, specifying the source of information. The type attribute is set to either real-world or synthetic depending on whether the data was directly derived from a GHTorrent record or synthesised as described above. The processor is the name of the script used to generate the record, i.e., this is the fully qualified name of a Java class. Finally, the timestamp field holds date and time when the record was created.

The actual JSON format of the dataset is defined by a set of JSON schemas for each transaction type (i.e., for each HTTP method). For space limitations, we do not include those schemas, but they can be found in the repository, in *schemas* folder. These schemas comply with the JSON Schema draft 4 specification [27].

V. METRICS

The GHTraffic dataset comprises three different editions: Small (S), Medium (M), and Large (L). The S dataset includes HTTP transaction records created from *google/guava* [28] repository and takes up to 49.9 MB of disk space. Guava is a popular Java library containing utilities and data structures. It is a medium-sized large active project, and sourcing an edition from a single project has the advantage of creating a coherent dataset. The M dataset of size 345.2 MB includes records from the *npm/npm* [29]. It is the popular de-facto standard package manager for JavaScript. The L dataset contains 3.73 GB of data that were created by selecting eight repositories containing large and active projects on GitHub as of 2015, including *rails/rails* [30], *docker/docker* [31], *rust-lang/rust* [32], *angular/angular.js* [33], *twbs/bootstrap* [34], *kubernetes/kubernetes* [35], *Homebrew/homebrew* [36], and *symfony/symfony* [37].

Table II, III, and IV presents several metrics about the current status of these three datasets.

VI. ACCESSING AND USING GHTRAFFIC

The different editions of the GHTraffic dataset can be downloaded by using the following URLs¹¹:

- <https://zenodo.org/record/1034573/files/ghtraffic-S-1.0.0.zip>

¹¹The dataset is published on Zenodo [38]. It is a data repository platform hosted at the European Organization for Nuclear Research Data Center, which was specifically designed to provide long-term preservation of all forms of research output.

Table II
TRANSACTIONS PER HTTP METHOD

Method	S	M	L
POST	7,193	32,130	508,664
GET	3,117	22,692	344,474
PATCH	4,286	30,807	468,080
DELETE	2,341	16,457	246,180
PUT	3,662	15,945	238,115
HEAD	1,796	15,130	245,127

Table III
TRANSACTIONS PER HTTP RESPONSE CODE

Response Code	S	M	L
200	4,649	22,163	391,903
201	1,796	8,808	150,662
204	3,588	5,756	82,554
400	2,717	13,807	196,474
401	547	19,302	291,831
404	5,909	43,658	646,346
422	1,868	12,626	196,678
500	1,321	7,041	94,192

Table IV
TRANSACTIONS PER RECORD TYPE

Type	S	M	L
Real-world	2,853	13,355	241,241
Synthetic	19,542	119,806	1,809,399

- <https://zenodo.org/record/1034573/files/ghtraffic-M-1.0.0.zip>
- <https://zenodo.org/record/1034573/files/ghtraffic-L-1.0.0.zip>

We also provide access to the scripts used to generate GHTraffic, including a VirtualBox image with a pre-configured setup. Note that due to the use of random data generation these scripts will produce slightly different datasets at each execution. Using the scripts, users can modify the configuration properties in *config.properties* file in order to create a customised version of GHTraffic dataset for their own use. The *readme.md* file included in the distribution provides further information on how to build the code and run the scripts. Scripts can be accessed by cloning the repository <https://bitbucket.org/tbhagya/ghtraffic.git> or by downloading the pre-configured VirtualBox image from <https://zenodo.org/record/1034573/files/ghtraffic-artifact-1.0.0.zip>.

VII. THREATS TO VALIDITY

As depicted in Table IV, the size of synthetic data exceeds the size of data extracted from the snapshot by a factor of nine. This leaves the possibility that GHTraffic does not reflect realistic workloads. To mitigate this threat, we ensured that the request/response format for all these transaction types was sampled and validated using a toy GitHub repository. While this does not mean that these

transactions have actually occurred, it guarantees that they are syntactically and semantically correct. The representation of the transactions has information about whether they are synthetic or not, and users of GHTraffic can use this to completely remove or reduce the ratio of synthetic data by applying filters.

We acknowledge that GHTraffic was generated from a 2-year-old snapshot of GHTorrent. As noted earlier, this design decision was made to produce a dataset large enough to facilitate the use cases described, but still manageable with typical resources available to researchers and practitioners. We also provide access to the scripts used to generate GH-Traffic, and users can utilise these scripts in order to generate customised versions from newer instances of GHTorrent if needed.

VIII. CONCLUSION

In this paper, we have described the GHTraffic dataset suitable for experimenting on various aspects of service-oriented computing. It is derived from reverse-engineering a GHTorrent snapshot according to the GitHub Issue API specification. We hope that this dataset will find uses in many areas of research.

In future work, it would be interesting to extend this by adding similar datasets from other service providers, using similar processes and tools.

REFERENCES

- [1] D. Box *et al.*, “Simple object access protocol (SOAP) 1.1,” 2000, accessed Feb. 02 2018. [Online]. Available: <https://www.w3.org/TR/2000/NOTE-SOAP-20000508/>
- [2] E. Christensen *et al.*, “Web services description language (WSDL) 1.1,” 2001, accessed Feb. 02 2018. [Online]. Available: <https://www.w3.org/TR/wsdl>
- [3] R. T. Fielding and R. N. Taylor, *Architectural styles and the design of network-based software architectures*. University of California, Irvine Doctoral dissertation, 2000.
- [4] R. T. Fielding *et al.*, “Hypertext transfer protocol–HTTP/1.1 (RFC2616),” 1999, accessed Feb. 02 2018. [Online]. Available: <https://tools.ietf.org/html/rfc2616>
- [5] R. D. Peng, “Reproducible research in computational science,” *Science*, 2011.
- [6] C. Collberg and T. A. Proebsting, “Repeatability in computer systems research,” *Communications of the ACM*, 2016.
- [7] S. Krishnamurthi and J. Vitek, “The real software crisis: Repeatability as a core value,” *Communications of the ACM*, 2015.
- [8] J. Michelsen and J. English, *Service Virtualization: Reality Is Overrated*. Apress, 2012.
- [9] T. Mackinnon, S. Freeman, and P. Craig, “Endo-testing: unit testing with mock objects,” 2000.
- [10] “SPECweb2009,” accessed Feb. 02 2018. [Online]. Available: <https://www.spec.org/web2009/>
- [11] W. D. Smith, “TPC-W: Benchmarking an ecommerce solution,” 2000.
- [12] “RUBiS: Rice University Bidding System,” 2013, accessed Feb. 02 2018. [Online]. Available: <http://rubis.ow2.org/>
- [13] “DARPA Intrusion Detection Data Sets,” 2000, accessed Feb. 02 2018. [Online]. Available: <https://www.ll.mit.edu/ideval/data/>
- [14] C. Gimnez, A. P. Villegas, and G. Maranon, “CSIC 2010,” 2010.
- [15] S. C. Versteeg *et al.*, “Opaque service virtualisation: a practical tool for emulating endpoint systems,” in *Proceedings ICSE ’16*. ACM, 2016.
- [16] S. C. Versteeg, J. S. Bird, N. A. Hastings, M. Du, and J.-D. Dahan, “Entropy weighted message matching for opaque service virtualization,” 2017, US Patent 9,582,399.
- [17] S. M. Blackburn *et al.*, “The DaCapo benchmarks: Java benchmarking development and analysis,” in *Proceedings OOPSLA’06*. ACM, 2006.
- [18] E. Tempero *et al.*, “The Qualitas Corpus: A curated collection of Java code for empirical studies,” in *Proceedings APSEC’10*. IEEE, 2010.
- [19] J. Dietrich, H. Schole, L. Sui, and E. Tempero, “XCorpus—An executable Corpus of Java Programs,” 2017.
- [20] S. Bajracharya *et al.*, “Sourcerer: a search engine for open source code supporting structure-based search,” in *Proceedings OOPSLA ’06*. ACM, 2006.
- [21] A. Frank, “UCI machine learning repository,” 2010, accessed Feb. 02 2018. [Online]. Available: <http://archive.ics.uci.edu/ml/index.php>
- [22] G. Gousios, “The GHTorrent dataset and tool suite,” in *Proceedings MSR’13*, 2013.
- [23] G. Gousios, M. Pinzger, and A. v. Deursen, “An exploratory study of the pull-based software development model,” in *Proceedings ICSE’14*. ACM, 2014.
- [24] B. Vasilescu, V. Filkov, and A. Serebrenik, “StackOverflow and GitHub: Associations between software development and crowdsourced knowledge,” in *Proceedings SocialCom’13*. ASE/IEEE, 2013.
- [25] P. P.-S. Chen, “The entity-relationship model—toward a unified view of data,” *TODS*, 1976.
- [26] A. Wright, “JSON Schema: A Media Type for Describing JSON Documents,” Technical Report. Internet Engineering Task Force, Tech. Rep., 2016.
- [27] F. Galiegue and K. Zyp, “JSON Schema: core definitions and terminology draft-zyp-json-schema-04,” *Working Draft*, 2013.
- [28] K. Boumillon and J. Levy, “Guava,” 2010, accessed Feb. 02 2018. [Online]. Available: <https://github.com/google/guava>
- [29] “npm,” accessed Feb. 02 2018. [Online]. Available: <https://github.com/npm/npm>
- [30] “Rails,” accessed Feb. 02 2018. [Online]. Available: <https://github.com/rails/rails>
- [31] “The Moby Project,” accessed Feb. 02 2018. [Online]. Available: <https://github.com/docker/docker>
- [32] “The Rust Programming Language,” accessed Feb. 02 2018. [Online]. Available: <https://github.com/rust-lang/rust>
- [33] “AngularJS,” accessed Feb. 02 2018. [Online]. Available: <https://github.com/angular/angular.js>
- [34] “Bootstrap,” accessed Feb. 02 2018. [Online]. Available: <https://github.com/twbs/bootstrap>
- [35] “Kubernetes,” accessed Feb. 02 2018. [Online]. Available: <https://github.com/kubernetes/kubernetes>
- [36] “Homebrew (Legacy),” accessed Feb. 02 2018. [Online]. Available: <https://github.com/Homebrew/homebrew>
- [37] “Symfony,” accessed Feb. 02 2018. [Online]. Available: <https://github.com/symfony/symfony>
- [38] “Zenodo,” accessed Feb. 02 2018. [Online]. Available: <https://zenodo.org/>