EPA Public Access

Author manuscript

IEEE Softw. Author manuscript; available in PMC 2021 October 06.

About author manuscripts

Submit a manuscript

Published in final edited form as:

IEEE Softw. 2020 October 6; 38(1): 61–67. doi:10.1109/ms.2020.3029468.

Scientific Software Testing Goes Serverless: Creating and Invoking Metamorphic Functions

Xuanyi Lin,

University of Cincinnati

Michelle Simon,

U.S. Environmental Protection Agency

Nan Niu

University of Cincinnati

Abstract

Our function-as-a-service (FaaS) framework transformed end users' questions into automated tests for scientific software. Our case study illustrates the FaaSification of scientific software testing and the importance of value-driven evaluations by focusing on real-world defect detection.

MANY SCIENTIFIC RESEARCHERS rely on software to perform their research. In climate research, for instance, scientists and policy makers rely on code simulations to inform their recommendations and decisions.¹ Because scientific software needs to produce trustworthy results, rigorous software engineering practices should be adopted to assure quality.

Although testing is a mainstream approach toward software quality, scientific software developers face many challenges,² one of which is that only a small group of scientific domain experts (SDEs) who are software developers themselves could write the tests in relation to the scientific theories and models, and yet their primary goal is to further science.³ Fortunately, science is a community endeavor, and the user community is the cornerstone underlying the sustainable development of any scientific software. However, without the proper tools, serving the user community poses additional burdens on the SDE developers.

To alleviate the burdens, we present a function-as-a-service (FaaS) framework aiming to support the SDEs in efficiently processing software users' questions and further transforming the questions into automatic tests. Because users raise their concerns on an as-needed basis, building a monolithic service (for example, via the traditional web client-server architecture) with availability and capacity would not be cost-effective. FaaS breaks the monolith into functions that can be separately instantiated, executed, and destroyed, and hence, reduces the operational cost. The functions in our work are extracted from the software forum's posts and transformed into system-level test cases. For example, the two functions, *increase_bio_retention_cells()* and *decrease_runoff()*, are generated from the post expressing, "the simulated surface water runoff is expected to decrease when bioretention cells are added to an area of study."

In this article, we demonstrate an automated FaaSification of scientific software testing by creating the functions, such as *increase_bio_retention_cells()* and *decrease_runoff()* for manipulating and monitoring input and output parameters, to perform metamorphic testing.^{4,5} In addition, we investigate different mechanisms for invoking the metamorphic testing functions and evaluate these mechanisms based not only on cost and performance but also on the capability of defect detection, which is directly valuable to the SDE developers. Our work illustrates a service discovery for an actual software engineering task, and our empirical results offer concrete insights into making value-driven strategies in a serverless context. Next, we describe the specific scientific software motivating our FaaS development.

Storm Water Management Model

The Storm Water Management Model (SWMM) (<https://www.epa.gov/water-research/storm-water-managementmodel-swmm>) is a software system developed by the U.S. Environmental Protection Agency (EPA) and others. The original development began in 1971, and the most recent release of SWMM 5.1.014 occurred in February 2020. SWMM computes the runoff quantity and quality from primarily urban areas. Its simulation engine implements hydrological modeling, green infrastructure controls, and more, and it is written in C/C++ with approximately 45,500 lines of code (LOC), which can be deemed as a medium scientific system (1,000—100,000 LOC).² In addition to the medium code size and the test oracle problem faced, SWMM is representative of scientific software requiring many physical parameters and having a large community (for example, active user forums).

The five decades of evolution of the software exemplifies the sustained support from the SWMM community. Thousands of SWMM studies have been carried out worldwide by hydrologists, engineers, and water resources management specialists.^{4,5} Despite the global adoptions of SWMM, the software development and maintenance of the official releases of the software remain largely local to the EPA. For the SWMM developers, supporting the user community becomes important, although oftentimes it is time consuming. For example, Open SWMM (<https://www.openswmm.org/>), one of the most vibrant user forums, received a total of 400 posts in 2019, among which about one-third expressed questions on usage, and two-thirds indicated bugs, errors, or other inquiries into the software's behavior. However, only two core SDE developers from the SWMM team actively and periodically answered users' inquiries, leaving 52 questions (13%) unresolved, even though others also contributed to the discussions in Open SWMM. Our objective is to support the SDE developer in better handling the users' posts.

Discovering and Orchestrating Metamorphic Testing Functions

FaaS deploys individual functions and charges for the time that the code is actually executing without the users having to pay for idle resources traditionally bundled in monolithic services.⁶ As the infrastructure/cloud providers (for example, Amazon, Google, and so on) take on the responsibility for capacity planning and resource optimization, the application developers need to focus only on the logic for processing client requests. Rather than continuously running servers, FaaS deploys functions that operate as event handlers,

allowing for a serverless architecture to create smaller, better-isolated modules that can be more easily maintained and replaced.⁶

Figure 1(a) presents an overview of our FaaS framework. Our experience with SWMM suggests that the SDE would periodically handle a batch of posts to maximize the efficiency in serving the scientific community. The batch, therefore, represents the request of our FaaS framework. The main contribution of our work is to use FaaS to make the metamorphic testing of scientific software more cost-effective. The remainder of this section details the generation and orchestration of the metamorphic testing functions and discusses the results of our SWMM case study.

Function Creation

In our FaaS framework, the creation of the decoupled functions is driven by whether a software user's post can be transformed into automatic tests, as shown in Figure 1(a). For the posts that our generator fails to convert (for example, questions about an input file's format), we ask the SDE to handle them manually. Note that our framework aims to augment but not to replace the SDE.

Our generator attempts to turn a post into metamorphic tests.^{4,5} In its simplest form, such a test consists of a change in the input and a corresponding and expected change in the output. In SWMM, for example, judging whether a single software execution's runoff is correct requires a test oracle which may not be available in the scientific software.² Metamorphic testing would run SWMM twice: one source test case with an existing subcatchment and a follow-up test case by increasing a node's surcharge while keeping all the other inputs intact. Thus, we would expect the follow-up test case's runoff to be greater than the source test case's runoff, effectively overcoming the lack of a test oracle. In this example, *increase_node_surcharge()* and *increase_subcatchment_runoff()* are two functions corresponding to input and output changes, which we refer to as *metamorphic testing functions*.

For scientific software like SWMM, the behavior relies on a large multiparameter input space.² This motivates us to construct multiple functions to separately carry out metamorphic testing. Figure 1(a) illustrates our tool pipeline for turning a post into metamorphic testing functions: using Python's natural language toolkit (NLTK) (<https://www.nltk.org/>) to tokenize a post's content, the fuzzywuzzy implementation of the Levenshtein distance (<https://github.com/seatgeek/fuzzywuzzy>) to match a token to a list of manually codified SWMM input/output (I/O) variables, and WordNet's synsets (<https://wordnet.princeton.edu/>) to recognize and unify a list of our predefined verb set: {"alter," "change," "decrease," "drop," "grow," "increase," "reduce," "rise," "switch," and "vary"}. Figure 2(a) provides some postfunction mapping examples.

Function Invocation

As FaaS makes the charge on a per-invocation basis,⁷ it is crucial to decide which functions to execute in parallel in the container and which ones to deploy in a later round of service computation. We define *round* to be a unit that runs a subset of the metamorphic testing

functions in the FaaS container, and the remaining functions have either completed their executions or are waiting in the queue. Figure 2(b) shows that the first decision node is triggered if a bug is found relative to Post_j in the batch where an expected output relation in Post_j fails to hold. We then inform the SDE about Post_j and remove all of the functions associated with Post_j from the current round, R_i , as well as from the function queue. Meanwhile, we resubmit the functions of R_i whose executions have not reached 5 min⁶ to the next round, R_{i+1} . The second decision node of Figure 2(b) demonstrates that once a function runs 5 min and does not reveal any bug, it is removed from the function queue and will not be resubmitted to R_{i+1} . This process iterates until the function queue is empty, indicating that all of the metamorphic testing functions from the batch have been executed. Different invocation mechanisms set different priorities and hence incur different costs. To explore the best mechanism to deploy our FaaS framework, we investigate spatial locality, temporal timeliness, and a random invocation baseline.

- *Random:* From the identified functions, a subset is chosen randomly at each round to run the metamorphic tests.
- *Warm caches:* Baldini et al.⁸ preferred function invocations and took advantage of the warm caches in a container. Amazon Web Service's (AWS) Lambda freeze/thaw cycle shares a similar priority of reusing the hot container. Following this mechanism, the more frequently occurred functions [for example, fn_1 in Figure 2(a)] will be invoked prior to less common ones (for example, fn_2).
- *Speedy first:* Kijak et al.⁹ realized the importance of the FaaS deadline constraint and separated short runtime functions from slower ones. In our SWMM case study, we developed a speed predictor based on the empirical results of 200 randomly sampled metamorphic testing functions. A function is estimated to be faster if the expected simulation period is shorter, the input file is smaller, and the input parameter is less involved in the floating-point operations. Figure 2(a) illustrates each function's estimated speed (slow, medium, or fast); speedy functions will be invoked ahead of less speedy ones.

Results

Our Python code crawled 5,362 Open SWMM posts and batched them chronologically on a biweekly interval, mimicking an SDE's use of our FaaS framework to help process user posts on a biweekly basis. Each batch averaged 18 posts, 67% of which were successfully transformed into metamorphic testing functions with our tool pipeline, shown in Figure 1(a) (averaging 4.5 functions per post).

OpenFaaS (<https://www.openfaas.com/>) is an open source project built in 2016. Mohanty et al.¹⁰ demonstrated OpenFaaS's high adoption rate among four serverless frameworks. Therefore, we used OpenFaaS to implement our metamorphic testing functions and performed our experiments by using a two core, 4-GHz CPU with a 1-GB container. We ran the experiment iteratively on OpenFaaS for all of the Open SWMM batches and initiated a container for deploying each batch, followed by destroying the container when the batch's processing was completed. Figure 3(a) indicates the number of SWMM bugs found and the

cumulative FaaS deployments made. It is unsurprising that the random deployment requires more rounds to detect defects; however, the warm caches mechanism remains close to the random baseline in Figure 3(a). Speedy first, in contrast, shows superior performance, especially as the number of detected bugs increases.

Paying attention to the discovered bugs delivers direct value to the SDE and also makes the FaaS performance–cost analysis more sensible. Figure 3(b), therefore, plots the runtime and the memory usage of 100 rounds in line with the distribution patterns of Figure 3(a). Figure 3(b) helps visualize the time–space tradeoffs. Warm caches localize the memory usage, yet the runtime performance is compromised. Using the normative rate of AWS’s Lambda (US \$0.000000834/100ms for 512-MB service),⁶ we estimate that the total FaaS costs of the random, warm caches, and speedy first mechanisms are US\$1.10834096, US\$0.89947624, and US\$0.70758582, respectively, with speedy first being a winning way of function invocation in our SWMM case study.

Although threats to validity exist (especially the generalizability of our results beyond SWMM and our experimental settings), our findings can offer insights into a pragmatic mindset to support the scientific software developers in a serverless context. First, a pure FaaS approach might not be able to fully handle a software testing task. In our framework, a local server running SWMM is instrumented alongside OpenFaaS. However, we believe that FaaS provides a lightweight front end to accept a batch of functions, and the speedy first invocation strategy enables the dynamic execution of these functions to deliver tangible values to the software engineers. Manually handling users’ posts typically costs a SWMM SDE a couple of hours weekly; compared to the manual work, FaaS offers a batch processing mode allowing automated, metamorphic tests to be performed as needed and within minutes. Second, due to the hybrid model of FaaS and local server architectures, trust and security issues arise. Our current assumption is that the SDE developer would make trustworthy requests to the FaaS framework. However, in more distributed settings, a trust gatekeeper,⁶ or functions serving as gatekeepers, may be required, which in turn adds complexity into the space–time tradeoffs.

FaaS dynamically deploys multiple, less correlated computations, but the cost–performance evaluations have been based on functions like floating-point operations, numerical calculations, and file conversions.^{6,7} To software engineers, these functions are value neutral. Our work not only automates the FaaSification of a concrete software engineering task, but it also illuminates the value-driven evaluation of serverless computing in terms of bug detection and computation cost. The speedy first mechanism yielded promising results, which saved 26% runtime on average and 36% of AWS’s Lambda expenditures in comparison with the random mechanism. In addition to using FaaS to make scientific software testing more cost-effective, our work also contributes to the hybrid architectures of FaaS and a local server responsible for running and testing SWMM. Our multithreaded implementation on the local server is in line with the parallelization emphasized by Spillner et al.¹¹ Our ongoing work explores FaaS visual feedback, hardens request security, generates more adequate test cases,¹² and investigates the extension of FaaS to a Representational State Transfer application.

Acknowledgments

This work is supported in part by the U.S. National Science Foundation (award CCF 1350487) and the U.S. Environmental Protection Agency. The U.S. Environmental Protection Agency (EPA), through its Office of Research and Development, partially funded and collaborated on the research described herein. It has been subjected to the Agency's peer and administrative review and has been approved for external publication. Any opinions expressed in this article are those of the authors and do not necessarily reflect the views of the Agency therefore, no official endorsement should be inferred. Any mention of trade names or commercial products does not constitute endorsement or recommendation for user.

Biography

XUANYI LIN is a Ph.D. candidate in the University of Cincinnati's Department of Electrical Engineering and Computer Science, Cincinnati, Ohio, USA. His research interests include automated software testing, scientific software development, and requirements engineering. Lin received his M.Sc. in computer science from the University at Albany, State University of New York. Contact him at linx7@mail.uc.edu.

MICHELLE SIMON is a senior chemical engineer at the U.S. Environmental Protection Agency Office of Research and Development, Cincinnati, Ohio, USA. Her research interests include stormwater modeling, green infrastructure, and wastewater infrastructure. Simon received her Ph.D. in environmental science from the University of Arizona. Contact her at simon.michelle@epa.gov.

NAN NIU is an associate professor in the University of Cincinnati's Department of Electrical Engineering and Computer Science, Cincinnati, Ohio, USA. His research interests include requirements engineering, scientific software development, and human-centric computing. Niu received his Ph.D. in computer science from the University of Toronto. He is a Senior Member of IEEE. Further information about him can be found at <https://homepages.uc.edu/~niunn/>. Contact him at nan.niu@uc.edu.

References

1. Easterbrook S and Johns TC, "Engineering the software for understanding climate change," *Comput. Sci. Eng.*, vol. 11, no. 6, pp. 65–74, Nov-Dec 2009. doi: 10.1109/MCSE.2009.193.
2. Kanewala U and Bieman JM, "Testing scientific software: A systematic literature review," *Inform. Softw. Technol.*, vol. 56, no. 10, pp. 1219–1232, Oct. 2014. doi: 10.1016/j.infsof.2014.05.006.
3. Basili VR et al., "Understanding the high-performance-computing community: A software engineer's perspective," *IEEE Softw.*, vol. 25, no. 4, pp. 29–36, Jul-Aug 2008. doi: 10.1109/MS.2008.103.
4. Lin X, Simon M, and Niu N, "Hierarchical metamorphic relations for testing scientific software," in *Proc. IEEE/ACM 13th Int. Workshop Softw. Eng. Sci. (SE4Science)*, Gothenburg, Sweden, 6 2018, pp. 1–8. doi: 10.1145/3194747.3194750.
5. Lin X, Simon M, and Niu N, "Exploratory metamorphic testing for scientific software," *Comput. Sci. Eng.*, vol. 22, no. 2, pp. 78–87, Mar-Apr 2020. doi: 10.1109/MCSE.2018.2880577.
6. Adzic G and Chatley R, "Serverless computing: Economic and architectural impact," in *Proc. 11th Joint Meeting Found. Softw. Eng. ESEC/FSE*, Paderborn, Germany, Sept. 2017, pp. 884–889. doi: 10.1145/3106237.3117767.
7. Bortolini D and Obelheiro RR, "Investigating performance and cost in function-as-a-service platforms," in *Proc. Int. Conf. P2P, Parallel, Grid, Cloud Internet Comput.(3PGCIC)*, Antwerp, Belgium, Nov. 2019, pp. 174–185. doi: 10.1007/978-3-030-33509-0_16.
8. Baldini I et al., "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*, Chaudhary S, Somani G, and Buyya R Eds., New York: Springer-Verlag, 2017, pp. 1–20.
9. Kijak J, Martyna P, Pawlik M, Balis B, and Malawski M, "Challenges for scheduling scientific workflows on cloud functions," in *Proc. IEEE 11th Int. Conf. Cloud Comput. (CLOUD)*, San Francisco, 7 2018, p. 460–467. doi: 10.1109/CLOUD.2018.00065.
10. Mohanty SK, Premsankar G, and Francesco MD, "An evaluation of open source serverless computing frameworks," in *Proc. CloudCom*, Nicosia, Cyprus, Dec. 2018, pp. 115–120.
11. Spillner J, Mateos C, and Monge DA, "FaaSter, better, cheaper: The prospect of serverless scientific computing and HPC," in *Proc. Latin Amer. High Performance Comput. Conf. (CARLA)*, Buenos Aires, Argentina, Sept. 2017, pp. 154–168. doi: 10.1007/978-3-319-73353-1_11.
12. Peng Z, Lin X, and Niu N, "Unit tests of scientific software: A study on SWMM," in *Proc. Int. Conf. Comput. Sci. (ICCS)*, Amsterdam, The Netherlands, 6 2020, pp. 413–427. doi: 10.1007/978-3-030-50436-6_30.

FIGURE 1.

(a) An overview of our FaaSification of batch processing user forum's posts. (b) The FaaS hosts metamorphic testing functions that modify the input parameters and monitor the output parameters, which are coordinated with a local software as a service (SaaS) server that runs and tests SWMM in a multithreaded way. (c) Code snippets illustrating the metamorphic testing function generations (decreasing the "routing step" input is expected to decrease the "flow" output) as well as the local server coordination.

	fn_1	fn_2	fn_3	fn_4	fn_5	fn_6
	I: Routing Step	I: Time Series	I: Rain Gauge	I: Surcharge	O: Runoff	O: Flow
Post ₁	decrease()		change()			decrease()
Post ₂	change()	change()	change()		change()	change()
Post ₃				increase()	decrease()	
Frequency	2	1	2	1	2	2
Speed	Fast	Fast	Medium	Slow	Slow	Medium

(a)

(b)

FIGURE 2.

(a) The mapping between posts and FaaS functions and (b) the batch running metamorphic testing functions in the FaaS container.

FIGURE 3.

(a) The value to the SDE developer measured by the number of SWMM defects detected and the cumulative FaaS rounds computed and (b) the runtime and memory utilization distributions over 100 rounds of FaaS computation.