Log-driven Testing of Microservice Systems with Transformers

Raffaele Della Corte[a], Roberto Pietrantuono[a,b], Stefano Russo[a]
[a]*DIETI - Università degli Studi di Napoli Federico II*
Via Claudio 21, 80125, Naples, Italy
[b]*CINI - Consorzio Interuniversitario Nazionale per l'Informatica*
M.S. Angelo, Via Cinthia, 80126 Naples, Italy
{raffaele.dellacorte2, roberto.pietrantuono, stefano.russo}@unina.it

***Abstract*—Regression testing enhances software reliability by detecting regressions in new versions. Regression test suites often lack awareness of real-world product/service usage, potentially leading to undetected faults and ineffective testing scenarios.**

We propose `LogTesT`, a transformer-based approach that learns from event logs and system traces to automatically generate service invocation sequences that mimic observed system behaviors. These sequences enhance regression test suites by exposing past real execution patterns. A preliminary experimentation on a realistic benchmark demonstrates its potential.

***Index Terms*—logging, tracing, transformers, testing**

I. Introduction

Modern software development involves frequent updates and short release cycles, especially for service-based software like microservices. Regression testing is crucial for such systems to detect breakages in new releases. Significant research exists on regression testing [1], particularly in continuous integration environments, focusing on Test Selection & Prioritization (TS&P) [2]–[4]. Machine Learning is employed to use historical data for predicting test case outcomes, thus helping to select/prioritize likely-to-fail tests [5]–[7]. These strategies often rely on static data sources from past executions, such as test and/or code metrics from past tests execution extracted from versioning systems. Despite good results, valuable real-world data from continuous integration and deployment (which is usually available in continuous integration/deployment and DevOps through continuous monitoring) is often underutilized, mining the representativeness of conducted tests. This data, reflecting actual user interactions and real environments, is more informative than development-time data.

This study introduces `LogTesT` (Log-driven Testing with Transformers), a microservices testing strategy that uses operational log data to understand system behaviors and enhance regression test suites with representative test cases. The strategy involves learning the relationship between system behaviors (e.g., occurrence of 500 HTTP status codes, high response times) and service invocation sequences using a fine-tuned generative Large Language Model (LLM). During inference, the model generates new invocation sequences based on input features from logs, such as sequences more likely to lead to high 500 HTTP status occurrences or high latency. Preliminary results on a widely-used realistic benchmark for microservices, `TrainTicket` [8] shows how `LogTesT` can expose a remarkable higher failure rate compared to a recent baseline technique, `uTest` [9], as well as a higher class coverage.

Our approach differs from the state-of-the-art in that: (i) it uses operational data to identify tests predicting real-world outcomes across future releases, leveraging continuous feedback in DevOps and microservices in an *ex vivo* configuration; (ii) while it can prioritize or select tests, it primarily aims to augment the regression suite with new field-informed tests.

The proposal extends our previous work [10] by: (i) including a test cases generation step, where the outcome of the LLM is used to generate test cases; (ii) applying the proposal to an actual system, i.e., `TrainTicket`; (iii) comparing the proposal with an existing approach, i.e., `uTest`.

II. Tests generation strategy

An overview of the proposal is shown in Fig. 1. The strategy uses two types of input: *event logs* and *system trace data*. **Event logs** are a byproduct of the software system execution, since they are naturally emitted during operation. Event logs are sequences of text lines – typically stored in log files – reporting about the runtime behavior of a system [11], including entries highlighting the occurrence of system failures. **System trace data** accounts for various information about the use of the services composing a system, allowing to track source-destination of the services invocation, along with the response codes, latency, etc. A *pre-processing stage* is performed to map each trace, representing a particular system invocation, with the event log entries corresponding to that invocation. A portion of the obtained dataset is then used to *fine-tune* an LLM for the task of generating sequence of service invocations from event logs. The fine-tuned LLM is then queried to generate new sequences of service invocations given raw logs observed in the field, representing a behavior that the tester aims to reproduce. The obtained sequences are then used for *test cases generation*.

Pre-processing. Given a set of logs and traces, pre-processing aims to map each trace $\mathcal{T}_i$ (identified by a trace ID i) with the event logs generated during the invocations captured by that trace. To this aim, this stage: (*i*) extracts the

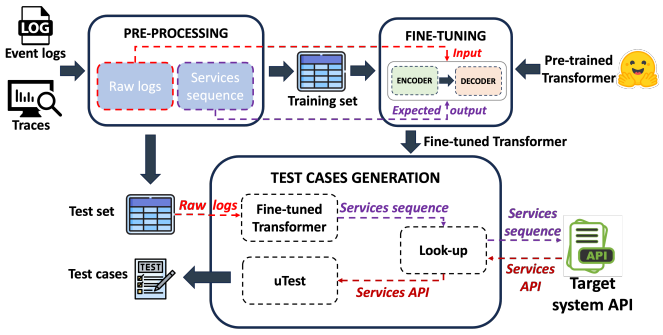

Fig. 1. Transformer-based strategy for regression test cases generation.

time window of each trace (i.e., the time difference between the last and the first event of the trace) as well as the *sequence of services* involved in the invocation represented by the trace; (*ii*) groups together the log entries that fall in the extracted time window (hereinafter *raw log*); (*iii*) labels the raw log entries with the extracted services sequence.

These steps produce a dataset encompassing, for each trace ID, *i)* the services sequence involved in the trace; *ii)* the raw log entries mapped with the trace, and other data extracted from the trace and logs, i.e., latency (in seconds) of the entire interaction described by the trace, the list of endpoints involved in the trace, the number of log entries for each log severity level (e.g., WARNING, ERROR) and the amount of `http` response codes (e.g., 200, 500) – all these features. Finally, the dataset is split into Training (50%) and Test (50%) sets, used for model fine-tuning and test cases generation, respectively.

Fine-tuning. We use a T5-small model, a reduced version of T5, a transformer-based sequence-to-sequence model. T5-small is an encoder-decoder model architecture, which is pre-trained on the colossal clean crawled corpus (C4) dataset and on a mixture of unsupervised and supervised tasks, such as natural language inference, question answering, etc. T5 models each problem into a text-to-text format. The model receives a text input including the desired task to be performed as the prefix; its output is in text format. During this stage we fine-tuned the pre-trained model (retrieved from the Hugging Face transformer library) on the task of generating sequence of service invocations from raw logs. This task is modeled as a *translation* task, where the input is represented by the raw logs, i.e., the interesting log entries observed in operation, while the expected output is the services sequence, i.e., the sequence of invocations generating those log entries.

The fine-tuning has been performed by using the Training set obtained from the pre-processing stage. The training set is further divided in 90% for training and 10% for testing, with the latter used for the evaluation of the model performance after each epoch of training. We trained our model for overall 18 epochs, with the *Levenshtein distance* used to evaluate the model performance after each epoch. In our context, this metric allows evaluating the distance between two sequences.

Test cases generation. The fine-tuned transformer is used to generate the test cases suggested by the raw logs, which can be used to enhance regression test suites. The test cases generation leverages `uTest` [9], a pairwise combinatorial strategy that makes use of OpenAPI specifications of microservices to generate test cases, execute them and gather the results.

During the test cases generation step each entry of the Test set is analyzed via three main steps: (*i*) given an entry of the Test set, its raw logs field is submitted to the fine-tuned transformer, which generates the related services sequence; (*ii*) the obtained sequence is used for a look-up operation into the Services API of the target system, which aims to find the API of the services included within the sequence provided by the transformer; (*iii*) the obtained Services API are then provided to uTest to generate test cases encompassing valid input values.

After generating the test cases, uTest executes them on the target system, providing details like amount of succeeded and failed tests, coverage metrics. In this study, we analyze both the *failure rate*, i.e., the number of failures exposed by the tests, and the *coverage (2 classes) metric* defined in [12].

TABLE I
COMPARISON OF RESULTS: `LogTesT` VS. UTEST.

Technique	Test suites	Test cases	Tested paths	Failures	Average coverage	Failure rate
`LogTesT`	1,000	88,158	22	49,370	57.2%	56.0%
uTest	1	4,678	1,187	1,027	50.1%	21.9%

III. EVALUATION

To validate the proposed strategy, we apply `LogTesT` to `TrainTicket`, a well-known open source microservice architecture benchmark, composed of 41 microservices [8].

Our setup allows the collection of both event logs and traces generated by each microservice in the TrainTicket system. We collect the data during a 60 minutes execution, during which TrainTicket is stimulated through a publicly available load generator [13] based on `Locust` [14]. The collected data encompasses around 1M log entries and 150k trace events. We apply the described pre-processing stage to the collected raw data, obtaining a pre-processed dataset with 2k entries. Those entries are then equally split in Training and Test sets.

The fine-tuning stage is applied using the Training set as described in Section II, obtaining the T5-small transformer tuned for our translation task, generating services sequence from raw logs. The generated Test set and the fine-tuned transformer are then used for test cases generation, along with the `swagger` json files describing the TrainTicket API (used with uTest), as described in Section II and depicted in Fig. 1.

We evaluate the results obtained from the application of `LogTesT` on TrainTicket, which are reported in the first row of TABLE I. We obtained 1,000 test suites, one for each entry of the Test set, with overall 88,158 test cases, targeting 22 API paths. Each test suite has been executed on TrainTicket by means of uTest, obtaining a 57.2% coverage (2 class) and a 56% failure rate (with 49,370 tests failed with the 500 code).

To compare `LogTesT` with a baseline, we apply uTest on TrainTicket, providing it with the whole API specification without any guidance from `LogTesT`. This leads to the generation of a single test suite, encompassing 4,678 test cases, which address 1,187 API paths. The second row of TABLE

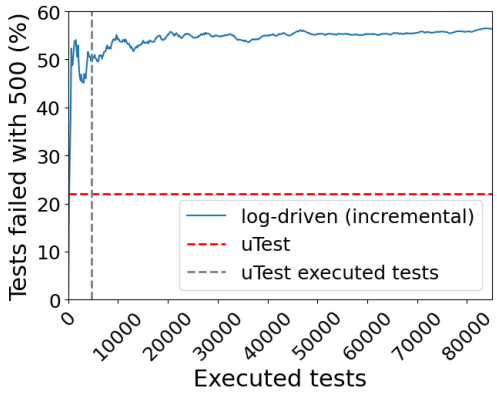

Fig. 2. Failure rate.

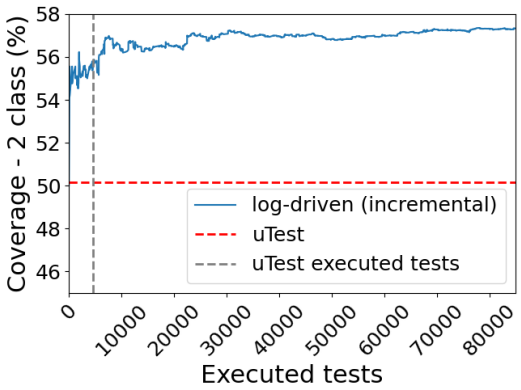

Fig. 3. Coverage - 2 classes.

I reports the uTest results. After the test suite execution, we obtained 50.1% coverage (2 classes) and 21.9% failure rate.

Comparing the results, it can be easily noted that `LogTesT` exhibited a remarkable higher failure rate with respect to uTest. Fig. 2 shows the comparison of the failure rate for both `LogTesT` (blue line) and uTest (dashed lines). It is important to note, that uTest generated a single test suite; therefore, in Fig. 2 just one sample is depicted for uTest (i.e., the cross between the gray and red dashed lines). Instead, for `LogTesT` a sample for each test suite is considered, reporting the incremental number of executed tests and failures triggered during the whole testing campaign. uTest obtained around 21.9% of failed tests with 500 after executing 4,678 tests, while `LogTesT` obtained around 51% of failed tests with 500 after the same amount of tests. Therefore, `LogTesT` has more than doubled the number of triggered failures with respect to uTest, running the same amount of tests. We also compare the coverage values, which are shown in Fig. 3. It can be noted that `LogTesT` outperforms uTest also in terms of coverage. uTest obtained a coverage of 50.1% for 2 classes, after executing 4,678 tests. `LogTesT` obtained a coverage of 57.2% for 2 classes, after running the same amount of tests.

Overall, the evaluation provides some promising results, since after running the same amount of tests, the proposal exposed a remarkable higher failure rate when compared to uTest, as well as a higher class coverage. This suggests the `LogTesT` can lead to discover more issues in the target system at the same cost compared with uTest. Another aspect to consider is the amount of paths addressed by the two approaches. uTest addressed more than 1.1k paths of the TrainTicket API, while `LogTesT` generated test cases addressing just 22 paths. This is possible since the selection of the services to test has been guided by the logs collected during operation.

IV. Conclusion

The paper proposed `LogTesT`, an LLM-based testing strategy for microservice systems that uses event logs to understand failing system behaviors and enhance regression test suites with representative test cases. The proposal leverages a transformer-based model to learn from event logs and system traces to generate service invocation sequences that mimic observed failing behaviors. `LogTesT` has been applied on the TrainTicket subject, exhibiting promising results in terms of failure rate and coverage, outperforming uTest.

Acknowledgment

This work was supported in part by the European Union's Horizon 2020 program under the Marie Skłodowska-Curie grant agreement No 871342 "uDEVOPS", and by the SERENA-IIoT project, which has been funded by EU - NGEU, Mission 4 Component 1, CUP J53D23007090006, under the PRIN 2022 (MUR - *Ministero dell'Università e della Ricerca*) program (project code 2022CN4EBH).

References

[1] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[2] E. Knauss et al. Supporting continuous integration by code-churn based test selection. In *IEEE/ACM 2nd International Workshop on Rapid Continuous Software Engineering (RCoSE)*, pages 19–25. IEEE, 2015.

[3] M. Gligoric, L. Eloussi, and D. Marinov. Practical regression test selection with dynamic file dependencies. In *2015 International Symposium on Software Testing and Analysis (ISSTA)*, pages 211–222. ACM, 2015.

[4] B. Busjaeger and T. Xie. Learning for test prioritization: An industrial case study. In *24th ACM SIGSOFT Int'l Symposium on Foundations of Software Engineering (FSE)*, pages 975–980. ACM, 2016.

[5] Y. Pang, X. Xue, and A. S. Namin. Identifying Effective Test Cases through K-Means Clustering for Enhancing Regression Testing. In *12th International Conference on Machine Learning and Applications (ICMLA)*, pages 78–83. IEEE, 2013.

[6] A. R. Lenz, A. Pozo, and S. R. Vergilio. Linking software testing results with a machine learning approach. *Engineering Applications of Artificial Intelligence*, 26(5):1631–1640, 2013.

[7] A. Bertolino, B. Miranda, A. Guerriero, R. Pietrantuono, and S. Russo. Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration. In *Proc. of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, pages 1–12. ACM, 2020.

[8] X. Zhou et al. Benchmarking microservice systems for software engineering research. In *Proc. of the 40th Int'l Conference on Software Engineering: Companion Proceeedings*, page 323–324. ACM, 2018.

[9] L. Giamattei, A. Guerriero, R. Pietrantuono, and S. Russo. Assessing Black-box Test Case Generation Techniques for Microservices. In A. Vallecillo, J. Visser, and R. Pérez-Castillo, editors, *Quality of Information and Communications Technology*, pages 46–60, Cham, 2022. Springer International Publishing.

[10] R. Della Corte and R. Pietrantuono. Towards log-driven testing through transformers: A preliminary study. In *2024 19th European Dependable Computing Conference (EDCC)*, pages 103–106, 2024.

[11] M. Cinque, R. Della Corte, and A. Pecchia. An empirical analysis of error propagation in critical software systems. *Empirical Software Engineering*, 25(4):2450–2484, 2020.

[12] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés. Test coverage criteria for restful web apis. In *Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, A-TEST 2019, page 15–21. ACM, 2019.

[13] Locust load generator for trainticket. https://github.com/rajibhossen/ts-locust-load-generator. Accessed: 2025-06-06.

[14] Locust. https://locust.io/. Accessed: 2025-06-06.