PREPRINT - Computer Science - Research and Development manuscript No.
(will be inserted by the editor)

Troubleshooting Serverless Functions

A Combined Monitoring and Debugging Approach

Johannes Manner · Stefan Kolb · Guido Wirtz

Abstract Today, Serverless computing gathers pace and attention in the cloud computing area. The abstraction of operational tasks combined with the auto-scaling property are convincing reasons to adapt this new cloud paradigm. Building applications in a Serverless style via cloud functions is challenging due to the fine-grained architecture and the tighter coupling to back end services. Increased complexity, loss of control over software layers and the large number of participating functions and back end services complicate the task of finding the cause of a faulty execution. A tedious but widespread strategy is the manual analysis of log data.

In this paper, we present a semi-automated troubleshooting process to improve fault detection and resolution for Serverless functions. Log data is the vehicle to enable a posteriori analysis. The process steps of our concept enhance the log quality, detect failed executions automatically, and generate test skeletons based on the information provided in the log data. Ultimately, this leads to an increased test coverage, a better regression testing and more robust functions. Developers can trigger this process asynchronously and work with their accustomed tools. We also present a prototype *SeMoDe* to validate our approach for Serverless functions implemented in Java and deployed to AWS Lambda.

Keywords Serverless Functions · FaaS · AWS Lambda · Monitoring · Debugging · Testing

Johannes Manner
E-mail: johannes.manner@uni-bamberg.de

Stefan Kolb
E-mail: stefan.kolb@uni-bamberg.de

Guido Wirtz
E-mail: guido.wirtz@uni-bamberg.de

Distributed Systems Group, University Bamberg, An der Weberei 5, 96047 Bamberg, Germany

1 Introduction

Serverless is quite a new computing paradigm and started its rise in 2012 [1]. Nowadays, some argue [2] that Serverless is the latest option in cloud computing and *Function as a Service (FaaS)* is the fourth and latest service model in this area [3], besides the established service models *Infrastructure as a Service (IaaS)*, *Platform as a Service (PaaS)* and *Software as a Service (SaaS)*.

To date, there is no clear definition of *Serverless* and some authors [4,5] argue that servers are also transparent in other cloud paradigms such as PaaS [6] not solely FaaS. However, when talking about Serverless, the majority of authors implicitly addresses FaaS. In the following, we also use FaaS and its ecosystem equivalent to Serverless. FaaS [7] is considered as an architectural paradigm, where stateless functions are deployed to a FaaS platform, which abstracts all operational tasks and removes the burden of writing operational logic from FaaS users. The hype is explicable, as FaaS providers exercise more control over the software stack [8], which facilitates auto-scaling of short-lived and context-unaware cloud functions. Functions are executed in response to events. A single event triggers the FaaS platform to execute a function on-demand. Therefore, it either starts a new instance of the function or reuses an existing one. If no demand is present for some time, the FaaS platform scales the function to zero and terminates all instances to avoid idling. The auto-scaling property is the basis for a calculation model where users only pay when function instances are running. This granular billing has positive effects on costs of FaaS architectures compared to IaaS and PaaS solutions [9]. Also the execution time of compute-intensive workloads profits from the parallelism, which is provided by the FaaS platform. As JONAS ET AL. [10] pointed out, developers are more familiar with writing clean and performant single-threaded code, but lack experience with multi-threading and suitable synchronization of shared memory.

In contrast, the benefits of fine-grained, cohesive functions lead to a complex application architecture, as the overall number of components rises. As a consequence, the likelihood of failures and logical bugs increases, because functions are also part of different applications. Troubleshooting these erroneous functions is time-consuming and tedious. Large log files and log drains, which collect data from different functions, often hide the cause of an error. Finding all mandatory parameters is challenging and some are often missing. In this case, a reproduction of the input and context is not possible and impedes troubleshooting of the function. Therefore, contribution of this paper is an approach for:

- Semi-automation to troubleshoot cloud functions consistently via appropriate log messages
- Continuous improvement of function quality by enhancement of test cases in quality and quantity

Our agenda is as follows: In Section 2, we present existing literature about monitoring cloud applications in general with an emphasis on logging. Based on these insights, we present our troubleshooting concept for cloud functions in Section 3 and our prototype in Section 4. Future work in Section 5 concludes this paper.

2 Related Work

Researchers [4, 11] noted that a lack of tooling is present due to the early stage of FaaS. Monitoring, logging, and debugging cloud functions are the foundations of the presented concept, but have not been directly investigated for FaaS according to our knowledge.

SPRING [12, 13] defines seven layers for monitoring cloud computing. Facility, network, hardware and OS layer are completely monitored by FaaS providers. Layer five *middleware* and layer six *application*, which is the collections of cloud functions, are partly provider and partly user monitored. The middleware contains the monitoring and logging service, which stores the log data in log groups per cloud function. This high-level monitoring approach is the basis for several activities like *Accounting and Billing*, *Fault Management* and *Performance Management* [14]. All high-level activities include a provider and a consumer perspective. Users are the last layer and not further investigated.

System logs are often written as plain text messages without a defined data schema and enriched with further environment parameters dependent on the implementation of the logging service. Additionally, every developer chooses an individually defined level of detail to provide him and other developers with sufficient information to resolve errors or logical bugs a posteriori. Tools like *LogEnhancer* [15] try to mitigate inconsistencies by enhancing existing log statements to achieve a better information quality but do not ensure that all input and context parameters are logged. The correctness of event generation based on system log analysis [16] is not sufficient without appropriate log messages.

Semi-automated test generation [17] reduces the developer effort significantly. It also improves incomplete test suites and the overall test coverage. The human part in this process is to decide, if generated tests are correct or adjustments on the assert conditions are required.

3 Troubleshooting FaaS

3.1 Log-based Debugging

Log-based debugging is a double-edged sword, but essential to analyze bugs a posteriori. Debugging the live environment is not supported by most of the FaaS platforms, because this reduces the FaaS provider's control over the platform. The obstacles of logging are the addition of non-functional source code, which results in a modification of the function's source code and a scaling problem of manual analysis, if mature tooling for screening and information extraction is missing. Generating log data consumes further resources. Firstly, the execution time of functions rises according to the number and size of log statements. This results in higher costs, because cloud functions' execution time is billed in millisecond chunks. Secondly, the logging service is another back end service in the provider's ecosystem. It causes further costs to communicate with the FaaS platform and to store log data in a cloud database. A trade-off between information quality and additional costs is important for the acceptance of the troubleshooting concept in general.

Advantages are a consistent execution history, since log data is persisted in a database and preserved against modification. The consistent order of events within the log data enables an offline reconstruction of bugs, if all required information is included in the log data. This analysis decouples the debugging process from the FaaS provider and enables a time-independent investigation.

3.2 Troubleshooting Process

The troubleshooting process, depicted in Figure 1, is divided into three phases and assigns each process step an area of responsibility. Some of these steps are explicitly introduced to facilitate the concept.

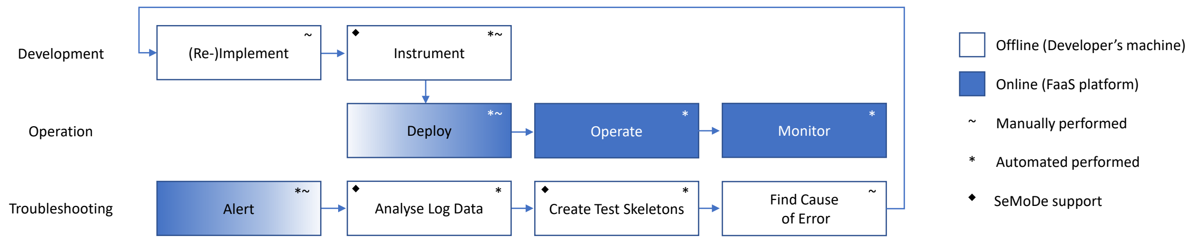

Fig. 1: Section of Cloud Function Life Cycle with Troubleshooting Elements

3.2.1 Development

The development phase consists of implementation and instrumentation of the cloud function. To enable an easier portability between providers, the suggestion in the development phase is, to separate all platform logic from the business logic. To achieve a high test coverage and establish a test suite for regression testing, the recommendation is to write several tests for each cloud function to get quick feedback on the functional correctness during development. Cloud functions are black boxes and executed in an isolated environment. Based on these characteristics, we argue that logging the *input*, *context* and *output* of cloud functions is sufficient to reproduce bugs a posteriori. The instrumentation of cloud functions to log these parameters is an essential element in the overall troubleshooting process. This is done by adding appropriate logging statements inside the code to establish a custom data schema for instrumentation messages. A more generic solution without modifications to the business code is the use of interceptors or annotations, if supported by the programming language. Cloud functions are stateless and therefore shared state is not a problem at all. Access to shared services like a cloud database are not considered in this instrumentation approach. The parameters are logged in a standardized, machine-readable format like JSON. After instrumenting the cloud function, another test cycle ensures the correctness of the instrumentation code itself and tests the functional correctness as well to exclude side effects.

3.2.2 Operation

Deploying a single cloud function is straightforward. A FaaS user bundles the source code and all dependencies in an archive and uploads this archive. Deployment and operation is completely managed by the FaaS provider. Monitoring activities support FaaS providers to ensure a constant quality of service by monitoring all layers of the hardware and software stack. FaaS users are informed by metrics about the liveliness of their cloud functions. *Performance Management* is essential to ensure a constant quality of service. Experiments on multiple VMs [18] showed a variation in execution time and performance. Due to different technical implementations of FaaS platforms and the fine-grained nature of cloud functions, a monitoring service needs a mechanism for specifying a corridor for execution times, w.r.t. the average time needed for executing the function, and a deviation rate. Thereby a custom implementation of the monitoring service, which polls the logs and analyses them, or an integrated backend service, which is present in rudimentary form in all mature FaaS ecosystems, is used to monitor the cloud functions. Based on these settings, the monitoring service identifies executions, where cloud functions exceed or deceed the corridor. The metadata of these executions is passed to a troubleshooting endpoint, like an email account or another cloud function, where a developer starts the troubleshooting process manually or the metadata serves as a new event triggering automated test generation. *Fault Management* is one of the most difficult challenges in cloud computing [18]. After detecting an error, required actions send notifications to predefined endpoints and visualize the errors in form of a dashboard. An assessment of the error and its context information support developers to determine if the error is related to a cloud function and must be resolved or if the error is FaaS platform-specific and the provider must be informed.

3.2.3 Troubleshooting

The notifications, which are sent by the monitoring services are alerts on the FaaS users side. Therefore, alerts are the interface between FaaS user and provider and a trigger for the developer to start the troubleshooting process manually or define an event driven endpoint, which automatically analyses system logs and creates test cases. The first step in analyzing the log data is to process consistent log events, which represent single executions, because the messages in most of the logging services are only plain text and not grouped yet. Grouping these plain text messages can be achieved by the order of the plain text messages within the logging service or via a request identifier, which is a metadata field included in every log message. The mechanism to correctly identify different function executions is thus logging service specific. Individual helper implementations are needed to extract the events properly. The next step in the troubleshooting process is to filter these log events by a condition, e.g., for Java the most suitable condition is *'Exception'* to retrieve all log events of failed executions. False negatives, where the condition does not cover all erroneous executions, are not a problem, if the error handling within the cloud function's code propagates all errors to the logging service. Also logical bugs are addressable with this filter condition. Logging failed calls to third party services, like databases, enables an indirect investigation of the circumstances, which led to the failures. Finally to tackle all these scenarios, the parameters *input*, *context*, *output*, which were instrumented during development, are extracted and prepared for the creation of test skeletons. A template approach is chosen, where placeholders are replaced with the extracted data from the step before. Naturally, different templates are needed for each programming language. The last step is the error resolution, where a developer takes the generated tests, evaluates the input and updates the assert conditions, which are left blank during test skeleton creation. This is the case, because computing the semantically correct condition out of the input data is not feasible. This step includes local debugging to inspect the source code in detail with the parameters that led to a failed execution. If an error is function-related, the developer returns to the implementation phase, resolves the error and starts the process anew. Some steps, like the instrumentation, are optional in the second iteration and can be skipped.

3.3 Assessment

The presented concept decouples the debugging aspect from the FaaS platform. This enables an asynchronous analysis and a time-independent investigation of bugs. This has advantages for both parties. A provider grants no rights to underlying software layers, which would soften the abstraction of operational tasks. A developer has no timely limitation for debugging and can work with familiar, mature tooling on his own machine. The semi-automated process solves the tedious searching for all parameters to reproduce bugs and to start troubleshooting promptly. Due to the test generation and the integration of these tests in the test suite, the robustness of the cloud function improves continuously and the increasing number of test cases supports regression testing to be compatible to prior versions of the cloud function. The test case enhancement, as the second contribution of this paper, results from the additional number of test cases in the test suite and the recognition of edge cases during production. This is necessary, because a cloud function can be part of different applications and must guarantee a stable behavior over time. Logging all relevant parameters of a cloud function is an additional effort in execution time. Furthermore, a logging service with a cloud database is needed to store the log data. These two aspects directly influence the cost structure, which is an inherent drawback of the concept. Generating test skeletons from failed executions leads to the problem of semantic duplicates. Imagine a situation, where 100 events are executed by 20 instances of the same cloud function. All executions failed because of the same error. Assumed that there are no equal events, 100 test files are generated, but only a single test file is needed to tackle the problem. Resolving this problem automatically by filtering the error's cause and location is problematic, because some parameter settings in this set of semantic test duplicates cause the same error due to distinct reasons. Another limitation of the presented concept is the emulation of the FaaS platform and the settings of the instances, which execute the event-triggered cloud functions. Dev-prod parity is hard to achieve, as a local environment can simulate the FaaS platform, but does not obtain a completely equivalent reconstruction. A further restriction is the programming language and ecosystem dependency, because test templates are different for each programming language and analyzing log data is dependent on the used logging service. Furthermore, the instrumentation and analysis is closely related to each other. Each programming language needs an utility mechanism to define instrumentation statements in an appropriate data format between instrumentation and analysis.

4 Prototype SeMoDe

SeMoDe[1] is our prototype to instrument cloud functions and generate tests based on log data from faulty cloud function executions. It includes utility methods, shown in Listing 1, for instrumenting AWS Lambda functions to log *input* and *output*, as described in Section 3.2.1. Both `instrumentFunction` method calls are hard coded as first and last statement within the handle method in the actual prototyping phase.

SeMoDe requires five parameters to execute the test generation functionality. The AWS region, where the cloud function is deployed, the log group name of the AWS Cloud Watch logging service, a search string to filter log events and the start and end time for retrieving log streams. Listing 2 shows SeMoDe's Java template for

[1] `https://github.com/johannes-manner/SeMoDe`

Listing 1: Instrumentation Methods for AWS Lambda

```
public static void instrumentFunction(String
  handlerClass, String handlerMethod, String
  inputClass, Object input, String outputClass);

public static void instrumentFunction(Object output);
```

test generation. All terms in capital letters are placeholders for the logged parameters of the instrumentation phase. To support developers with additional context information, the log data from the logging service is added as a comment within the test file.

Listing 2: Template File for Test Generation

```
1  /**   FUNCTIONLOG    */
2  public class FILENAME {
3    private static INPUTCLASS input;
4
5    @BeforeClass
6    public static void createInput()
7        throws IOException {
8      String jsonInput = INPUTJSON;
9      input = new ObjectMapper()
10           .readValue(jsonInput, INPUTCLASS.class);
11   }
12
13   @Test
14   public void testLambdaFunctionHandler() {
15     HANDLERCLASS handler = new HANDLERCLASS();
16     OUTPUTCLASS output =
17           handler.HANDLERMETHOD(input, null);
18
19     Assert.assertEquals(??, output);
20   }
21 }
```

The condition in line 19 needs a manual adaption, because it is not possible to compute the semantically correct output. Based on the input string and the context information provided trough the log data, the developer has to decide about the expected output, which is compared with the output generated by the invocation of the functional handle method.

The test file is an entry point for debugging and serves as a basis for further test cases. A developer starts the process with a syntactically correct condition in line 19. He is then able to start debugging the function to find the cause of the error and to resolve it. Finally, he considers the output of the computation, checks this output with his expectation and updates the expected value in line 19 with the semantically correct output. At last, he adds this generated and updated test to the test suite.

5 Future Work

The next step is to investigate the performance drawback to convince practitioners to adapt this troubleshooting concept for cloud functions. Also, the increased time consumption with different amounts of input data is on our agenda to tackle the problem of runaway costs. A study about mature logging services, their contingents, conditions and throughput can assist FaaS users to select an appropriate logging service. Investigating the semantic test duplicate problem is another task to filter duplicates automatically. Adapting the assert condition within the generated test skeletons is the most difficult part, which needs further considerations, what to test beyond the input-output relation of a function to get a more stable test set over time, e.g., testing the schema of input and output to support backwards compatibility. We also aim at extending the prototype to support further programming languages and FaaS platforms.

To conclude, *SeMoDe* is a first step towards a unified tool for troubleshooting cloud functions.

References

1. K. Fromm. Why The Future Of Software And Apps Is Serverless, 2012. `https://readwrite.com/2012/10/15/why-the-future-of-software-and-apps-is-serverless/`. Last accessed 2018-01-31.
2. N. Savage. Going Serverless. *Commun. ACM*, 61(2), 2018.
3. P. Mell and T. Grance. The NIST definition of cloud computing. 2011.
4. I. Baldini et al. *Serverless Computing: Current Trends and Open Problems*. 2017.
5. P. Sbarski. *Serverless Architectures on Aws: With Examples Using Aws Lambda*. Manning Publications, 2017.
6. S. Kolb and G. Wirtz. Towards Application Portability in Platform as a Service. In *Proc. SOSE*, 2014.
7. E. van Eyk et al. The SPEC Cloud Group's Research Vision on FaaS and Serverless Architectures. In *Proc. WoSC*, 2017.
8. S. Hendrickson et al. Serverless Computation with open-Lambda. In *Proc. HotCloud*, 2016.
9. M. Villamizar et al. Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures. In *Proc. CCGrid*, 2016.
10. E. Jonas et al. Occupy the Cloud: Distributed Computing for the 99%. In *Proc. SoCC*, 2017.
11. M. Roberts and J. Chapin. *What is Serverless?* O'Reilly Media, Inc. CA, US, 2017.
12. J. Spring. Monitoring Cloud Computing by Layer, Part 1. *IEEE Security Privacy*, 9(2), 2011.
13. J. Spring. Monitoring Cloud Computing by Layer, Part 2. *IEEE Security Privacy*, 9(3), 2011.
14. G. Aceto et al. Cloud monitoring: Definitions, issues and future directions. In *Proc. CLOUDNET*, 2012.
15. D. Yuan et al. Improving Software Diagnosability via Log Enhancement. In *Proc. ASPLOS*, 2011.
16. M. Kobayashi et al. Discovering Cloud Operation History through Log Analysis. In *Proc. AnNet*, 2017.
17. M. Kellogg. Combining Bug Detection and Test Case Generation. In *Proc. SIGSOFT*, 2016.
18. M. Armbrust et al. A View of Cloud Computing. *Communications of the ACM*, 53(4), 2010.