A Framework for Evaluating Integration Testing Criteria in Serverless Applications

Stefan Winzinger[a] and Guido Wirtz[b]
Distributed Systems Group, University of Bamberg, An der Weberei 5, 96047 Bamberg, Germany
{*stefan.winzinger, guido.wirtz*}*@uni-bamberg.de*

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

Abstract: Serverless applications are based on Function-as-a-Service (FaaS) platforms where serverless functions interact with other cloud-specific services. The integration of these components is crucial for the application's functionality and must be adequately tested. Testing criteria can help here by supporting developers in evaluating test suites and identifying missing test cases. This paper presents a framework for evaluating integration testing criteria in serverless applications by showing the processes needed for evaluating the criteria. The process defined is shown for some control and data flow criteria and applied to some serverless applications showing the feasibility of the approach.

1 INTRODUCTION

Serverless computing became popular with the introduction of Amazon's *AWS Lambda* in 2014 and is now also provided by other cloud providers like Microsoft's *Azure Functions* or Google's *Cloud Functions* (Baldini et al., 2017).

The serverless functions are the core of serverless applications and are executed in an instance supporting the corresponding programming language where the function is written. These instances are managed by the cloud platform provider, which can decide whether to create new instances or shut down old instances after some time of inactivity. Therefore, developers cannot rely on the instance's state between two function calls and have to assume that the instance is stateless. This statelessness enables the cloud platform provider to scale the functions automatically.

However, since the state is not guaranteed to be kept between serverless function calls, other services have to be used to keep the state of the application. These are typically other cloud platform services like data storage services where data can be stored between function calls. The combination of serverless functions with other cloud services builds a serverless application.

However, the complexity that arises from the combination of serverless functions and other cloud services needs to be tested. Since it is difficult for developers to know if a test suite is complete, integration testing criteria can help evaluate test suites and support the identification of missing test cases. By interviewing experts in the domain of serverless computing, integration testing was identified as a crucial problem in (Lenarduzzi et al., 2021). Furthermore, the need for coverage criteria on the integration level and the challenge of measuring coverage was identified in (Lenarduzzi and Panichella, 2021).

In our recent work, we tackled these problems by introducing a model for serverless applications in (Winzinger and Wirtz, 2019b) and establishing testing criteria for integration testing in (Winzinger and Wirtz, 2019a; Winzinger and Wirtz, 2020). Furthermore, we showed how the testing criteria can be measured in (Winzinger. and Wirtz., 2021) and how test cases can be automatically generated in (Winzinger and Wirtz, 2022). The criteria were finally evaluated in (Winzinger and Wirtz, 2023).

In this work, we present our framework for evaluating integration testing criteria in serverless applications that summarizes the previous work and contributs the processes needed for the evaluation of the criteria. In contrast to other work (Frankl and Weiss, 1991; Hutchins et al., 1994), where test criteria were evaluated at the code level, our approach focuses on integration test criteria and supports the automatic generation of test cases.

[a] https://orcid.org/0000-0002-4526-286X
[b] https://orcid.org/0000-0002-0438-8482

Winzinger, S. and Wirtz, G.
A Framework for Evaluating Integration Testing Criteria in Serverless Applications.
DOI: 10.5220/0013283100003950

In *Proceedings of the 15th International Conference on Cloud Computing and Services Science (CLOSER 2025)*, pages 167-173
ISBN: 978-989-758-747-4; ISSN: 2184-5042

2 FRAMEWORK FOR EVALUATING INTEGRATION TESTING CRITERIA

We provide an approach for the evaluation of integration testing criteria in serverless applications. A rough overview of the creation of test cases and their evaluation of their fault detection potential is given in Figure 1 showing the workflow.

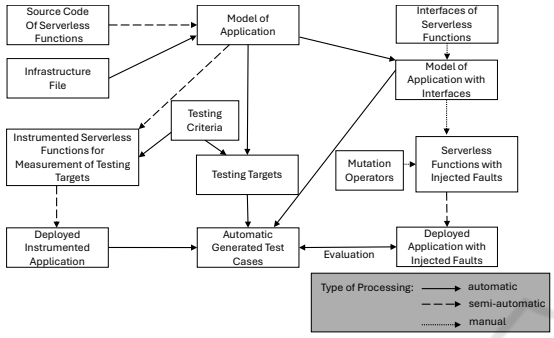

Figure 1: Workflow of framework.

The testing criteria require certain aspects of an application to be tested. The aspects required for the testing criteria can be abstracted away in a model representing the application which builds the foundation of the process. This model should contain all relevant information required for the identification of coverage criteria and later steps in the evaluation process. By using the source code of the serverless functions of the application and the infrastructure file, the file can be created semi-automatically.

Based on the model and testing criteria, testing targets that need to be covered to meet the corresponding coverage criteria can be identified. Depending on the testing criteria, a more or less descriptive model is required.

These testing criteria are used in the next step to create test cases, where at least one test case is created for each testing target. Ideally, these test cases are created automatically to support the objectivity of the created test cases. The automatic creation requires additionally an instrumented version of the serverless application which can measure the testing targets covered and a model of the application which provides interfaces for the containing serverless functions.

These test cases are evaluated for fault detection potential using mutation testing. In mutation testing, mutation operators are needed to define how a fault is injected into an application. Such an application is called a mutant, which is finally deployed. Test cases are executed on these mutants to measure the fault detection potential. Based on the measurement of the fault detection potential of the single test cases, the criteria are finally evaluated.

In the following, the phases and conditions are described in more detail.

2.1 Model Creation

In our approach, applications are modeled as a dependency graph where the nodes represent the components of the application. These components can be divided into serverless functions, data storage components, and other services provided by the cloud platform. The arcs between the components initialize the start of the action between a component, e.g., if an arc from a serverless function is directed to a data storage, it indicates that the serverless function uses the data storage. Furthermore, the arcs are annotated with additional information when a data storage component is accessed, indicating the kind of CRUD operation used.

Additionally, the serverless functions contain information about their interface since this information is needed to invoke the serverless functions for testing directly. To support data flow testing, information about the location where the data are created and used is also saved in the model.

The creation of such a model can be semi-automated by reading the infrastructure file of the application and the source code of the serverless functions. While the infrastructure file can be used to identify all components of the applications and some relations between them, the source code has to be parsed to identify all relations, interfaces, and data flow information between these components, which can be supported by a good parser, but not fully automated.

2.2 Test Criteria

Based on the model, both control flow and data flow criteria were selected.

Taking inspiration from (Linnenkugel and Mullerburg, 1990), where the criteria are defined at the module level, the following coverage criteria focusing on the control flow are suggested for serverless applications in (Winzinger and Wirtz, 2019a; Winzinger and Wirtz, 2020)which can be applied by using the previously defined model.

- *All-resources (AllRes)* requires that every resource is executed at least once. A resource is the instance of a service like a serverless function or a data storage.
- *All-resource-relations (AllRel)* requires that every relation between resources is executed (e.g., all edges between the nodes are covered).

Similarly to the control flow criteria, the data flow criteria consider interactions where several resources are involved. Thus, inspired by (Linnenkugel and Mullerburg, 1990), data flow criteria are suggested in the following:

If x is a definition of a value within a serverless function that is used by another resource, then:

- *All-resource-defs* (AllDefs) requires that every x is at least used once in another resource without being redefined before its usage.
- *All-resource-uses* (AllUses) requires that every x is used by all usages of x in other resources without being redefined before its usage.
- All-resource-defuse (*AllDefUse*) requires that:
  - each definition of a value within a serverless function that 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.

These criteria are the criteria that are investigated in the approach. If other criteria are selected or proposed, it might be required that the model of the previous step is adapted to provide all relevant information. Furthermore, the following steps might also need adaptations to produce adequate test cases.

2.3 Measurement of Testing Targets

The coverage of testing targets must be measured when test cases are executed, which is required both for the automatic generation of test cases and the final evaluation where the potential of the coverage criteria is evaluated. Therefore, in (Winzinger. and Wirtz., 2021) a general approach was introduced to measure the data flow coverage criteria mentioned above using serverless functions. In addition to this work, we also extended our approach to measure control flow criteria. By passing the relevant information between the components needed for the data flow and the control flow criteria and evaluating these values, the coverage can be measured.

Figure 2 shows a box plot with the median execution times when a serverless function calls another serverless function synchronously on *AWS* with different measurements. In addition to the previous work and the control flow and data flow measurement, *Open-Telemetry*, a standardization approach to effectively observe applications, was measured using *Jaeger* as a tracking backend. Also, Amazon's *X-Ray* was additionally used for comparison, which is the monitoring solution of Amazon that can be applied to trace and visualize the resources used by a request.

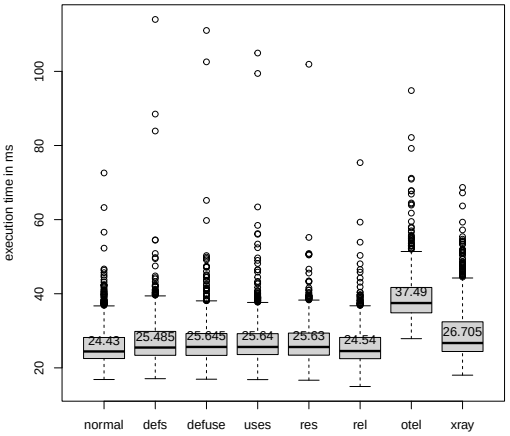

Figure 2: Measurement of coverage.

The measurement was also applied to two other scenarios in which a data storage service was used. For all three scenarios and all the criteria implemented, the effect size was calculated indicating how strong the overhead of the instrumentation was. For all scenarios and criteria implemented, the effect size was less than 0.19 which is still small according to (Cohen, 1992) while the other measurement approaches showed an effect size of at least 0.35.

2.4 Automatic Generated Test Cases

For the evaluation, test cases are automatically generated according to (Winzinger and Wirtz, 2022) which saves time in the generation of numerous test cases and makes the creation more objective.

For each testing target, a specific test case is generated. The approach for the creation of test cases identifies the first potential serverless function that might have to be invoked to cover the specific testing target. This is done statically, by analyzing the model of the serverless function and analyzing related components of the serverless functions where the testing targets are located. In the second step, the serverless functions are executed with concrete data which are generated according to the following heuristics.

- **Same Input:** Use the value of a key-value pair of a previous input with the same key.
- **Random Input:** Use the value of a random key-value pair of a previous input.
- **Same Output:** Use the value of a key-value pair of a previous output with the same key.
- **Random Output:** Use the value of a random key-value pair of a previous output.

- **Same Values:** Use the same random value for all values.

Up to ten different input data were generated for each heuristic and structure of serverless functions that was identified for each testing target. Table 1 shows for three different applications the coverage where for each application more than 70% of the testing targets could be covered.

Table 1: Automatic created test cases.

	Testing Targets	Automatically Covered	Manually Covered
Application 1			
All-Resources	17	17 (100%)	0 (0%)
All-Relations	18	18 (100%)	0 (0%)
All-Defs	12	9 (75.00%)	3 (25.00%)
All-Defuse	18	16 (88.89%)	2 (11.11%)
All-Uses	38	24 (64.16%)	14 (36.84%)
Σ	103	84 (81.55%)	19 (18.45%)
Application 2			
All-Resources	11	10 (90.91%)	1 (9.09%)
All-Relations	15	12 (80.00%)	3 (20.00%)
All-Defs	5	5 (100%)	0 (0.00%)
All-Defuse	11	11 (100%)	0 (0.00%)
All-Uses	14	14 (100%)	0 (0.00%)
Σ	56	52 (92.86%)	4 (7.14%)
Application 3			
All-Resources	22	22 (100%)	0 (0.00%)
All-Relations	56	48 (85.71%)	8 (14.29%)
All-Defs	14	11 (78.57%)	3 (21.43%)
All-Defuse	57	43 (75.44%)	14 (24.56%)
All-Uses	244	163 (66.80%)	81 (33.20%)
Σ	393	287 (73.03%)	106 (26.97%)

For testing targets where no test case was created automatically, test cases were created manually keeping them as small as possible to avoid unnecessary coverage of other testing targets.

For each test case created for a testing target, ten random test cases were also created that contain the same number of serverless functions that are invoked directly as the original test case with random input data. In addition, the input data for the test cases are generated randomly.

However, this approach produces test cases that cover the testing targets without having a test case oracle indicating how the test case should behave. Just looking for errors thrown during test case execution is not enough, since these errors can also be an intended behavior (Barr et al., 2015). Therefore, a test oracle is needed to check if the application behaves as intended when a test case is executed.

The original application is chosen as the test oracle producing the intended behavior since test cases are needed for the evaluation to detect injected faults. The results and logs generated by the functions were recorded for each test case and assigned to the test case. The dynamic parts of the outputs, such as time stamps, were identified and excluded, as the outputs are not constant when the test case is executed again.

2.5 Measurement of Injected Faults

Since there are rarely enough real faults available that can be used for the evaluation of the testing criteria, faults are deliberately injected into the application. This is done using mutation operators, which define how a fault is injected into the application and are used to predict the effectiveness of test cases and test suites for real faults (Andrews et al., 2005; Andrews et al., 2006; Just et al., 2014). The available mutation operators described in (Delamaro et al., 2001a; Delamaro et al., 2001b; Vincenzi et al., 2001; Ghosh and Mathur, 2001; Rodríguez-Baquero and Linares-Vásquez, 2018) either change or remove the potential data that are passed between the components. Therefore, the following mutation operators were defined by changing the data that are usually transmitted as a key-value pair in JSON:

- **DelKey:** Delete key of key-value pair
- **DelVal:** Delete value of key-value pair
- **RepVar:** Replace the value of the variable with a random value
- **NegVar:** Negate boolean value
- **RemRet:** Remove return statement
- **RemCal:** Remove call to other resource

Figure 3 shows the interfaces of serverless interfaces that we identified in our previous work (Winzinger. and Wirtz., 2021).

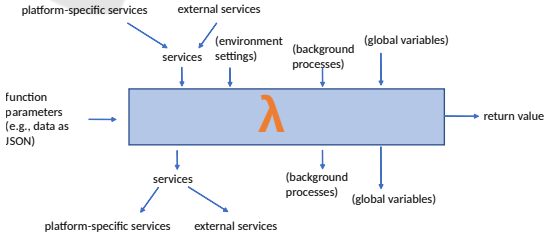

Figure 3: Data interfaces of serverless functions.

We applied the operators to the function parameters, the return values, and the values passed to and received from the services.

For each mutant generated by the mutation operator, a new version of the program is needed and has to be deployed. By instrumenting the source code with feature flags that can be activated by environment variables, a single version of the program was

enough where the faults could be activated individually. Each test case was run with each mutant to see if it could detect the injected fault.

Table 2 shows how many mutants could be killed by all test cases belonging to a certain criterion where the corresponding randomly generated test cases killed for most of these suites fewer mutants.

2.6 Evaluation of Integration Testing Criteria

All automatically and manually generated test cases for the testing targets and their randomly generated counterparts were added to a pool to build test suites that cover a certain target coverage of a specific testing target.

This test pool was used for the creation of test suites of different sizes focusing on the fulfillment of the coverage criteria investigated. For each coverage criterion and each x, where x is a number between 1 and the maximum number of feasible testing targets, 200 test suites were created which cover at least x testing targets of the corresponding coverage criterion to have a broader space of test suites fulfilling a certain coverage value. The algorithm shown in Algorithm 1 was applied which is adapted from (Andrews et al., 2006).

Algorithm 1: Test Suite Generation from Test Pool.

Require: Test Pool P, Target Coverage t
  $suite \leftarrow \emptyset$
  $suiteCoverage \leftarrow 0$
  while $suiteCoverage < t$ **do**
    $tc \leftarrow$ random tc of P
    $P \leftarrow P \setminus \{tc\}$
    $newCoverage \leftarrow$ coverage of $suite \cup \{tc\}$
    if $newCoverage > suiteCoverage$ **then**
      $suite \leftarrow suite \cup \{tc\}$
      $suiteCoverage \leftarrow$ coverage of $suite$
    end
  end
  return $suite$

A similar test suite with a similar number of test cases was created for each of the test suites created. These new test suites were constructed by randomly selecting test cases from the test pool without rejecting any test case to have test suites of equal size and make them more comparable since the same number of test cases are used.

Since the number of mutants killed by the test suites was not normally distributed, the Mann-

Table 2: Coverage of all test cases for criteria (max number of covered mutants of the best corresponding random suite in brackets).

	All Res	All Rel	All Defs	All DefUse	All Uses
App1					
DelKey	136 (115)	136 (114)	156 (136)	178 (159)	182 (170)
DelVal	136 (115)	136 (114)	156 (136)	178 (159)	182 (170)
RepVar	57 (49)	57 (49)	123 (71)	139 (84)	147 (104)
NegVar	15 (12)	15 (13)	15 (16)	18 (17)	19 (17)
RemRet	14 (12)	14 (12)	12 (14)	15 (15)	16 (15)
RemCal	8 (7)	8 (8)	13 (10)	16 (10)	16 (11)
Σ	366 (308)	366 (306)	475 (367)	544 (443)	562 (487)
App2					
DelKey	42 (35)	42 (35)	37 (34)	39 (35)	39 (35)
DelVal	45 (36)	45 (36)	40 (35)	42 (36)	42 (36)
RepVar	45 (27)	47 (30)	45 (27)	44 (31)	44 (31)
NegVar	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
RemRet	9 (7)	7 (8)	5 (8)	6 (8)	6 (8)
RemCal	4 (3)	4 (3)	3 (3)	4 (3)	4 (3)
Σ	145 (108)	145 (109)	130 (107)	135 (113)	135 (113)
App3					
DelKey	162 (130)	265 (168)	214 (162)	284 (169)	314 (195)
DelVal	190 (152)	312 (196)	253 (190)	334 (198)	365 (225)
RepVar	172 (132)	327 (181)	271 (171)	361 (180)	373 (206)
NegVar	1 (1)	10 (1)	10 (1)	10 (1)	10 (1)
RemRet	16 (13)	26 (18)	15 (16)	29 (18)	32 (24)
RemCal	18 (15)	41 (20)	35 (18)	45 (21)	48 (22)
Σ	559 (440)	981 (583)	798 (558)	1063 (585)	1142 (672)

Whitney test was chosen with a significance level of 1% to compare them and measure their effect sizes if the null hypothesis had been rejected.

Figure 4 visualizes the comparison of the test suites generated for *AllUses* and their counterparts generated for an application where the interpolated lines indicate that *AllUses* kills more mutants if the test cases were selected with this criterion.

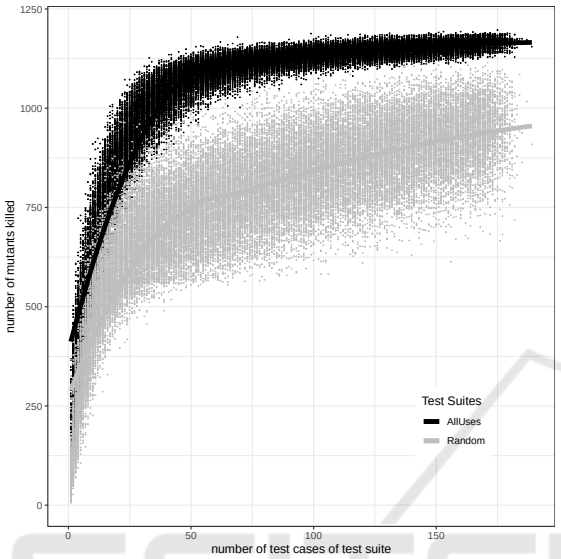

Figure 4: Comparison of test Suites for *AllUses* and random created suites of an application.

The systematically generated test suites detected significantly more mutations for all applications and criteria than their randomly generated counterparts. For the data flow criteria, the effect size was greater than 0.5, which indicates a strong effect according to (Cohen, 1992), and supports the hypothesis. However, only the test suites for *App2* and the criterion *AllRel* had an effect size greater than 0.5 for the control flow criteria. For the other applications, the effect sizes ranged from 0.11 to 0.28, indicating only a small effect.

In addition to the previous work, a significant advantage of the systematically created test cases compared to the random ones could also be shown when the manually generated test cases and their randomly assigned counterpart were excluded from the test suite.

Furthermore, also when the number of mutants was reduced to mutants on the interfaces where direct communication to other components occurred, the criteria-based test suites were significantly better for the test suites built with and without manually added test cases.

All criteria are also compared with each other considering their test suites that fulfilled 100% of their testing targets. Since the data flow criteria build a subsumption hierarchy, the stricter data flow criteria killed at least as many mutants as their subsumed ones. There was a strong effect that could be shown for most of the cases while *AllDefUse* showed no significant improvement compared to *AllDefs* in *App2*. This could be due to the architecture of *App2* where the number of potential uses for each definition is lower than in the other applications. Furthermore, the workflow of *App2* triggers several serverless functions that cause the coverage of additional testing targets within a test case. Thus, it is possible to meet the stricter coverage criteria with a relatively small number of test cases in *App2*, in contrast to the other applications where the more stringent coverage criteria must execute more test cases to be fulfilled, resulting in a higher coverage.

As a result, it could not be demonstrated that data flow criteria consistently outperform control flow criteria, since the detection potential is highly dependent on the application.

3 CONCLUSION AND FUTURE WORK

This work showed how integration testing criteria can be measured for serverless applications and applied the approach to some integration testing criteria on three serverless applications. The evaluated criteria were shown to be more efficient than random-generated test cases. However, there is no best criterion among the evaluated criteria. The data or control flow criteria are more efficient depending on the application. Therefore, we recommend not only relying on a single criterion but combining them with other criteria and test cases created based on the testers' experience.

For future work, more criteria can be evaluated using the framework introduced. Furthermore, if an application is available in which many real-world faults have already been detected, it would be worthwhile to evaluate the criteria based on these faults. Also, microservices pose an interesting area where our approach can be adapted transferring the criteria and tooling to this area.

REFERENCES

Andrews, J., Briand, L., Labiche, Y., and Namin, A. (2006). Using mutation analysis for assessing and comparing

testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624.

Andrews, J. H., Briand, L. C., and Labiche, Y. (2005). Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on Software engineering - ICSE '05*. ACM Press.

Baldini, I., Cheng, P., Fink, S. J., Mitchell, N., Muthusamy, V., Rabbah, R., Suter, P., and Tardieu, O. (2017). The serverless trilemma: Function composition for serverless computing. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - Onward! 2017*. ACM Press.

Barr, E. T., Harman, M., McMinn, P., Shahbaz, M., and Yoo, S. (2015). The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525.

Cohen, J. (1992). Statistical power analysis. *Current Directions in Psychological Science*, 1(3):98–101.

Delamaro, M., Maidonado, J., and Mathur, A. (2001a). Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247.

Delamaro, M. E., Maldonado, J. C., Pasquini, A., and Mathur, A. P. (2001b). Interface mutation test adequacy criterion: An empirical evaluation. *Empirical Software Engineering*, 6(2):111–142.

Frankl, P. G. and Weiss, S. N. (1991). An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria. In *Proceedings of the symposium on Testing, analysis, and verification*, TAV-4. ACM.

Ghosh, S. and Mathur, A. P. (2001). Interface mutation. *Software Testing, Verification and Reliability*, 11(4):227–247.

Hutchins, M., Foster, H., Goradia, T., and Ostrand, T. (1994). Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In *Proceedings of 16th International Conference on Software Engineering*, ICSE-94. IEEE Comput. Soc. Press.

Just, R., Jalali, D., Inozemtseva, L., Ernst, M. D., Holmes, R., and Fraser, G. (2014). Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. ACM Press.

Lenarduzzi, V., Daly, J., Martini, A., Panichella, S., and Tamburri, D. A. (2021). Toward a technical debt conceptualization for serverless computing. *IEEE Software*, 38(1):40–47.

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

Linnenkugel, U. and Mullerburg, M. (1990). Test data selection criteria for (software) integration testing. In *Systems Integration '90. Proceedings of the First International Conference on Systems Integration*. IEEE Comput. Soc. Press.

Rodríguez-Baquero, D. and Linares-Vásquez, M. (2018). Mutode: generic JavaScript and node.js mutation testing tool. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM.

Vincenzi, A. M. R., Maldonado, J. C., Barbosa, E. F., and Delamaro, M. E. (2001). Unit and integration testing strategies for C programs using mutation. *Software Testing, Verification and Reliability*, 11(4):249–268.

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.

Winzinger., S. and Wirtz., G. (2021). Data flow testing of serverless functions. In *Proceedings of the 11th International Conference on Cloud Computing and Services Science - CLOSER,*, pages 56–64. INSTICC, SciTePress.

Winzinger, S. and Wirtz, G. (2022). Automatic test case generation for serverless applications. In *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE.

Winzinger, S. and Wirtz, G. (2023). Comparison of integration coverage criteria for serverless applications. In *2023 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE.