Serverless Testing: Tool Vendors' and Experts' Points of View

Lenarduzzi Valentina, Panichella Annibale

DOI: 10.1109/MS.2020.3030803

Serverless Testing: Tool Vendors' and Experts' Point of View

Valentina Lenarduzzi
.LUT University, Finland. valentina.lenarduzzi@lut.fi

Annibale Panichella
TU Delft, The Netherlands. A.Panichella@tudelft.nl

***Abstract*—Serverless architecture is an emerging design style for cloud-based software systems. Testing serverless applications plays an important role in software quality assurance. However, currently, there is no consensus on how to test and debug such systems properly. Moreover, the current lack of mature tooling is a central challenge. We designed and conducted three interviews among two tools vendor leaders in the serverless domain (Epsagon and Thundra) and one expert in the field (Yan Cui), investigating the good and bad practices and several open issues. The current status of testing and debugging in serverless-based applications depicted by the experts helped us to highlight issues and challenges that need to be deeply investigated.**

■ SOFTWARE TESTING is a critical activity in Quality Assurance (QA) to assess whether a software project successfully meets its requirements and specifications. Software tests can be of different granularity levels. Considering the continuous evolution of cloud-based software systems, testability becomes an important factor, especially for integration and regression testing. For example, integration testing[1] can be challenging for large systems with a complex configuration. The goal is to test the component's services and detect anomalies associated with their interplaying interactions. For monolithic systems, testing and debugging can be done locally before deployment [5],[3].

Cloud-based applications introduce new challenges due to the increasing interaction complexity and the non-negligible difference between the local and deployment (remote) environment [7][6]. In microservices applications, each microservice is a small monolithic system to test in the local environment using well-established unit-testing strategies and criteria (e.g., branch coverage and mutation testing). The crucial difference is the assessment of the integration between the different microservices, as well as assessing the scalability of the overall architecture when considering different configuration alternatives.

Technical issues increase when considering serverless-based applications, which are "hosted by a third-party service, eliminating the need for server software and hardware management. Applications are broken up into individual functions (e.g., Lambda functions[2]) that can be invoked and scaled individually"[5]. A function is tiny and should be tested through unit tests. However, functional testing and debugging involve many functions and not a single one.

When several functions or external services

[1] Testing the interactions of the code units against external services that the application relies on

[2] https://docs.aws.amazon.com/Lambda/latest/dg/welcome.html

Author Version - IEEE Software, vol. 38, no. 1, pp. 54-60, Jan.-Feb. 2021, doi: 10.1109/MS.2020.3030803

are integrated, it is difficult and sometimes impossible to test and debug in a local environment because most environmental dependencies are only available at runtime [4]. To circumvent this issue, serverless applications can be tested directly in production or in a specific cloud development environment with the disadvantage of paying for test invocations and the production workload [4]. Moreover, developers generally adopt the good practice of running unit tests locally for each function by mocking environments to integration testing, via canary releases or A/B tests [4].

Integration testing is difficult to conduct due to the lack of emulation platforms, and because tests in production may have side-effects [4], [15]. If testing in a cloud-based system is difficult, debugging adds further complexity. Developers should track and observe requests between services, databases, caches, and even external API calls. Developers need to determine which application part is being called during a specific test scenario or execution. Moreover, they should identify which are the bottlenecks, and how the cache is helping save time by reducing round trips to the database.

Fault injection also allows finding and fixing unpredictable faults, such as database overloads, simulating the communication failures among different microservices, and other components. Another potential negative factor is the different programming languages used to implement cloud-based systems. This problem is addressed by testing services that use external containers that host different test designs and implementations. Several patterns were defined for serverless-based applications [8]. However, specific patterns for testing are not defined by practitioners yet. This leads to a lack of consensus on how to better test and debug [2], [4], [5]. For this reason, we conducted a survey with two tools vendors (Epsagoin and Thundra) and one practitioner (Yan Cu) with several years of experience in the field. The goal of this work is to identify good and bad practices for testing and debugging serverless-based applications. We are interested in understanding the complementary perspectives of both tool practitioners and vendors. The former can help us understand the common challenges they face when developing serverless-based applications. The latter allows us to understand both the challenges they face in developing their tools (based on microservices architectures), as well as the common challenges their customers report.

Setting the Stage: The Survey

The survey was composed by the following questions:

- What are the main challenges when testing serverless applications?
- How do you currently perform testing?
- What type of debugging strategy do you apply?
- How fine-grained is the instrumentation?
- How do developers identify the root cause of failure?
- How do you create crash-reproducing test cases?
- When do you decide testing is enough?
- What are your top wishes for improving testing?

We collected the information through open-ended questions, running the survey as a face to face interview of 30 minutes. The two authors collected the answers separately and then checked possible inconsistencies in the report. The disagreements was discussed and clarified. The transcript of each interview is available in the online appendix[3].

The experts point of view

We interviewed two tools vendor leaders and one experienced practitioners in the field: Ran Ribenzaft Co-Founder & CTO of **Epsagon**, Emrah Şamdan Leadership member of **Thundra**, and **Yan Cui**, an AWS Serverless Hero[4].

How to Test Serverless Applications

Epsagon designs test cases to cover the possible events and data values that each function receives from the outside. Similarly, they test for possible values Lambda functions may return. More specifically, they test the Lambda functions against failure scenarios and test how the functions recovers/handles in case of a failure. For example, in case of a failure, a Lambda function should re-send data to another function

[3] https://figshare.com/s/4ec5e8bd043571d61b6a

[4] https://aws.amazon.com/developer/community/heroes/yan-cui/

(synchronous case) or retry until the data expires (asynchronous case). So, they test for non-functional problems like robustness, resilience, and so on. Developers rely on both a checklist and their experience when testing serverless applications. Developers write unit-level and integration-level test cases for the applications and the local machine/environment, also relying on mocking frameworks.

Thundra adopts various levels of testing. Unit testing is done locally (e.g., in the IDE) for the components under maintenance (e.g., a service that handles accounts). However, tests need to mock data from other microservices that interplay to implement a given functionality. For component-level testing, developers consider components as black-box where data is shipped into the entry points; at the end state of the API, they verify whether they obtain the expected value (assertions). In particular, developers have to be sure to hit the API and the integration points. Given the dynamic natures of serverless applications, developers need to focus on testing the single microservices' interactions.

Yan Cui recommends reversing the testing pyramid to focus the testing effort more toward the integration points. This is because integration points are the most probable location of bugs or issues. However, developers have to be cautious with mocks: when they write integration tests, the risk is to hit the mocks rather than the actual services. Developers may test the mocks but not the assumptions about how integration is happening with the target services. We should consider that this services are often developed by different teams. On top of integration tests, Yan Cui recommend to deploy the system and to run end-to-end tests against the product. These tests are based on usage scenarios that execute an entire scenario (story), and that involve multiple (many) services.

Serverless testing challenges

According to Ran Ribenzaft, the most challenging part is replicating the real (cloud) environment in local machines. More traditional small-scale software can be deployed in the local machine for testing purposes. The testing environment is close to the real environment in which the system will be deployed and used. This is no longer possible in the cloud as we do not know upfront which container will be used during deployment. Unit testing still happens, and it is still feasible; then, integration testing with the different libraries is still applicable. However, system-level testing cannot be carried out in the local machine.

Another challenge is *measuring coverage* in Serverless applications. Typically, a serverless application has hundreds of Lambda functions that interact with one another. We can locally test code coverage at the unit level (e.g., branch coverage). However, the question is: *"how can we be sure that we tested all possible events a Lambda function may receive from other functions?"*

Emrah Şamdan was clear: first and foremost, the resources are managed by the cloud vendors. This means that local testing (on local machines) differs from remote testing (testing in the cloud). Therefore, testing the code in a realistic environment becomes much harder. Besides, it also changes how developers perform unit and integration testing. In our context, unit testing exercises single functions/services, while component-level testing involves multiple services that interact via HTTP requests or direct invocations. Testing locally requires extensive usage of mocks.

According to Yan Cui serverless-based systems are, by design, more scalable, easier to build loosely-coupled components. But the downside is that they are more challenging to test. In large event-driven systems, developers have one API call, and that triggers something to the event bus, which then triggers other Lambdas, and so on. Developers need to test the entire flow. The other challenge regards *how to run the whole application in local machines via simulation*.

Another challenge Yan Cui explained is when *dealing with asynchronous events*. A Lambda function publishes a message on the event bus. If developers write stubs in the streaming or event bus somewhere, they need to validate that the request is handled correctly (e.g., the database is successfully updated). Developers can write tests by checking the order of messages received in the last N seconds. To detect these messages, developers need to subscribe and capture the messages at the moment they were sent. Part of the tests will consist of determining which messages have been published to the event bus.

Of course, if developers think about a single account, that is okay. The problem comes when we have multiple messages published to the same event bus. This scenario is way more difficult to test due to possible side effects.

When do you decide that testing is enough?

All three participants considered this question particularly critical. For them, the decision about "when to stop testing" is driven by developers' experience and internal guidelines/check-list to consider (for the tool vendors). They also acknowledged that internal guidelines help (e.g., code coverage). In the following, we report the details of their practices and check-lists.

Ran Ribenzaft explained that they use different levels of testing. They also combine dynamic testing with strict code reviews. Code reviews follow a very strict check-list. Among the factors to consider, developers consider memory usage and running time. Expertise on performance testing is gain via hands-on sections with senior engineers.

Thundra heavily relies on unit testing. They push for higher-levels of testing, mostly for stable and well-established services. Unstable (prone to be changed) services would require actively maintaining and updating unit tests. Continuously changing both production and test code incurs in a non-negligible overhead.

Moreover, Thundra relies on functional testing (scenario-based) and API testing. The latter aims to verify that the changes do not break the APIs. Function testing is mostly done writing end-to-end testing. Test scenarios are designed upfront before coding (at the requirement gathering stage). Then, once the code is completed, they break test scenarios down into actual tests. Finally, they perform performance and load testing to make sure that our service can scale to a large number of requests and that they do not impact our customers' code.

Yan Cui targets obvious edge cases on top of code coverage. He is not a firm believer in coverage metrics because it is not a linear metric, and it requires more effort the closer it gets to 100%. Besides, developers should achieve large coverage but without considering edge scenarios that can frequently happen in practice. Moreover, Yan Cui also does some forms of fault-oriented tests (mutation testing).

This is a call to arms for defining test adequacy criteria that are specific for severless-based applications.

Crash-reproducing test cases

In the case of failures, Ran Ribenzaft recommends developers to reproduce the failure in their local machine using the data from instrumentation and distributed tracing. Reproduction is done by writing a test case (usually integration level) or test scenario. The test is also useful for future regression testing. Emrah Şamdan, for minor crashes, suggests to proceed with fixes, and then to write test cases for regression purposes (postfix tests). These are typically 60%-70% of the crashes. However, 20%-30% of the issues can be very complicated. In the latter case, Thundra retrieves the data and writes tests upfront for replication purposes.

If the crash is due to an unusual/unknown failure scenario, Yan Cui prefers to write crash-reproducing test cases as much as possible. If a crash happens, but developers know the most likely failing scenario, Yan Cui suggests writing the crash-reproducing test case after the fixing, mostly for regression purposes. However, *it is not always possible to write test cases* because the conditions under which it happens are extremely rare (e.g., the crash happens only when having more than 10K transactions simultaneously).

How to Debug Serverless Applications

Epsagon uses post-execution (offline) debugging: they instrument the code at the line level. Once the execution ends, developers can inspect how the variables changed during each step of the execution. They also have execution logs, but they are the last resource. Logging is complicated as log statements often need to be updated to store the relevant information. Logging updates can be done after a failure has happened and deployed for the next application versions.

Monitoring and debugging is complex for two main reasons for Thundra. First, when some exceptions are raised at some point in the chain (service), the actual root cause of the error can be different in the flow. Therefore, the complexity of debugging increases with the length of the chain. Thundra teams use distributed tracing to analyze how a call propagates through different services, functions, resources, and so on. Second, *many resources stream execution logs, which store the system behavior in chunks or pieces*. For example, we have Lambda function logs, API gateway logs, message queue logs, etc. Yan Cui heavily relies on instrumentation to increase observability, like using structured logs, a centralized logging framework. Developers also use log metrics that tell us what the performance is for different integration end-points. Though tracing, developers can see transactions as a flow/interaction of many Lambda functions. Yan Cui rarely relies on running Lambda functions locally because many issues can only be found in production and under certain conditions. These type of issues are not easy to reproduce locally.

What tool vendors need to improve testing and debugging

The main wish of Ran Ribenzaft would be the ability to record the states of variables from different Lambda functions and microservices when the application runs in the field. *That would allow to replicate the application states in a local machine*. This isn't very easy: big data to handle, reducing the overhead.

Another wish of Emrah Şamdan is to have a *tool that can automate the test case generation design/and generation* (i.e., writing test code in one click). Such tools should focus on end-to-end testing for serverless-based applications, i.e., generating tests from high-level specifications. These functional tests should exercise the microservices starting from their main APIs. Besides, integration test generators should automatically mock out the external services, reducing the effort needed to focus on tiny details. A tool that would help developers deciding whether to upgrade a library or not would be very helpful. Another Emrah Şamdan's wish is to have an *automated code review*. A kind of oracle that can tell developers if a Pull Requests is ready to be merged. These automated tools should allow developers to determine whether changes hinder the security, scalability, and interactions of the modified services with the unchanged part of the application.

Other considerations

During the discussion, we pointed out other important arguments.

- **Code review**. Thundra developers do not merge Pull Requests to the master branch without considering code review.
- **Managing and handling distributed logs.** Thundra localizes the right event service they are looking for and analyze the traces. These tools also provide performance information, like the number of performed calls, number of errors, memory usage, etc. They try to reconstruct a story (or scenario) from the raw, unformatted text. These tools help to navigate the traces, but the process remains manual.
- **Third-party libraries.** Thundra developers usually lock the versions they used in their systems. When they need to update a version (e.g., for security updates), they perform manual testing for every component and element that interacts with the library. Hence, they update libraries only when strongly needed.
- **How fine-grained is the instrumentation.** Epsagon uses the instrumentation with caution. Developers mostly instrument only some parts of the applications to avoid generating too much data and extra over-head. Developers decide what to instrument and what not based on their own experience. For Yan Cui, instrumentation should be around the integration points. Depending on the tool, developers can capture each request and response from HTTP requests. In production, Yan Cui prefers to deploy the targets applications by activating only around 1% of the debug-level logs to avoid generating too much data and incurring extra overhead. At the Lambda level, developers have the log at the entry and exit points. Besides, if the Lambda contains several branches, developers can instrument some of these branches based on past personal experience.
- **How do developers identify the root cause of failure?** To identify the root cause of failures, Epsagon developers use distributed tracing and manual inspection. Developers can create their own distributed tracing using open source standards (e.g., OpenTelemetry). Besides, other libraries are used to visualize the traces. This

allows us to inspect the application behavior in the execution trace/chain, e.g., how data is received and forwarded to/from other functions. In case of a failure, Yan Cui suggests to look at the log metrics to see the specific problem and check which transaction is responsible for bringing the success rate below the threshold. And then, once he identified a specific entry point, he started digging into that.

The Future Prospective

Based on participants' opinions, we pointed out some research opportunities for academics and points for collaborations with practitioners.

New dimensions to code coverage. In serverless-based applications, the complexity resides in the heavy interactions between different microservices and with the environment. Standard test adequacy criteria mostly measure the unit complexity (e.g., branches) of the Lambda functions. However, we foresee the definition or applications of integration-level criteria (e.g., coupling coverage [14]) over unit-level criteria in this context.

Debugging in distributed systems. A common issue with serverless-based applications is that debugging in the local environment does not reflect the complexity of the remote environment the applications are deployed in. Research effort focused on debugging techniques (e.g., fault localization [9], crash reproduction [10]) for monolithic applications. More research is needed for microservices since the execution path is spread over different microservices and logs.

Program comprehension via Big Data Analysis. Microservices generate various distributed logs with different formats, granularity levels, and purposes. Developers are called to reconstruct the executions scenarios via log analysis to understand what went wrong when running the application in the fields. While the industry has widely used tools and technologies to facilitate program comprehension, more research is needed to reduce the big log data developers need to analyze, and decide what to log.

Test Case Generation. These approaches have been widely investigated by the research community, mostly in the context of unit-level testing (e.g., EvoSuite ([12], Botsing ([11]). More recent tools focus on system testing, such as EvoMaster [13] for RESTful API. However, these tools focus on traditional code coverage metrics (e.g., branch coverage) that, as mentioned above, do not measure the main complexity of serverless applications. As highlighted by our participants, classic coverage criteria fall short for serverless-based applications. A new generation of tools should focus more on the interactions between Lambda functions and mocking external services. Finally, test generators should focus on end-to-end testing, allowing to test entire scenarios that involve many services at once (e.g., testing complete transactions).

Conclusion

This paper focuses on testing and debugging of serverless-based applications, considering expert opinions. We interviewed two of the main leaders in this domain and one consultant with a higher seniority level. Results depicted the current status of testing and debugging in serverless-based applications, highlighting issues and challenges.

We identified some potential aspects where academia and practitioners could synergically collaborate to fill gaps and improve the current status: (1) defining new and more appropriate test adequacy criteria for serverless applications; (2) defining new fault localization and crash reproducing techniques for microservice applications; (3) big data analytics for distributed logs; (4) test case generation tools specific for serverless applications and the interactions across Lambda functions.

Acknowledgment

The author thank Ran Ribenzaft (Epsagon), Emrah Şamdan (Thundra) and Yan Cui for taking the time to share their experience and provide their invaluable opinion.

■ REFERENCES

1. T. Clemson. Microservice Testing. https://martinfowler.com/articles/microservice-testing/ . 2014
2. I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, P. Suter. Serverless Computing: Current Trends and Open Problems. Research Advances in Cloud Computing. 2017.

3. Casale, G., Artač, M., van den Heuvel, W. et al. RADON: rational decomposition and orchestration for serverless computing. SICS Software.-Intensive. Cyber-Phys. Syst. (2019).
4. P. Leitner, Erik Wittern, J. Spillner and W. Hummer. A mixed-method empirical study of Function-as-a-Service software development in industrial practice. Journal of Systems and Software. 2019
5. J. Nupponen and D. Taibi. Serverless: What it Is,What to Do and What Not to Do. International Conference on Software Architecture (ICSA 2020). 2020
6. J. Soldani and D.A. Tamburri and W.J. Van Den Heuvel. "The pains and gains of microservices: A Systematic grey literature review. Journal of System and Software. 2018.
7. D. Taibi and V. Lenarduzzi and C. Pahl. Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation. IEEE Cloud Computing. 2017.
8. D. Taibi, N. El Ioini, C. Pahl and J. R. Schmid Niederkofler. Patterns for Serverless Functions (Function-as-a-Service): A Multivocal Literature Review. 10th International Conference on Cloud Computing and Services Science (CLOSER 2020). 2020
9. S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, B. & Keller. Evaluating and improving fault localization. In IEEE/ACM 39th International Conference on Software Engineering (ICSE). 2017.
10. M. Soltani, A. Panichella, A. Van Deursen. Search-based crash reproduction and its impact on debugging. IEEE Transactions on Software Engineering. 2018.
11. P. Derakhshanfar, X. Devroey, G. Perrouin, A. Zaidman, A. van Deursen, A. (2020). Search-based crash reproduction using behavioural model seeding. Software Testing, Verification and Reliability, 30(3). 2020.
12. G. Fraser, A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, 2011.
13. A. Arcuri. RESTful API automated test case generation with EvoMaster. ACM Transactions on Software Engineering and Methodology (TOSEM) 28.1 (2019): 1-37.
14. A. J. Offutt, A. Abdurazik, R. T. Alexander. An analysis tool for coupling-based integration testing. In Proceedings Sixth IEEE International Conference on Engineering of Complex Computer Systems. 2000.
15. D.Taibi, J.Spillner, K. Wawruch. Serverless: Where are we now and where are we heading? IEEE Software vol 38, no 1, Jan/Feb 2021.

Valentina Lenarduzzi is a researcher at the LUT University in Finland. Her primary research interest is related to data analysis in software engineering, software quality, software maintenance and evolution, with a special focus on Technical Debt. She obtained her PhD in Computer Science at the Università degli Studi dell'Insubria, Italy, in 2015. She also spent 8 months as Visiting Researcher at the Technical University of Kaiserslautern and Fraunhofer Institute for Experimental Software Engineering (IESE). In 2011 she was one of the co-founders of Opensoftengineering s.r.l., a spin-off company of the Università degli Studi dell'Insubria.

Annibale Panichella is an assistant professor in the Software Engineering Research Group (SERG) at Delft University of Technology (TU Delft) in the Netherlands. He is also a research fellow in the Interdisciplinary Centre for Security, Reliability, and Trust (SnT) at the University of Luxembourg, where he worked as Research Associate until January 2018. His research interests include but are not limited to security testing, evolutionary testing, AI-based software engineering, and empirical software engineering. He serves and has served as a program committee member for various international conferences (e.g., ICSE, GECCO, ICST, and ICPC) and as a reviewer for multiple international journals (e.g., TSE, TOSEM, TEVC, EMSE, STVR) in the fields of software engineering and evolutionary computation.