Data Flow Testing of Serverless Functions

Stefan Winzinger[a] and Guido Wirtz[b]
Distributed Systems Group, University of Bamberg, An der Weberei 5, 96047 Bamberg, Germany

Keywords: Serverless Computing, Faas, Integration Testing, Data Flow, Coverage Criteria.

Abstract: Serverless functions are a popular trend on the cloud computing market offered by many cloud platform providers. The statelessness of serverless functions enables the dynamic scalability by providing additional instances running these functions. However, statelessness doesn't guarantee the persistence of the state of a container running a serverless function for the next call. Therefore, serverless functions must interact with other services to save their state. This results in systems whose interaction with other services is complex and hard to test.
Considering the data flow resulting from the integration of different components is an adequate approach in an integration testing process. Therefore, we investigated the external factors influencing the execution of serverless functions to use this insight for the creation of a testing framework.
The framework helps measure important data flow coverage aspects supporting developers in their evaluation of test cases for the integration process of a serverless application. We showed that data flow criteria between serverless functions can be measured with a small overhead of run time making it attractive for developers to use.

1 INTRODUCTION

Serverless computing started to become popular with the introduction of Amazon's *AWS Lambda*[1] in 2014. Other tech companies, like Google[2], Microsoft[3] and IBM[4], followed and started to offer their own serverless solutions. Not only commercial serverless computing solutions are available but also open-source solutions, such as OpenFaas[5] or Fission[6]. In contrast to the proprietary ones, they can be deployed on an on-premise infrastructure.

Serverless computing is based on serverless functions, whereas *serverless* is in some way a misnomer since servers are still used in serverless computing. The term *serverless* means that servers are abstracted away. This enables developers to focus on the business logic without worrying about the infrastructure laying below (Baldini et al., 2017). In contrast to other cloud solutions, serverless functions only have to be paid for if they are actually used. The costs for the usage of serverless functions depend also on the composition of the function and its computing power assigned (Elgamal et al., 2018). But also other services provided by the cloud platform provider are usually used in a serverless application and have to be paid which could make alternative approaches like a VM running in the cloud cheaper, in particular if the computing workload is constant (Leitner et al., 2019).

The automatic scalability of serverless functions managed by the cloud platform provider makes them attractive for applications with an infrequent and bursty workload (Baldini et al., 2017). The scalability is based on the assumption that the serverless functions are stateless. Thus, cloud platform providers can host new instances running the serverless functions without violating the functionality of the application. Depending on the workload having to be processed by a serverless function, the cloud platform provider can deploy new instances of functions handling the requests or reduce the number of instances. However, statelessness simply means that the state within a serverless function is not guaranteed to be saved if the same function is executed again. Therefore, the state of an application is not eliminated, it is just moved to other parts of the application.

[a] https://orcid.org/0000-0002-4526-286X
[b] https://orcid.org/0000-0002-0438-8482
[1] https://aws.amazon.com/de/lambda/
[2] https://cloud.google.com/functions/
[3] https://azure.microsoft.com/en-us/services/functions/
[4] https://www.ibm.com/cloud/functions/
[5] https://github.com/openfaas
[6] https://fission.io/

Winzinger, S. and Wirtz, G.
Data Flow Testing of Serverless Functions.
DOI: 10.5220/0010439600560064
In *Proceedings of the 11th International Conference on Cloud Computing and Services Science (CLOSER 2021)*, pages 56-64
ISBN: 978-989-758-510-4
Copyright © 2021 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

The scalability provided by serverless functions is tempting and might give the impression that there are no bottlenecks at all. However, the services used to handle the state by reading or changing data are the actual bottlenecks of a serverless application.

Connecting these services to serverless functions leads to an overall complex structure which makes it hard to find faults. While stateless serverless functions simplify unit testing, integration testing becomes more complex in serverless applications and tool-support is still missing (Kratzke, 2018). Integration testing is an important testing phase for serverless functions and is a challenge in industrial practice (Leitner et al., 2019). In order to improve the testing process of serverless functions and their dependencies, we investigated how serverless functions interact with their environment and how the knowledge of the data flow with their environment can be used for testing.

Therefore, we sifted through the hosting platform *GitHub* to identify some publicly available serverless functions and analyzed their source code looking for external dependencies and their usage. The analysis showed that most of the serverless functions use external stateful services making it hard to scale serverless functions independently without considering the external services. But also the return values and parameters passed are potential data flows which have to be monitored.

These insights motivated us to implement a data flow testing framework measuring the coverage of important and relevant data flow criteria between serverless functions.

Furthermore, we described the workflow to measure the data flow of serverless applications and evaluated the run time for our framework. By interviewing some serverless experts, Lenarduzzi and Panichella stated in (Lenarduzzi and Panichella, 2021) that serverless applications need test adequacy criteria, in particular since serverless applications need to be tested with events generated by other serverless functions. Besides our previous work (Winzinger and Wirtz, 2019a; Winzinger and Wirtz, 2019b; Winzinger and Wirtz, 2020) where we addressed coverage criteria but didn't investigate the influence of external resources and measure the run time of an implementation of coverage criteria, to our knowledge, there is no work available yet handling this issue for serverless applications. Formal models of serverless computing were introduced in (Jangda et al., 2019) and (Gabbrielli et al., 2019) but didn't describe the influential factors of serverless functions in detail. While Lin et al. showed in their work (Lin et al., 2018a; Lin et al., 2018b) how serverless functions can be tracked causally which is useful for debugging, data flow criteria cannot be measured by their method. In (Sreekanti et al., 2020) an additional layer was added between serverless functions and the data storage service to improve fault-tolerance in contrast to our work where we only manipulated the access of serverless functions to data storage to measure the data flow. Furthermore, there is also other work available investigating serverless applications and their functions (Eismann et al., 2020), but without focus on the data flow between the serverless functions. Although coverage criteria considering the data flow for integration are not new (compare (Spillner, 1995; Frankl and Weyuker, 1988; Clarke et al., 1989)), they were not yet applied to serverless applications.

The structure of the paper is as follows. Section 2 describes how we investigated serverless functions and gives a model of data flows of serverless functions. The implementation of a testing approach for data flow coverage criteria is presented in section 3 followed by an evaluation of its run time. Finally, a conclusion is drawn and an outlook is given to future work in section 4.

2 DATA FLOW OF SERVERLESS FUNCTIONS

The data flows of serverless functions with their environment are relevant for the creation of test cases in order to cover the dependencies to other resources of the system. These data flows to other services can influence the state of the system. Therefore, we investigated several serverless functions in order to see how the execution process of a serverless function is influenced by external data flows to use these insights for our testing framework. Since there aren't many applications currently available using serverless functions, we searched for serverless functions on GitHub and analyzed them. We built a model showing the factors influencing the execution of serverless functions.

2.1 Selection of Serverless Functions

We decided to investigate serverless functions published on GitHub to build a model of factors influencing the execution of serverless functions. Similar to our previous work (Winzinger and Wirtz, 2020), we searched for applications focusing on *Amazon Web service* (AWS) and its usage with the *Serverless Framework*[7] on GitHub. This framework is often applied to describe the components of serverless appli-

[7]https://www.serverless.com/

Table 1: Languages used in applications.

Language	Number of projects
JavaScript	459
Python	59
Java	6
dotNet	3

cations and deploy their serverless functions. We used the *GitHub REST API*[8] for our search which provides, in contrast to *GHTorrent*[9], the possibility to search for filename and keywords contained in the files.

Files with the name 'serverless.yml' were selected which is the name of the file used by the *Serverless Framework*[10] describing the infrastructure of the application. Furthermore, the request to the API was set to filter for the keywords 'AWS' and 'handler' to ensure that serverless functions were used on *AWS*. Thus, files could be identified which are usable on *AWS Lambda* and contain serverless functions which are deployable by the *Serverless Framework*.

The API provided 1000 results for files describing serverless application with our search done on July 30, 2020. The files identified are part of 527 different projects. The projects were sorted by its number of stars which roughly indicates the popularity of a project. Afterwards, we analyzed the first 27 projects resulting in 323 serverless functions, after functions of projects had been sorted out which were not completely implemented, just boilerplate code or just a very simple demonstration. Furthermore, we only investigated projects with serverless functions written in JavaScript which was the most popular language used in the applications identified (see Table 1) in order to stay in the same domain of a programming language. The program with its data measured can be found online[11].

2.2 Investigation of Data Flows

We investigated several serverless functions and their structure to identify factors influencing the execution of serverless functions which have to be considered for the interaction of serverless functions with their environment. Therefore, we noted all types of calls requiring dependencies not running within the container of the serverless function.

We divided the calls made into calls made to platform-specific services, e.g., data storages, and calls made to external services like calls made via HTTP. If the service was a data storage service of Amazon (i.e., *DynamoDB* or *S3*) where data are saved or read, we took a note if data were written (e.g., updating, deleting or creating data) or read. The direct invocation of another serverless functions was noted too.

Additionally, we checked if the arguments passed to a serverless function were used at all. This indicates the direct influence of an external call. Furthermore, we evaluated how the usage of services influenced the return values of our functions.

Therefore, we categorized the return values of functions using services in *state*, *value* and *nothing*. *State* was noted if the return value just indicates that the execution was successful by returning a simple state message. The categorization *value* on the other hand was used for return values which were calculated depending on the input of other resources and contains more information than the pure state of the successful execution. If no return value was returned at all, *nothing* was used. Since usually the state of a service call is returned if its call fails, all return values investigated were influenced by the services called.

This categorization helps to see if there is a data flow between the calling resource and the callee or if the function is just called and only its successful execution is relevant for the caller (e.g., an orchestrator or a user).

Finally, we checked the function for the usage of object variables of a previous call and potential background processes.

2.3 Influential Factors of Serverless Functions

Based on our investigation, we could build a model of the interfaces of a serverless function describing both the influential factors to its execution coming from its environment and the influential factors of a serverless function to its environment (see Figure 1). In general, a serverless function gets the information needed to be processed by its arguments assigned to the parameters of the serverless function and can return a value. But there are also other factors we identified influencing the execution and the result of a serverless function.

Environment variables set for the configuration of the system can also be accessed by a serverless function during run time and influence its execution. A typical example of such a variable is the name used for a data storage. These variables cannot be set directly within a serverless function and are usually set

[8]https://docs.github.com/en/rest

[9]https://ghtorrent.org/

[10]https://www.serverless.com/

[11]https://github.com/snwinz/ServerlessApplicationSearcher/releases/tag/v1.0

when a new version of a serverless function is deployed. Therefore, they are not really relevant for our data flow evaluation since they can be considered as a constant factor after deployment.

Another factor which influences the execution process of a serverless function are processes running in the background. A serverless function can return a value and stop its execution without having background processes completed. These processes are continued if the same container is started again which can influence the state of the application even if the background process was started by another invocation. In the serverless functions we investigated, we didn't identify such processes started in the background.

A similar behavior occurs when a global variable is used in the same container again and changed during the execution of the container. If so, the value used by a previous call could influence the state of the application. If the instance variable is always instantiated the same way, it doesn't influence the system. However, we didn't identify such a behavior implemented in the serverless functions investigated where a previous call influences the value of a subsequent one. This behavior could be utilized in order to cache some data.

Besides these factors, there is also the possibility that a serverless function gets data to be processed by calling a service. The service can be platform-specific, like a data storage access, or external, like a call to an HTTP API. Of course, services can also be used to store data.

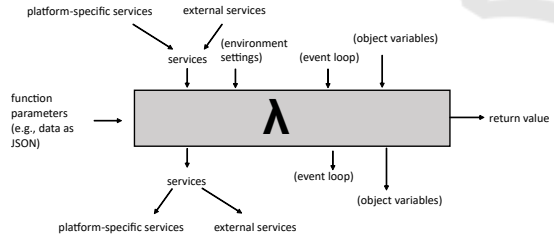

Figure 1: Data interfaces of serverless functions.

2.4 Analysis of the Interfaces of Serverless Functions

Our analysis showed that nearly all of the functions used services. Therefore, it is not enough to test only the function in isolation, but also other dependencies have to be considered. 306 ($\approx$ 94.44%) of the functions investigated used at least one service. A popular service containing and preserving the state of an application are data storages. More than half of the functions (162) used at least one platform-specific data storage like *DynamoDB* or *S3* where 142 of these ($\approx$ 87.65%) used at least once *DynamoDB*. 68 of these functions read data from and 77 changed data on *DynamoDB*. This shows the importance of platform services, in particular data storages.

Furthermore, all return values of the functions using services were influenced by the service called. Only 40 functions ($\approx$ 12.38%) didn't return anything explicitly at all, whereas nearly every third function (104) returned a value. The majority of the functions (179) returned a value indicating the successful execution of the function. A return value indicating only if the execution was successful is an indicator that the return value is only used for its orchestration, e.g., reexecuting the function by an orchestrator if the function fails. The parameters of a serverless function are instantiated by the events triggering the serverless functions and had also an effect on the execution of the serverless functions. 271 functions ($\approx$ 83.90%) used the parameter for its execution. However, there were only 13 serverless functions ($\approx$ 4.02%) invoking another serverless function directly.

Even if this analysis on GitHub cannot be representative for all serverless applications since many projects on GitHub are personal and inactive (Kalliamvakou et al., 2014), it still shows the different usages of serverless functions and its reliance on services, in particular data storages.

3 TESTING DATA FLOW BETWEEN FUNCTIONS

The investigation described in the previous that there are different ways serverless functions can exchange data with their environment. Even if there are several ways how the internal data flow during execution is influenced, the most relevant factor are services being called. So, the data flow of these services can influence the state of the system if data are written. If data are read, the execution process depends on an external state. Therefore, the state of the application resides in the data flow and the state of other services, mostly data storages. The services where the information is passed to, can process, save or route the information to another service. The most common service detected in our investigation was *DynamoDB* which is a key value storage. In general, each of the services called by a serverless function is a potential data storage where data could be saved and read by other parts of the system. Therefore, while developing a serverless application, it is not enough to test only the serverless function in isolation but also the influence of the data changed to the system. Thus,

not only the relations between services are tested, but also a direct influence between different services. We evaluated different coverage criteria and showed the applicability of data flow coverage for several applications in our previous work (Winzinger and Wirtz, 2020). Here, we show an implementation of a framework measuring the data flow coverage of a serverless application while focusing on different kinds of data flow: data flow between serverless functions via a service, here *DynamoDB*, data flow via return values and data flow via function invocation demonstrating the usage of a parameter set by an event. In addition, we present the work flow to measure this data flow and the minimal run time overhead of this measurement.

3.1 Data Flow Criteria

The data flow of the serverless functions is measured by modifying the source code with additional instrumentations. These instrumentations help to pass the information where the definition of a value took place and evaluate its usage. The data flow is only measured between serverless functions where the definition is in one serverless function and the usage of data is in another serverless function. Other services, like data storages, can interrupt the data flow between serverless functions. Therefore, the information of the data definition has to be stored in such services too.

The coverage criteria *all-resource-defs* and *all-resource-uses* were implemented in our tool which can be found online with the data of its run time analysis[12]. These criteria were introduced in (Winzinger and Wirtz, 2019a) and are defined as follows where x is a definition of a value within a serverless function which is used by another resource:

- *All-resource-defs:* requires that every x is at least used once in another resource without being redefined before its usage.
- *All-resource-uses:* requires that every x is used by all other usages of x in other resources without being redefined before its usage.

Applied to our serverless applications, resources are to be considered as serverless functions which use the value defined.

In contrast to all-resource-defs, all-resource-uses considers the coverage of uses explicitly. This results in a potential higher amount of test cases needed to fulfill this criterion. The maximal number of test cases required for both criteria to be fully fulfilled is listed in Table 2 if each testing aspect is covered by only one test case where I is the set of couplings (e.g., couplings via data storages or function invocations) and D_i are the defs and U_i the uses of a coupling $i \in I$. Serverless functions are often coupled externally by communicating through an external data storage. If there is a data storage with many functions writing and reading data from it and each test case is responsible for only one coverage aspect, the test cases to fulfill all-resource-uses are much higher. Therefore, we defined and implemented an additional coverage criteria. While the fulfillment of the all-resource-uses criterion requires that all def-use pairs are covered, we defined the weaker criterion *all-resource-defuse* which requires that:

Table 2: Coverage items of criteria.

Criterion	Maximal Number of Items
All-resource-defs	$\sum_{i \in I} \lvert D_i \rvert$
All-resource-defuse	$\sum_{i \in I} \lvert D_i \rvert + \lvert U_i \rvert$
All-resource-uses	$\sum_{i \in I} \lvert D_i \rvert \cdot \lvert U_i \rvert$

- each definition of a value within a serverless function which is used by another resource is used at least once in another resource without being redefined before its usage.
- each usage of a value defined in another resource is used at least once in combination with any definition without being redefined before its usage.

This requires less coverage objects than all-resource-uses but has the advantage that all usages have to be tested at least once. The three test criteria result in the subsumption hierarchy shown in Figure 2.

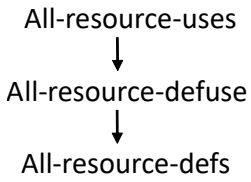

Figure 2: Subsumption hierarchy of data flow criteria.

3.2 Criteria Implementation

Our implementation of the criteria is done be extending our previous work (Winzinger and Wirtz, 2020). By instrumenting the source code with our tool, the information of the location where a value was defined is added to the arguments which are passed when the context of the serverless function is left. For example, if another serverless function is called where a value

[12]https://github.com/snwinz/ServerlessApplicationTool/releases/tag/v0.2-beta.1

is passed or a service like write access to data storage is made, the information of the definition of the data is passed too. Thus, when the value is used, a corresponding log statement can be created indicating that the definition is used with the actual usage. Based on our investigation from section 2, we implemented the data flow criteria for different kinds of data flows.

We implemented this for serverless functions which were invoked by other serverless functions by attaching the relevant information to the payload. Even if direct function invocation wasn't used very often in our analysis, this behavior is relevant, since the callee uses the information passed via a parameter of the event which is a common scenario, e.g., if a data storage triggers a serverless function when a new entry is added.

If a function returns a value, we attached also an identifier of its definition making it possible for the caller to interpret the source of the result. This was only implemented if the return value is a JSON-Object where the information can easily be attached to an additional field. If another data format is used, the usage of the additional information has to be adapted in a way that other resources using the return value can handle the additional information. Thus, if a string value should be returned, it would only make sense to attach the identifier of the definition if all services using the return value can handle it, e.g., by parsing the relevant value by a delimiter.

Finally, we implemented the criteria for serverless functions which are coupled via a data storage, here *DynamoDB*, by supporting write and get operations. Additionally, we implemented a support for the delete operation to measure the usage of deleted values. This required to replace the delete operations by write operations which added a marker to the corresponding value but removed the value. The get operation had to be replaced too in order to interpret the marked entry. If they identify a marker, an empty entry is returned to indicate that the entry is not available.

If a variable of a serverless function is used to pass information via a coupling, each of its definitions before the coupling was instrumented adding the information of the usage to the variable. Thus, when the variable is used, the latest definition of the variable can be read. The usage of variables is implemented similarly. Each usage of a variable coming from a coupling is instrumented by logging its usage. Thus, the first usage after a coupling can be identified later.

3.3 Workflow for Measuring Coverage

First of all, our program reads the source code with a parser. We used *ANTLR*[13] to create a parser for JavaScript. Other languages have to be implemented explicitly in the framework. The parser was adapted to identify relevant parts of the source code, e.g., statements responsible for passing data and definitions and usages of variables. These parts were enriched with JavaScript source code which added relevant information to the variables and added log statements to the source code. The generated source code has to be deployed afterwards and the test cases be run. In contrast to the implementation of (Offutt et al., 2000) where a global array is used to log usages, serverless applications are distributed. Therefore, we used CloudWatch for saving our information of usages, definitions, calls etc. However, in contrast to a global array where the values could be read directly, the log files have to be downloaded and evaluated explicitly. This is done after the execution of test cases. Monitoring the coverage of a system during its execution could be supported by scripts constantly polling and evaluating these logs. For each variable which was defined and passed, the log file is searched for an entry where the corresponding variable is used. This log statement contains the information of the definition which was originally attached to the variable passed and, if necessary, relevant information of the usage. By reading the log statements, all usages are counted and can be displayed.

3.4 Run Time Evaluation

This section shows the run time behavior of the code instrumented by our framework compared to the original code without instrumentation. We evaluated three different scenarios covering different kinds of data flow between serverless functions.

3.4.1 Scenarios

Our first scenario (Figure 3) is a serverless function calling another serverless function where the callee uses a value of the caller. The serverless function called returns a value to the calling serverless function which results in another coupling. Therefore, there exists a def-use pair both between the caller and the callee and the callee and the caller. The first def-use pair is a typical example for a serverless function using the argument of an event, whereas the second def-use pair is an example for a coupling through a return value. We measured the run time of the calling

[13] https://www.antlr.org/

function on cloud side since the caller calls the callee synchronously and has to wait for its answer. Therefore, the run time of the calling function includes the run time of the callee.

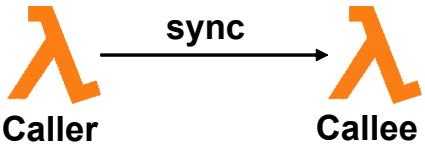

Figure 3: Model of function calling another function.

The second scenario (Figure 4) is a scenario with a write operation to a *DynamoDB* data storage whose value is read by another serverless function. Both functions are called synchronously by another serverless function whose run time is measured on cloud side. The read operation is set to be a consistent read which guarantees that the latest value of the data storage is read. Otherwise, only eventual consistency is guaranteed where potentially the latest write operation is not read. Thus, both serverless functions are coupled by a value on the *DynamoDB* storage.

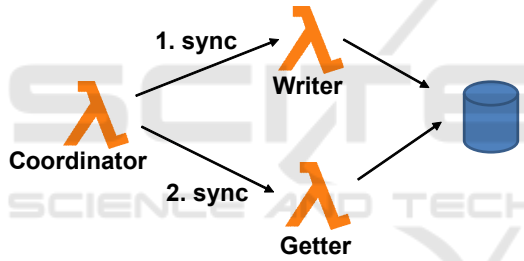

Figure 4: Model of scenario for coupling via a write operation.

The third scenario (Figure 5) is similar to the second one, but a value written to a data storage is deleted by a serverless function. This value is read by another serverless function with a consistent read like in the previous scenario. Both the deleting and reading functions are coordinated by a serverless function calling these functions synchronously. All values were written to the data storage before the coordinator function was called.

3.4.2 Execution

All tests were run on September 25, 2020. We executed each of these scenarios with its original source code and a version instrumented by our tool for each of the three coverage criteria. Furthermore, we verified that each def-use pair was tracked correctly.

Therefore, each scenario was executed with four different versions. The four versions were tested for

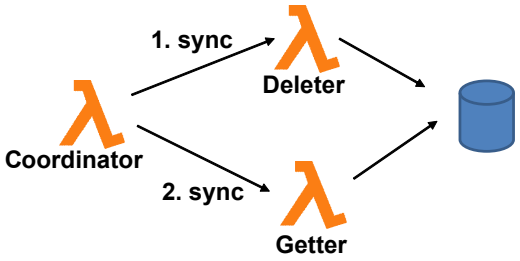

Figure 5: Model of scenario for coupling via a delete operation.

each scenario in one test run making sure that the utilization of resources of the cloud platform is similar. Each run was divided in 100 blocks where in each block each version was executed eleven times.

Before each single execution, the configuration of the serverless functions was upgraded. Thus, we could enforce a cold start of the serverless function for each first run of a block since the platform deploys a new container if the description of a serverless function is changed. This reduced the risk that a serverless function is only running on the same machine. Furthermore, we set the memory assigned to the maximal size of 3008 MB enforcing a deployment to a fast machine. Otherwise, slower machines could be compared to faster ones since sometimes if slower resources are assigned to a serverless function, faster ones are assigned by the cloud platform provider (Malawski et al., 2018; Figiela et al., 2018). Since logs got lost when the description of a serverless function was changed immediately after its execution, we waited 10 seconds after each execution block.

All in all, we compared 1000 warm runs for each of the versions of the scenarios and checked that the coupling was tracked.

3.4.3 Results

The results of our first scenario showed that there was nearly no difference in the execution times if a function was instrumented or it was run without any instrumentation, as can be seen in Figure 6 where a box plot is shown with the median execution time. The instrumentation is only limited to the addition of a value to the parameter and a few log messages where there were only small differences in the instrumentation added for the criteria.

The second criteria showed also no relevant differences in its run time (compare Figure 7). The instrumentation included additional checks for parameters passed and the value received from the data storage. The criteria themselves have only a few differences in its implementation by logging the concrete statement

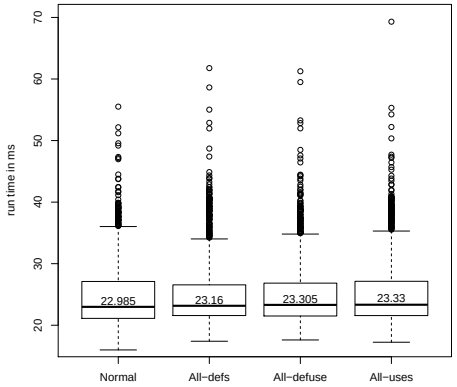

Figure 6: Box plot of the run times of caller scenario.

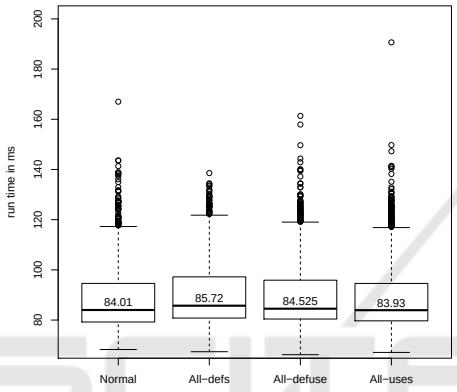

Figure 7: Box plot of run time of writer scenario.

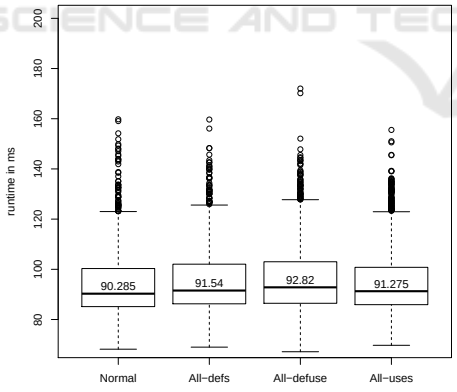

Figure 8: Box plot of run time of deleter scenario.

of the usage of a definition for all-resource-uses and all-resource-defuse. The median execution time for all-resource-uses was even faster, even if more statements have to be executed.

Our last scenario showed a bigger difference in its run time even if the scenario is quite similar to the previous one (compare Figure 8). However, the instrumented versions of this scenario don't use a delete operation but a write operation. Additionally, the entries on the data storage are overwritten which could also be a source of additional run time on the data storage caused by additional entries. Therefore, this scenario depends more on the service than the previous scenario where the same write operation was used with additional values whereas here the delete operation is replaced by a write operation.

There were only a small differences in the execution times of the instrumented scenarios compared to the original ones which shows that the execution time of the instrumentations is not so relevant for these scenarios. Even if these scenarios are very simple, they cover all the interfaces where an additional instrumentation is needed. Therefore, the execution of the instrumentation of a more complex function would only take much longer if it used more services or required more instrumentations to log the definitions and usages of variables used by its interfaces.

4 CONCLUSION AND FUTURE WORK

In this paper, we investigated the data flows of serverless functions with their environment. Most of the serverless functions of our analysis are influenced by its parameters passed and the usage of services, whereas data are mostly passed to the environment via services and return values. Therefore, we introduced a framework measuring the usage of these data flows. Serverless applications consist of many interconnected serverless functions and services whose complexity of interactions has to be covered. The measurement of the data flow with our framework doesn't require much additional run time. Therefore, developers can benefit from using it to detect and measure their data flows easily and create new test cases.

Our tool supports a dynamic coverage of data flows between serverless functions, whereas we plan to develop some static testing support for our future work, in particular to support the test case generation. Furthermore, since our tool measures the coverage after the execution of test cases, a live measurement of the coverage could support developers in the creation of test cases. There is still a lack of practical evaluations about the usage of the data flow coverage criteria for serverless applications. Therefore, we plan to interview some experts on the field and evaluate some real life applications.

REFERENCES

Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., and Suter, P. (2017). Serverless Computing: Current Trends and Open Problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer Singapore.

Clarke, L. A., Podgurski, A., Richardson, D. J., and Zeil, S. J. (1989). A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, 15(11):1318–1332.

Eismann, S., Scheuner, J., van Eyk, E., Schwinger, M., Grohmann, J., Herbst, N., Abad, C., and Iosup, A. (2020). Serverless applications: Why, when, and how? *IEEE Software*, 38(1):32–39.

Elgamal, T., Sandur, A., Nahrstedt, K., and Agha, G. (2018). Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 300–312. IEEE.

Figiela, K., Gajek, A., Zima, A., Obrok, B., and Malawski, M. (2018). Performance evaluation of heterogeneous cloud functions. *Concurrency and Computation: Practice and Experience*, page e4792.

Frankl, P. G. and Weyuker, E. J. (1988). An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498.

Gabbrielli, M., Giallorenzo, S., Lanese, I., Montesi, F., Peressotti, M., and Zingaro, S. P. (2019). No more, no less. In *Lecture Notes in Computer Science*, pages 148–157. Springer International Publishing.

Jangda, A., Pinckney, D., Brun, Y., and Guha, A. (2019). Formal foundations of serverless computing. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–26.

Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., and Damian, D. (2014). The promises and perils of mining GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*. ACM Press.

Kratzke, N. (2018). A Brief History of Cloud Application Architectures. *Applied Sciences*, 8(8):1368.

Leitner, P., Wittern, E., Spillner, J., and Hummer, W. (2019). A mixed-method empirical study of Function-as-a-Service software development in industrial practice. *Journal of Systems and Software*, 149:340–359.

Lenarduzzi, V. and Panichella, A. (2021). Serverless testing: Tool vendors' and experts' points of view. *IEEE Software*, 38(1):54–60.

Lin, W.-T., Krintz, C., and Wolski, R. (2018a). Tracing Function Dependencies across Clouds. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE.

Lin, W.-T., Krintz, C., Wolski, R., Zhang, M., Cai, X., Li, T., and Xu, W. (2018b). Tracking causal order in AWS lambda applications. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE.

Malawski, M., Figiela, K., Gajek, A., and Zima, A. (2018). Benchmarking Heterogeneous Cloud Functions. In Heras, D. B. and Bougé, L., editors, *Euro-Par 2017: Parallel Processing Workshops*, pages 415–426. Springer International Publishing.

Offutt, A., Abdurazik, A., and Alexander, R. (2000). An analysis tool for coupling-based integration testing. In *Proceedings Sixth IEEE International Conference on Engineering of Complex Computer Systems. ICECCS 2000*. IEEE Comput. Soc.

Spillner, A. (1995). Test criteria and coverage measures for software integration testing. *Software Quality Journal*, 4(4):275–286.

Sreekanti, V., Wu, C., Chhatrapati, S., Gonzalez, J. E., Hellerstein, J. M., and Faleiro, J. M. (2020). A fault-tolerance shim for serverless computing. In *Proceedings of the Fifteenth European Conference on Computer Systems*. ACM.

Winzinger, S. and Wirtz, G. (2019a). Coverage criteria for integration testing of serverless applications. In *13th Symposium and Summer School On Service-Oriented Computing*.

Winzinger, S. and Wirtz, G. (2019b). Model-based Analysis of Serverless Applications. In *2019 IEEE/ACM 11th International Workshop on Modelling in Software Engineering (MiSE)*. IEEE.

Winzinger, S. and Wirtz, G. (2020). Applicability of Coverage Criteria for Serverless Applications. In *2020 IEEE International Conference on Service Oriented Systems Engineering (SOSE)*. IEEE.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS