Automated Server Testing: an Industrial Experience Report

Chao Peng, Yujun Gao, Ping Yang
ByteDance
Beijing, China
{pengchao.x, gaoyujun, yangping.cser}@bytedance.com

***Abstract*—A server API bug could have a huge impact on the operation of other servers and clients relying on that API, resulting in service downtime and financial losses. A common practice of server API testing inside enterprises is writing test inputs and assertions manually, and the test effectiveness depends largely on testers' carefulness, expertise and domain knowledge. Writing test cases for complicated business scenarios with multiple and ordered API calls is also a heavy task that requires a lot of human effort. In this paper, we present the design and deployment of SIT, a fully automated server interface reliability testing platform at ByteDance that provides capabilities including (1) traffic data generation based on combinatorial testing and fuzzing, (2) scenario testing for complicated business logics and (3) automated test execution with fault localisation in a controlled environment that does not affect online services. SIT has been integrated into the source control system and is triggered when new code change is submitted or configured as scheduled tasks. During the year of 2021, SIT blocked 434 valid issues before they were introduced into the production system.**

***Index Terms*—server testing, traffic record and replay, automated testing**

I. Introduction

Server-side APIs are publicly exposed endpoints that accept requests from clients or other servers and return responses, usually in the format of JSON or XML. Modern server APIs usually adhere to the REpresentational State Transfer (REST) [1] or Remote Procedure Call (RPC) [2] architecture styles.

As server APIs serve as the protocol that allows servers and clients to communicate and share resources, they are playing a key role in heterogeneous software system integration [3]. Therefore, server testing is a critical task in modern software quality assurance: a server fault could cause an internal impact on other services and an external impact that influences user experience, resulting in financial and user losses.

In recent years, testing techniques have been widely studied for client-side applications (web and mobile apps) [4]–[11] and web services based on SOAP (Simple Object Access Protocol) and WSDL (Web Services Description Language) [12], [13]. However, few of the existing work focused on testing RESTful and RPC APIs [3].

Previously, the quality assurance (QA) for server APIs at ByteDance was mostly based on artificially crafted test cases and automated test case execution (as shown in Figure 1), which has the following limitations:

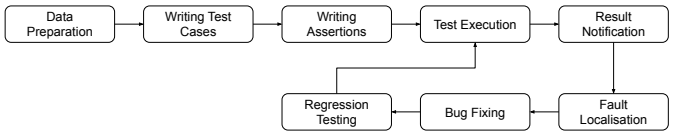

Fig. 1. Workflow of manual server testing

a) High test case writing and maintenance cost: A common practice of manual testing for server interfaces is to manually write test cases according to the interface specification. However, for a business scenario that is composed of multiple interfaces, it is time-consuming for testers to understand the description and business logic of the scenario, gather all interfaces related to the scenario and write test cases based on different combinations and orders of related interfaces. RPC interface testing, in particular, requires test cases to comply with preconditions such as importing IDLs (interactive data language) of the interface under test. When the interface changes, affected test cases need to be corrected manually. In addition, screening, filtering and upgrading test cases in regression testing mostly rely on the experience of testers or fixed rules to decide which test cases should be selected, removed or rewritten. Such practises are usually time-consuming.

b) Low test effectiveness: For reasons including rapid development and release of new products, there may not be sufficient interface tests. In addition, manual testing usually focuses on most important functionalities, resulting in low overall code coverage and ineffective quality assurance.

c) High effort required for test assertion: Test oracle checking is usually implemented by assertions using JSON Schema, which is coarse in granularity. Moreover, writing customised assertions is also costly and inflexible. When specifications of server interfaces change, the test oracle also needs to be updated manually.

Given these limitations, we propose SIT (Server Interface Testing), for scalable and effective server API testing. The goal of SIT is to help developers to uncover and fix bugs before they are introduced into the production system. We report on the deployment of SIT on the source control management system of ByteDance to support testing server APIs at a large

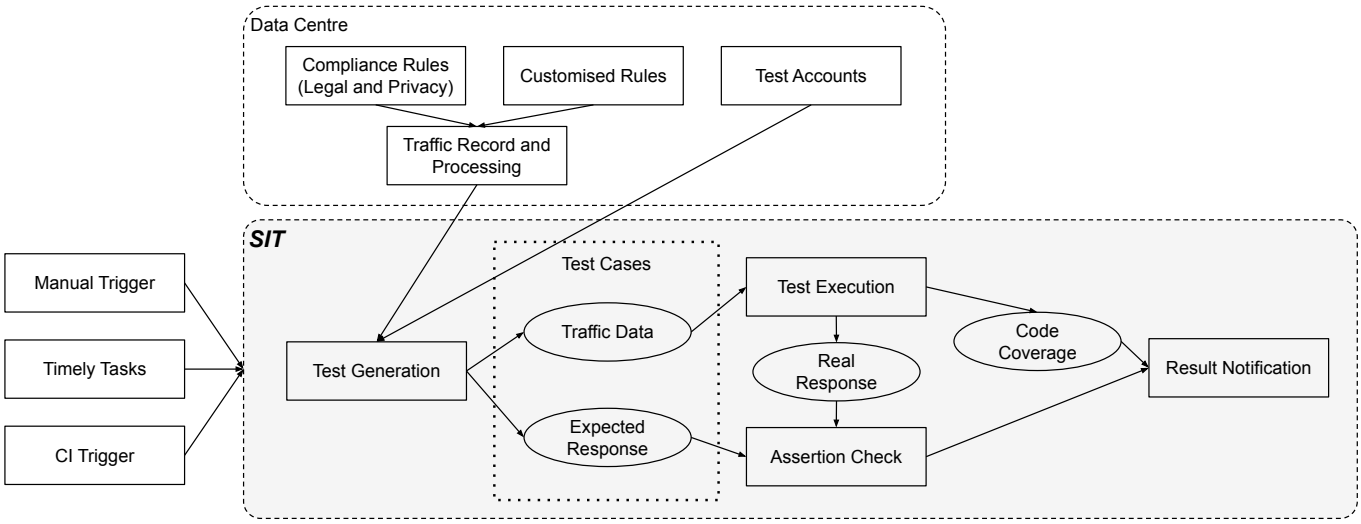

Fig. 2. Architecture and workflow of SIT

scale: 5 business lines (products)[1] with millions of lines of code, serving billions of end users over the year of 2021. Specifically, SIT uncovered 550 unique bugs. 78.9% of them were marked as valid issues and resolved by developers.

II. Our Approach

We present SIT, an automated server reliability testing framework that provides, a) automated traffic data[2] generation for single API and business scenarios with multiple APIs and b) test execution in a controlled sandbox and 3. fault localisation.

The overall architecture and workflow of SIT is presented in Figure 2. SIT can be triggered in three ways: a) manual triggering via a webpage frontend, b) scheduled tasks set up by developers and testers, and c) CI (continuous integration) when new code change is submitted or new server version is due to be deployed. The inputs to SIT include:

1) Anonymised and configured traffic data complying with legal and privacy rules are used as the input data pool, as described in Section II-A.
2) Customised rules can be defined by developers and testers to generate desired test inputs for specific fields, as described in Section II-B.
3) Test accounts are used to setup special test environment that require user logging in.

As for reliability testing, the test oracle (assertion) of SIT is that the API under test should not crash, no runtime exceptions or error log info are thrown during test execution and no failure return code is caught after execution.

In the rest of this section, we discuss details of the design and implementation of SIT's working phases.

[1]For confidential reasons, we are regretful that we cannot provide names of these products.

[2]Traffic data is used interchangeably with test inputs in the context of server testing in this paper.

A. Compliant and Anonymised Data Collection

SIT collects anonymised RESTful (via HTTP) and RPC traffic data from the TPC (Transmission Control Protocol) layer, including request/response headers and bodies. The data anonymisation process is performed by security and privacy departments under legal and privacy rules and the data accessible to SIT is always anonymised data.

After removing duplicated data, SIT stores APIs in the same order of the online traffic and corresponding anonymised or customised data as the input data pool. This makes sure that the generated data is more in line with actual user scenario, with higher diversity to balance test coverage and maintenance cost.

B. Support for Data Customisation

The anonymisation process replaces original data from sensitive fields by asterisk signs (*). When special values are needed to test specific business logic, SIT supports input data customisation: testers can select a set of data fields of interest and define input generation rules such as regular expressions for SIT to generate fake but meaningful data for these fields, or provided a list of candidate values for these fields.

C. Test Generation

Instead of direct traffic replay, SIT uses pair-wise combinatorial testing to generate new test cases from the stored traffic data. The rationale behind choosing pair-wise combination is that most failures are triggered by combinations of a few values and covering pairs could reveal more faults but keeping the number of test cases smaller than all combinations [14].

SIT samples the API traffic and collects all possible unique values for each field of the API and then generates pairs of the these values covering all combinations of values from any two fields. To generate test inputs for the API, SIT keeps picking values from different combinations until all pairs are exhausted.

For newly developed and changed APIs without applicable existing input candidates, we use fuzz testing to generate random test inputs based on the interface specification. In this case, SIT is still able to catch crashing bugs and runtime exceptions such as time-out and server-no-response exceptions during test execution.

D. Test Execution and Fault Localisation

Testing tasks can be triggered by developers manually, scheduled tasks, or CI changes. For manual tasks, the developer can specify the list of APIs and test configurations including customised data generation rules. Similarly, scheduled tasks are also configured with an API list and customisations with desired time interval. CI changes-triggered tasks only focus on APIs that have code or dependency changes by the merge requests or new service releasing tasks. All tasks are performed using test accounts in a controlled sandbox environment where services are compiled with source code instrumentation for line coverage measurement.

When a failure is detected during or after test execution, the original code change (merge request) or attempt to publish the service is blocked by SIT. In addition to this, fault localisation is also performed by SIT to help developers save debugging time. SIT extracts stack traces from test execution logs to pin-point to the code line that introduces the runtime failure, which is probably the cause of the bug. The original error information, call stack and collected code line are sent back to the developer for reference.

E. Scenario Testing

In the industrial context, single API testing is not suitable for business scenarios that requires testing multiple APIs in order with back-and-forth dependencies. Taking online livestreaming as an example: scenario-related APIs account for a high proportion of all interfaces, such as APIs for different types of live rooms, interactions between the host and the audience, interactive games, sending gifts, etc.

SIT proposes a scenario-based test case construction scheme based on a call link directed acyclic graph (DAG) diagram:

1) Collects the call chain of APIs from the TPC layer.
2) Extracts meta data information including user ID, start time and end time.
3) Calculates largest common request sequences from different flows using the LCS-Length-Table-Formulation algorithm.
4) Infers scenario-based interface parameter relationship and constructs the DAG diagram based on the request source, query body and other parameters, through sorting and building linked relationships.

Figure 3 shows a simplified example of the API call chain for livestreaming. The user needs to create live room and start livestreaming. During the livestreaming, the user can query the list of available live rooms and call another host to livestream together (one host will also appear in another's live room).

SIT first identifies the three APIs and their order to start livestreaming and a pair of these three APIs serve as the prerequisite for live calls. In addition, the order and parameter relationship of caller.call(callee) to make live call and callee.accept(caller) to accept live calls is also recorded by SIT. When constructing test cases for this scenario, SIT will choose two test accounts and maintain the caller and callee relationship for corresponding APIs. A possible test cases generated by SIT is illustrated in Figure 4.

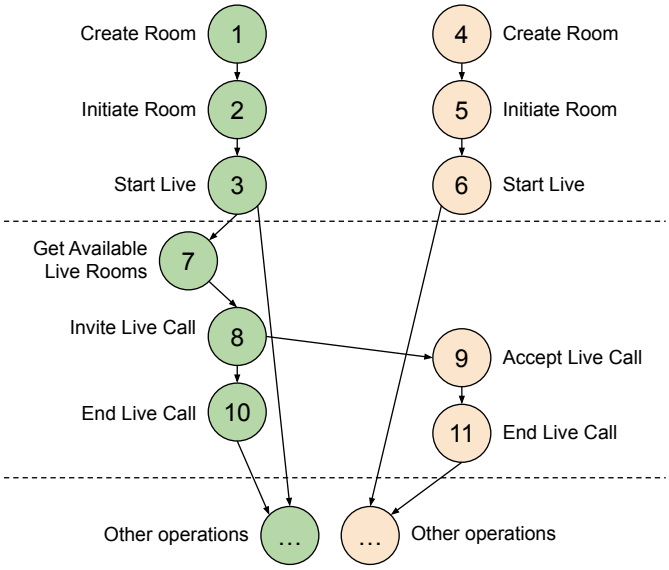

Fig. 3. Example livestreaming scenario

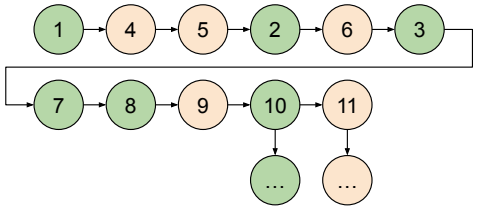

Fig. 4. A possible test case for livestreaming

III. Application In Practice

We report the industrial deployment of SIT on 5 business lines (products) during the year of 2021. During this period, SIT performed averagely 0.2 billion reliability testing tasks per week and 92.2% of these tasks were successfully finished. Causes to failed tasks include wrong parameter types and denied access to test accounts. We summarise some representational server bug types that were detected by SIT:

- **Faulty route configuration.** The route configuration updated is no longer compatible to devices running clients with older versions, resulting in wrong server response.
- **Missing function parameter.** Not enough parameters are used when making a function call. If the client is an old version, it would also send requests with less parameters.
- **Parameter type mismatch.** The type of the parameter in the requested API is changed in the server side but different from the client.

IV. Related Work

In this section, we discuss existing work on test input generation, split into client testing and server API testing.

Automated client testing is a well-studied area [15]. Existing work focuses on GUI (Graphical User Interface) state abstraction [16] and automated GUI event generation based on model-based testing [6], [7] and machine learning [17]–[19]. At ByteDance, there are several automated testing tools for mobile apps via GUI input generation, including Fastbot [5] for app reliability testing and CAT [9] that focuses on GUI elements impacted by code changes. These tools are effective in uncovering app crashes and ANR (Application Not Responding) problems in an industrial context. Unlike these client testing tools, SIT focuses on testing server applications based on traffic data generation and automated test assertions.

To test server APIs, several tools are widely used to write API requests manually and catch responses for testers to validate, including Postman [20], Apifox [21], etc. Automated API testing has received less attention than Service-Oriented Architectures (SOA) testing techniques [12], [13], although RESTful APIs are modern alternatives to SOAs [3]. Several tools and frameworks are proposed to test server APIs but require formal notations or manually defined models of API under test [22]–[24]. These techniques are hard to be adopted by the industry as testers have to invest time into getting familiar with them and update formal notations and models when changes are made to services and APIs. In terms of automated traffic generation, FAUSTA [25] is the closest to our work which also generates traffic data to test large-scale industrial server applications. FAUSTA synthesizes initial random test inputs based on service specification and predicts new inputs using Markov chain. However, to test services with dependencies, such as answering a call should be preceded by receiving a call, FAUSTA requires developers to define guided flows (orders required to call services), which hinders automation.

V. Conclusion and Future Work

This paper presents a framework named SIT for server API reliability testing at scale with traffic generation, scenario testing, test execution in a sandbox and fault localisation capabilities. The deployment of SIT at ByteDance helped developers to uncover and fix 434 unique issues before they were introduced into production systems during the year of 2021. In the future, we plan to guide test generation based on the coverage achieved to further improve test coverage and fault finding capabilities. We are also interested in combing SIT with existing static analysis tools to balance the completeness and time cost of both techniques.

References

[1] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[2] R. Srinivasan, "RPC: Remote procedure call protocol specification version 2," 1995.

[3] A. Martin-Lopez, "AI-driven web API testing," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, 2020, pp. 202–205.

[4] F. Y. B. Daragh and S. Malek, "Deep GUI: Black-box GUI input generation with deep learning," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 905–916.

[5] T. Cai, Z. Zhang, and P. Yang, "Fastbot: A multi-agent model-based test generation system," in *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*, 2020, pp. 93–96.

[6] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, "Practical GUI testing of Android applications via model abstraction and refinement," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 269–280.

[7] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based GUI testing of Android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 245–256.

[8] J. Sun, "SetDroid: detecting user-configurable setting issues of Android apps via metamorphic fuzzing," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2021, pp. 108–110.

[9] C. Peng, A. Rajan, and T. Cai, "CAT: Change-focused Android GUI testing," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 460–470.

[10] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for Android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 94–105.

[11] N. Alshahwan, X. Gao, M. Harman, Y. Jia, K. Mao, A. Mols, T. Tei, and I. Zorin, "Deploying search based software engineering with Sapienz at Facebook," in *International Symposium on Search Based Software Engineering*. Springer, 2018, pp. 3–45.

[12] M. Bozkurt, M. Harman, and Y. Hassoun, "Testing and verification in service-oriented architecture: a survey," *Software Testing, Verification and Reliability*, vol. 23, no. 4, pp. 261–313, 2013.

[13] G. Canfora and M. D. Penta, "Service-oriented architectures testing: A survey," in *Software Engineering*. Springer, 2007, pp. 78–105.

[14] M. Pezzè and M. Young, *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.

[15] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein, "Automated testing of Android apps: A systematic literature review," *IEEE Transactions on Reliability*, vol. 68, no. 1, pp. 45–66, 2018.

[16] Y.-M. Baek and D.-H. Bae, "Automated model-based Android GUI testing using multi-level GUI comparison criteria," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 238–249.

[17] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Humanoid: A deep learning-based approach to automated black-box Android app testing," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1070–1073.

[18] T. A. T. Vuong and S. Takada, "Semantic analysis for deep q-network in Android GUI testing." in *SEKE*, 2019, pp. 123–170.

[19] J. Eskonen, J. Kahles, and J. Reijonen, "Automating GUI testing with image-based deep reinforcement learning," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 2020, pp. 160–167.

[20] Postman. (2022). [Online]. Available: https://www.postman.com

[21] Apifox. (2022). [Online]. Available: https://www.apifox.cn

[22] S. K. Chakrabarti and R. Rodriquez, "Connectedness testing of RESTFul web-services," in *Proceedings of the 3rd India software engineering conference*, 2010, pp. 143–152.

[23] C. Benac Earle, L.-Å. Fredlund, Á. Herranz, and J. Mariño, "Jsongen: a quickcheck based library for testing JSON web services," in *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang*, 2014, pp. 33–41.

[24] P. Lamela Seijas, H. Li, and S. Thompson, "Towards property-based testing of RESTFul web services," in *Proceedings of the twelfth ACM SIGPLAN workshop on Erlang*, 2013, pp. 77–78.

[25] K. Mao, T. Kapus, L. Petrou, A. Hajdu, M. Marescotti, A. Löscher, M. Harman, and D. Distefano, "Fausta: Scaling dynamic analysis with traffic generation at WhatsApp," in *2022 15th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2022.