Challenges in Regression Test Selection for End-to-End Testing of Microservice-based Software Systems

Daniel Elsner
Daniel Bertagnolli
Alexander Pretschner
firstname.lastname@tum.de
Technical University of Munich
Munich, Germany

Rudi Klaus
rudi.klaus@t-systems.com
T-Systems International
Munich, Germany

ABSTRACT

Dynamic regression test selection (RTS) techniques aim to minimize testing efforts by selecting tests using per-test execution traces. However, most existing RTS techniques are not applicable to microservice-based, or, more generally, distributed systems, as the dynamic program analysis is typically limited to a single system. In this paper, we describe our distributed RTS approach, `microRTS`, which targets automated and manual end-to-end testing in microservice-based software systems. We employ `microRTS` in a case study on a set of 20 manual end-to-end test cases across 12 versions of the German COVID-19 contact tracing application, a modern microservice-based software system. The results indicate that initially `microRTS` selects all manual test cases for each version. Yet, through semi-automated filtering of test traces, we are able to effectively reduce the testing effort by 10–50%. In contrast with prior results on automated unit tests, we find method-level granularity of per-test execution traces to be more suitable than class-level for manual end-to-end testing.

CCS CONCEPTS

• **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

Software testing, regression test selection, microservice architectures, end-to-end testing, manual testing

ACM Reference Format:
Daniel Elsner, Daniel Bertagnolli, Alexander Pretschner, and Rudi Klaus. 2022. Challenges in Regression Test Selection for End-to-End Testing of Microservice-based Software Systems. In *IEEE/ACM 3rd International Conference on Automation of Software Test (AST '22), May 17–18, 2022, Pittsburgh, PA, USA*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3524481.3527217

1 INTRODUCTION

Regression testing is a software testing activity that is regularly performed to ascertain that changes have not inadvertently altered previous system behavior [15]. Yet, with increasingly large test suites and shorter software (delivery) life-cycles, running all tests after each change is often too costly [4, 30]. To reduce the costs of regression testing, regression test selection (RTS) [7, 13, 21–23, 31, 33] techniques have been extensively studied since the 1970s [5].

Traditional RTS techniques identify a subset of tests by comparing new code changes with per-test code dependencies. Collecting these per-test code dependencies can either be achieved through *static* or *dynamic* program analysis at the level of basic-blocks [11, 21], functions/methods [6, 17, 32, 33], classes/files [7, 8, 12, 13], modules [24], or combinations thereof [25, 28, 31]. In dynamic RTS techniques, these per-test code dependencies can also be interpreted as per-test execution traces. Since most techniques are incapable of collecting dependencies across system boundaries, they mainly target unit testing. Yet, in microservice-based systems, or, more generally, highly distributed systems, checking for functional correctness is no longer limited to individual units or modules, but especially requires anticipation of interface and interaction bugs [16, 33]. Hence, existing RTS techniques are not directly applicable, as these systems need to be tested through integration or end-to-end tests that operate across service or system boundaries.

While there are a few studies on RTS for web applications and services [17, 19, 33], they have several limitations in the context of microservices: First, existing RTS techniques targeting web applications only instrument server code for tracing tests [17, 19]. However, excluding client code is problematic in microservice-based systems, where (rich) web or mobile client applications often contain large parts of the business logic and orchestrate calls to different microservices. Second, even though protocol- and language-agnostic distributed tracing approaches are provided by observability frameworks such as `OpenTelemetry`[1], existing RTS techniques are implemented for monolingual systems [33] and specific communication protocols, such as Hypertext Transfer Protocol (HTTP) [17, 19]. Third, high instrumentation overhead of RTS techniques may preclude their applicability in practice [2, 3, 18], but existing studies lack analyses on the introduced instrumentation overhead during service startup and test execution. Finally, due to the complexity of automating end-to-end tests, these tests are often performed manually [10, 19]. Yet, no prior RTS study investigates RTS for manual end-to-end tests in microservice-based systems.

[1]`OpenTelemetry`: https://opentelemetry.io/

In this paper, we propose microRTS, a distributed RTS technique suitable for manual or automated end-to-end testing in microservice-based software systems. microRTS is implemented on top of well-established distributed tracing infrastructure and Java bytecode manipulation libraries, to enable automated instrumentation of arbitrary Java microservices at runtime. We evaluate the influence of several implementation aspects on the instrumentation overhead in the open-source microservice benchmark application TeaStore [29].

We further present a case study that was conducted together with our industry partner T-Systems[2] on a subset of the manual end-to-end test suite on 12 software versions of the German COVID-19 contact tracing application, Corona-Warn-App (CWA), a modern microservice-based software system. Since the CWA accesses backend microservices through a rich mobile client (we only consider Android), we design microRTS to collect per-test execution traces from mobile clients as well as microservices, leading to more complete traces. In contrast with prior results on automated unit tests [7, 12], we find class-level granularity of test traces to be too coarse grained for RTS of manual end-to-end tests in this context, essentially leading to *retest-all*. With traces at method-level granularity, microRTS initially still selects all test cases across the 12 studied versions. However, after closer inspection of the reasons behind a test being selected, we find that the manual tests are commonly imprecisely specified. For instance, while all manual tests cover the CWA start screen, most of them do not test any start screen functionality. Consequently, by semi-automatically filtering out irrelevant parts of the test traces, microRTS can exclude 10–50% of tests.

To foster more research on RTS for integration or end-to-end testing in microservice-based systems, we discuss challenges and elaborate on experiences when implementing and applying microRTS in a real-world context.

2 RELATED WORK & STATE-OF-PRACTICE

Among the many existing RTS studies already referenced, we consider the following to be most relevant for the context of this work:

Nakagawa et al. [19] propose a method-level RTS technique for manual end-to-end tests for Java web applications. By sending a custom header with each HTTP request to the server, a tester's browser can be mapped to accessed methods, assuming a one-to-one mapping of HTTP requests and Java Virtual Machine (JVM) threads. The results of the industrial case study on two web applications indicate that it is likely that all tests need to be executed for large modifications or changes to common code parts. As the proposed RTS technique is not able to trace more than one web server, it is yet unsuitable in a microservice context.

Long et al. [17] propose the RTS tool WebRTS, which supports collecting file-level per-test execution traces across multiple instances of a web server. It is designed for Java web applications with server-side page rendering and is limited to communication using the HTTP protocol which confines the applicability to a small subset of microservice-based systems. Unfortunately, although publicly available, WebRTS lacks an adequate user documentation and further relies on JVM bytecode instrumentation from an external library that lacks English documentation[3]. For this reason, we were neither able to fully comprehend, nor to apply their tool in preliminary experiments on Java microservices.

Zhong et al. [33] propose the RTS technique TestSage that targets web service testing at Google. TestSage supports C++ services and performs function-level instrumentation using XRay[4]. While TestSage reduces the testing time by 34%, it does not support parallel test tracing and is limited to homogeneous C++ web services.

In summary, we are not aware of any prior work that investigates the potential and challenges of RTS in the context of manual end-to-end testing for microservice-based systems.

3 DISTRIBUTED TEST SELECTION

When a microservice-based software system is tested in an end-to-end fashion, restarting the system under test (SUT) between each test case is not always feasible. This is because such systems often involve tens or even hundreds of services. Hence the startup process is expensive and time-consuming [33]. Consequently, when collecting per-test execution traces, we need to take into account that multiple tests are executed on the same deployed service instance, either sequentially or in parallel. We thus require segmentation of collected traces according to the tests' execution time frame and link covered code parts to the test that executed them [17]. In the following, we describe how microRTS collects precise distributed per-test execution traces for end-to-end tests and performs change-based test selection.

While microRTS currently supports microservices written in languages that target the JVM (e.g., Java, Kotlin) and instrumentation of Android mobile clients, the concepts are agnostic to the actually used programming language or platform. We chose Java as our case study subject is written in Java and Kotlin (see Sec. 5).

3.1 Distributed Tracing

The core principle behind *distributed tracing* is *context propagation*: A *context* contains (at least) a unique identifier that identifies a *trace* and is transferred in and across services in a distributed system [27]. The trace thereby encapsulates all requests related to an individual transaction. To enable context propagation, clients and services are instrumented to be able to create, transfer, and access context information embedded into requests. Consider Fig. 1 that depicts an instrumented service, where context information is extracted from inbound requests and injected into outbound requests. Furthermore, *trace points* can be inserted into the instrumented service that define actions such as attaching metadata to the context.

The implementation of the required code instrumentation for the middleware (e.g., HTTP client libraries) can be performed using well-established, polyglot distributed tracing and observability frameworks, such as OpenTracing[5] or OpenTelemetry. microRTS uses OpenTelemetry to automatically instrument Java microservices by attaching a Java Agent [1] that performs Java bytecode instrumentation at runtime. Thereby, the instrumented microservice will extract the context from inbound requests and we can link the context with custom code instrumentation as described next.

[2]T-Systems is one of the largest European information technology service providers: https://www.t-systems.com/

[3]JVM-SANDBOX: https://github.com/alibaba/jvm-sandbox

[4]LLVM XRay: https://llvm.org/docs/XRay.html

[5]OpenTracing: https://opentracing.io/

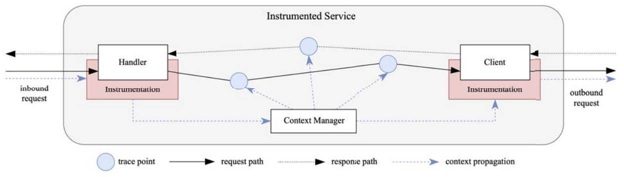

Figure 1: Context propagation in an instrumented service (inspired by Shkuro [26])

3.2 Code Instrumentation

We have described why the SUT typically cannot be restarted after each test in microservice-based systems. Therefore, if several tests are executed one after another, we need a code instrumentation that also takes into account already created objects from previously executed tests [17]. Thus, similar to pre-existing RTS techniques [19, 33], microRTS instruments microservices (and clients) at method-level granularity rather than class-level, as only instrumenting JVM class loading or object creation would miss if tests call methods of already existing objects. However, since we aim to investigate benefits of test trace granularity more closely (see Sec. 5), microRTS offers to control if test traces are stored (and aggregated) at method- or class-level granularity. Furthermore, existing approaches for collecting test traces differ regarding the strategy to export information about covered methods (i.e., *coverage probes*) during runtime [14, 20]. microRTS offers to export coverage probes directly after they fire or in batches, and supports writing coverage probes into a file or sending them via Transmission Control Protocol (TCP) sockets to a central *trace collector*.

We implement the method-level code instrumentation using a Java Agent and the ByteBuddy bytecode manipulation library[6]. Thereby, whenever a method is entered, the instrumentation stores a coverage probe in the *coverage tracer*. A coverage probe contains the method's signature, the name of the surrounding class, and the current context's trace identifier. Depending on how microRTS is configured, the coverage probes are written into a file or sent via TCP to the trace collector, either one-by-one or in batches. Fig. 2 illustrates how coverage information is collected from instrumented microservices. Additionally, the client is connected to a *test listener* that is responsible for maintaining the context for test cases in *test logs*, to later on link coverage probes to test cases. In Sec. 5, we describe how we implemented a test listener for the Android client of the CWA. microRTS further implements compile-time instrumentation of Android mobile clients using the AndroidBuddy library[7], as in contrast to the JVM, Android does not allow runtime instrumentation.

3.3 Test Selection

For change-based test selection, microRTS uses the changeset since the last time a test suite was executed from the version control system (VCS), together with the collected method-level test traces from the last test execution. microRTS then parses the .java and .kt files from the changeset and computes (1) the set of changed classes and (2) the set of changed methods by comparing checksums of each method's source code, similar to existing RTS techniques [31]. Using these two sets and the test traces, microRTS then determines affected tests through class- or method-level selection. In Sec. 5, we describe the effects of using class- and method-level selection in a manual end-to-end testing context.

[6]ByteBuddy: https://bytebuddy.net/
[7]AndroidBuddy: https://github.com/LikeTheSalad/android-buddy

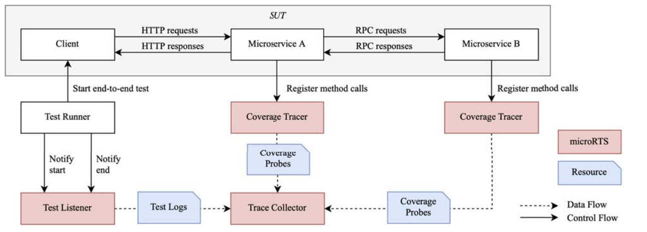

Figure 2: Overview of the microRTS test tracing architecture

4 EFFICIENCY EVALUATION

In order to analyze the efficiency of microRTS's instrumentation, we conduct experiments on the TeaStore microservice benchmark application. We thereby strive to answer the following research questions (RQs):

- **RQ$_1$**: How much instrumentation overhead does microRTS introduce at system startup and during testing?
- **RQ$_2$**: How does the granularity of coverage probes and their export strategy affect instrumentation overhead?

4.1 Experimental Setup

TeaStore is an open-source microservice reference application used by researchers to analyze and test novel techniques for microservice-based systems [9, 29]. We conduct our experiments on version *v1.4.0* of the TeaStore, consisting of 6 Java microservices that we orchestrate using Docker-Compose. Since TeaStore currently only contains unit tests, we implemented a set of 23 automated end-to-end tests using the testing framework Cypress. Our test suite covers each feature of the application by at least one test case and was merged by the project's maintainers into the main code base[8].

To execute our experiments, we instrument each microservice and the Cypress test runner with microRTS and run all 23 test cases with different (1) coverage probe collection granularity (method- or class-level) and (2) coverage probe export strategies (in-memory, file, or socket export). We repeat the experiments 30 times to account for variations in the runtimes and measure the average system startup and testing runtime for all configurations of microRTS and compare them to executions without any instrumentation (NoInst).

4.2 Discussion of Results

RQ$_1$: Startup and Runtime Instrumentation Overhead. The results of the comparative analysis between microRTS and NoInst show that the performance impact of microRTS is more significant during services' startup (+67.5%) than during testing (+18%). By re-running the tests using *only* OpenTelemetry's instrumentation, we see that most of the overhead stems from OpenTelemetry, which already adds 40% to the startup and 10.5% to the testing runtime.

[8]TeaStore Pull Request: https://github.com/DescartesResearch/TeaStore/pull/203

We further observe that the overhead caused by `microRTS` is significantly higher during the first test compared to the mean overhead. The reason is inherent to dynamic bytecode instrumentation, which transforms Java bytecode files on the fly when they are first loaded by the JVM `ClassLoader`.

RQ_2: Granularity and Export Strategies of Coverage Probes. The granularity at which `microRTS` is configured to export coverage probes does not significantly affect the instrumentation overhead: method-level granularity adds roughly 1.2% overhead compared to class-level granularity in total test suite execution time. The reason why storing and exporting coarser-grained class-level coverage probes is not far more efficient is that `microRTS` still needs to instrument all methods as explained in Sec. 3.2.

Regarding the chosen coverage probe export strategy, we find that in-memory is the fastest strategy because it does not export probes until service shutdown. Perhaps surprisingly, the file export strategy only adds around 1% of runtime overhead when compared to in-memory, despite the I/O overhead. Finally, although used in prior studies [20, 33], the socket strategy has a comparatively high runtime overhead of 6.7% compared to in-memory.

5 CASE STUDY: CORONA-WARN-APP

To evaluate `microRTS` in a real-world microservice-based system, we conduct a case study on the manual end-to-end test suite of the Corona-Warn-App (CWA) to answer **RQ_3**: How much manual end-to-end testing effort reduction can be achieved using `microRTS`?

5.1 Experimental Setup

The CWA is the official German Covid-19 contact tracing application, based on a decentralized, microservices architecture. The source code of the microservices and mobile clients is open-sourced and available on Github[9], easing reproducibility and extension of our case study. Service providers such as our industry partner T-Systems are responsible for end-to-end regression testing of new versions and releases of the CWA. Therefore, they use a manual regression test suite that is not publicly available. As currently the test cases for release testing are selected manually, automated and systematic tool-support through `microRTS` can be beneficial.

For our experiments with `microRTS`, we prepare a suitable testing environment: We (1) instrument seven CWA microservices, (2) instrument the mobile client (only Android), (3) patch or mock requests to external services such as Google's Exposure Notification System, as they can exclusively be used by authorized official health agencies, and (4) orchestrate all instrumented services with `Docker-Compose`, as neither the staging, nor the production environment configuration are publicly available. We then execute 20 manual end-to-end test cases provided by T-Systems for version *2.5.1* of the CWA. These tests are still executable without limitations in our experimental setup and we only instrument the seven services required for the provided test cases. We implement a small Android sidecar application, where we can start and stop a manual test case, which internally initializes and closes a tracing context.

To evaluate the potential of `microRTS`, we determine the set of selected tests on all (12) CWA release candidate versions between *2.5.1* and *2.6.1*. We include these release candidates to gain insights on how shorter testing cycles affect RTS results. Furthermore, we compare class- and method-level RTS as their effectiveness is not unequivocal in existing RTS literature [7, 17, 19].

[9]Corona-Warn-App (CWA): https://github.com/corona-warn-app

5.2 Discussion of Results

RQ_3: Test Effort Reduction. The initial results show that RTS at method- and class-level already selects all 20 tests for the first version, namely between *2.5.1* and *2.6.0-RC0*. As a result, RTS between *2.5.1* and all other subsequent versions has the same outcome.

To understand the underlying reasons, we investigate the causes for selection: First, 100% of the test selections for all versions have been caused by changes in the Android client, both using class- and method-level RTS. This highlights the importance of instrumenting client code as well. Second, all test cases include various shared covered methods, which originate primarily from the home screen where all tests start or end according to the test case specifications. Yet, surprisingly, only 9 out of the 20 test cases effectively verify functionality of the home screen. To determine if the other 11 test cases would have been selected even if their traces started on their respective sub-page, we proceed by refining the test traces. During this refinement step, we remove all covered methods that are associated to the home screen for the 11 test cases. Using class-level RTS nothing changes: all tests are selected. However, when using method-level RTS the selected tests for *2.6.0-RC0* are reduced by 50%, only 10 out of 20 tests are selected; for the subsequent versions and the next release *2.6.1*, up to 18 out of 20 tests are selected (90%).

Hence, we can conclude that the effectiveness of our RTS approach for manual end-to-end testing is highly dependent on the precision of test specifications. Through semi-automated pruning of test traces using domain knowledge, we are able to exclude up to 50% of tests, but only when using fine-grained method-level RTS. Our results confirm findings from prior RTS research on manual testing, where RTS effectiveness was limited with shared covered code parts in test traces or with large changesets [19].

6 CONCLUSION

In this paper, we introduce `microRTS`, a dynamic RTS technique for microservice-based systems. By combining established distributed tracing infrastructure with code instrumentation, `microRTS` collects per-test execution traces at method- or class-level across services and clients, thereby enabling test selection for automated or manual end-to-end tests. We further present a case study on RTS for manual end-to-end tests in the CWA, a real-world microservice-based system. Our initial results show that if manual tests are specified rather coarse-grained, `microRTS` can not provide any benefits over retest-all. However, when pruning per-test execution traces using domain knowledge, we are able to exclude up to 50% of tests. These findings confirm prior research on manual testing and show that manual end-to-end testing of microservice-based systems is particularly intricate to optimize.

ACKNOWLEDGMENTS

This work was funded by the German Federal Ministry of Education and Research (BMBF), grant SOFIE 01IS18012B. The responsibility for this article lies with the authors.

REFERENCES

[1] 2017. Java Agent API. https://docs.oracle.com/javase/9/docs/api/java/lang/instrument/package-summary.html

[2] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the International Symposium on the Foundations of Software Engineering*. 235–245. https://doi.org/10.1145/2635868.2635910

[3] Daniel Elsner, Florian Hauer, Alexander Pretschner, and Silke Reimer. 2021. Empirically Evaluating Readily Available Information for Regression Test Optimization in Continuous Integration. In *Proceedings of the International Symposium on Software Testing and Analysis*. 491–504. https://doi.org/10.1145/3460319.3464834

[4] Kurt Fischer, Farzad Raji, and Andrew Chruscicki. 1981. A Methodology for Retesting Modified Software. In *Proceedings of the National Telecommunications Conference*. 1–6.

[5] Kurt F. Fischer. 1977. A test case selection method for the validation of software maintenance modifications. In *Proceedings of International Computer Software and Applications Conference*. 421–426.

[6] Ben Fu, Sasa Misailovic, and Milos Gligoric. 2019. Resurgence of Regression Test Selection for C++. In *Proceedings of the International Conference on Software Testing, Verification and Validation*. 323–334. https://doi.org/10.1109/ICST.2019.00039

[7] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Ekstazi: Lightweight Test Selection. In *Proceedings of the International Conference on Software Engineering*. 713–716. https://doi.org/10.1109/icse.2015.230

[8] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *Proceedings of the International Symposium on Software Testing and Analysis*. 211–222. https://doi.org/10.1145/2771783.2771784

[9] Johannes Grohmann, Martin Straesser, Avi Chalbani, Simon Eismann, Yair Arian, Nikolas Herbst, Noam Peretz, and Samuel Kounev. 2021. SuanMing: Explainable Prediction of Performance Degradations in Microservice Applications. In *Proceedings of the International Conference on Performance Engineering*. 165–176. https://doi.org/10.1145/3427921.3450248

[10] Roman Haas, Daniel Elsner, Elmar Juergens, Alexander Pretschner, and Sven Apel. 2021. How can manual testing processes be optimized? Developer survey, optimization guidelines, and case studies. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1281–1291. https://doi.org/10.1145/3468264.3473922

[11] Mary Jean Harrold, Alessandro Orso, James A. Jones, Tongyu Li, Maikel Pennings, Saurabh Sinha, Ashish Gujarathi, Donglin Liang, and S. Alexander Spoon. 2001. Regression test selection for Java software. *ACM SIGPLAN Notices* 36, 11 (2001), 312–326. https://doi.org/10.1145/504311.504305

[12] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An Extensive Study of Static Regression Test Selection in Modern Software Evolution. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 583–594. https://doi.org/10.1145/2950290.2950361

[13] Owolabi Legunsen, August Shi, and Darko Marinov. 2017. STARTS: STAtic regression test selection. In *Proceedings of the International Conference on Automated Software Engineering*. 949–954. https://doi.org/10.1109/ase.2017.8115710

[14] Claire Leong, Abhayendra Singh, Mike Papadakis, Yves Le Traon, and John Micco. 2019. Assessing Transition-Based Test Selection Algorithms at Google. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. 101–110. https://doi.org/10.1109/icse-seip.2019.00019

[15] Hareton K.N. Leung and Lee White. 1989. Insights into regression testing. In *Proceedings of the International Conference on Software Maintenance*. 60–69.

[16] Hareton K.N. Leung and Lee White. 1990. A study of integration testing and software regression at the integration level. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Press, Silver Spring, MD, 290–301. https://doi.org/10.1109/icsm.1990.131377

[17] Zhenyue Long, Zeliu Ao, Guoquan Wu, Wei Chen, and Jun Wei. 2020. WebRTS: A Dynamic Regression Test Selection Tool for Java Web Applications. In *Proceedings of the International Conference on Software Maintenance and Evolution*. 822–825. https://doi.org/10.1109/ICSME46990.2020.00102

[18] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive Test Selection. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. 91–100. https://doi.org/10.1109/ICSE-SEIP.2019.00018

[19] Takao Nakagawa, Kazuki Munakata, and Koji Yamamoto. 2019. Applying modified code entity-based regression test selection for manual end-To-end testing of commercial web applications. In *Proceedings of the International Symposium on Software Reliability Engineering Workshops*. 1–6. https://doi.org/10.1109/ISSREW.2019.00033

[20] Raphael Noemmer and Roman Haas. 2020. An Evaluation of Test Suite Minimization Techniques. In *Software Quality: Quality Intelligence in Software and Systems Engineering*, D Winkler, S Biffl, D Mendez, and J Bergsmann (Eds.). Vol. 371. Springer,, 51–66. https://doi.org/10.1007/978-3-030-35510-4_4

[21] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling regression testing to large software systems. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 241–251. https://doi.org/10.1145/1029894.1029928

[22] Gregg Rothermel and Mary Jean Harrold. 1997. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology* 6, 2 (1997), 173–210. https://doi.org/10.1145/248233.248262

[23] Gregg Rothermel, Mary Jean Harrold, and Jeinay Dedhia. 2000. Regression test selection for C++ software. *Software Testing, Verification and Reliability* 10, 2 (2000), 77–109. https://doi.org/10.1002/1099-1689(200006)10:2<77::AID-STVR197>3.0.CO;2-E

[24] August Shi, Suresh Thummalapenta, Shuvendu K. Lahiri, Nikolaj Bjorner, and Jacek Czerwonka. 2017. Optimizing Test Placement for Module-Level Regression Testing. In *Proceedings of the International Conference on Software Engineering*. 689–699. https://doi.org/10.1109/ICSE.2017.69

[25] August Shi, Peiyuan Zhao, and Darko Marinov. 2019. Understanding and Improving Regression Test Selection in Continuous Integration. In *Proceedings of the International Symposium on Software Reliability Engineering*. 228–238. https://doi.org/10.1109/issre.2019.00031

[26] Yuri Shkuro. 2019. *Mastering Distributed Tracing: Analyzing performance in microservices and complex systems*. Packt Publishing Ltd.

[27] Benjamin H Sigelman, Luiz Andr, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Technical Report. https://doi.org/dapper-2010-1

[28] Marko Vasic, Zuhair Parvez, Aleksandar Milicevic, and Milos Gligoric. 2017. File-level vs. module-level regression test selection for .NET. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 848–853. https://doi.org/10.1145/3106237.3117763

[29] Joakim Von Kistowski, Simon Eismann, Norbert Schmitt, Andre Bauer, Johannes Grohmann, and Samuel Kounev. 2018. TeaStore: A micro-service reference application for benchmarking, modeling and resource management research. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. 223–236. https://doi.org/10.1109/MASCOTS.2018.00030

[30] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: A survey. *Software Testing Verification and Reliability* 22, 2 (2012), 67–120. https://doi.org/10.1002/stv.430

[31] Lingming Zhang. 2018. Hybrid regression test selection. In *Proceedings of the International Conference on Software Engineering*. 199–209. https://doi.org/10.1145/3180155.3180198

[32] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2013. FaultTracer: A spectrum-based approach to localizing failure-inducing program edits. *Journal of Software: Evolution and Process* 25, 12 (2013), 1357–1383. https://doi.org/10.1002/smr.1634

[33] Hua Zhong, Lingming Zhang, and Sarfraz Khurshid. 2019. TestSage: Regression test selection for large-scale web service testing. In *Proceedings of the International Conference on Software Testing, Verification and Validation*. 430–440. https://doi.org/10.1109/icst.2019.00052