On the Feasibility of Serverless Functions in the Context of Auto-Graders

Sebastian Serth
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
sebastian.serth@hpi.de

Maximilian Paß
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
maximilian.pass@student.hpi.de

Christoph Meinel
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
christoph.meinel@hpi.de

2023 IEEE 2nd German Education Conference (GECon) | 979-8-3503-4813-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/GECON58119.2023.10295106

***Abstract*—Learners interested in acquiring fundamental programming skills may choose from a variety of different offers, including Massive Open Online Courses (MOOCs). Usually, these courses not only include lecture videos and multiple-choice quizzes, but also feature hands-on programming exercises, allowing learners to apply their newly acquired knowledge right away. Since solving these exercises requires access to a programming tool chain, most MOOCs embed their exercises in a web-based environment supplying necessary tools. One of these so-called auto-graders is CodeOcean, which allows learners to write and run code or receive automated feedback. While a web-based auto-grader lowers the entry barrier for learners to get started, providing sufficient resources for all code executions poses an additional challenge for the MOOC provider, especially during high-demand periods. Therefore, we evaluated serverless functions as offered by cloud computing providers for the use in auto-graders and conducted a Randomized Control Trial. Although serverless functions at first appear to be slower compared to our existing containerized execution of learners' code, they convinced with more constant execution times in high-demand periods.**

I. Introduction

Acquiring fundamental programming skills requires "learning by doing" [1], i.e., writing and executing source code and observing the corresponding program behavior. Therefore, learners need access to the respective programming tools, such as a compiler, interpreter, or runtime. While some learners might feel comfortable setting up the required tools themselves, others might encounter difficulties, creating an initial hurdle before they actually begin programming [2]. Especially in Massive Open Online Courses (MOOCs) targeting beginners, the teaching team cannot support individual learners to overcome this hurdle and prepare a suitable setup on their local machines [3]. Instead, a web-based environment is desired, allowing learners to access relevant tools just with a web browser, eliminating the need for any local setup. In addition to providing an educational environment for learners to write and execute code, so-called auto-graders also assess learners' code submissions and provide automatic feedback. By integrating auto-graders with a MOOC platform, learners can apply their newly acquired knowledge in hands-on assignments and have their score reflected in the course progress.

One of the auto-graders developed for the MOOC context is CodeOcean [4]. So far, the web-based development environment has been employed in more than 50 courses teaching more than 100,000 learners Java, Python, Ruby, and R basics. Despite other features, CodeOcean allows learners to run their code and request automated feedback, which is generated by executing teacher-defined tests in the respective programming language. In the current setup, a set of backend microservices is involved to execute learners' code within predefined containers, featuring the required tools required to run code in the respective programming language.

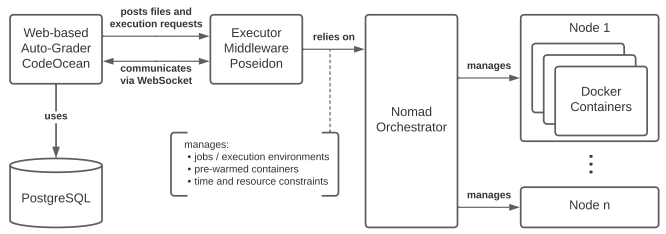
With the work at hand, we question the status quo of using containerized environments for executing code, and evaluate the feasibility of serverless functions (as defined in Section III). Therefore, we address the following research questions:

RQ1. Which impact have serverless functions on key requirements of code executions, such as execution times and technical stability?
RQ2. How do learners compare serverless functions to containerized code executions?
RQ3. Which open questions arise from using serverless functions in a programming MOOC?

II. Background and Status Quo

Our auto-grader CodeOcean was specifically designed for the use in large-scale MOOCs, with three main requirements in mind (see [5]): Interactivity (allowing real-time interactions), Scalability (for an increasing number of parallel users) and Flexibility (regarding the programming languages used). The resulting architecture, proven in MOOCs with more than 17,000 active learners, is shown in Figure 1.

For each programming language, CodeOcean uses a dedicated execution environment containing language-specific tools to run the learners' source code. So far, all execution environments were realized with Docker containers, which were assigned to a single learner upon request and destroyed after an inactivity period. To serve all learners simultaneously, we need many parallel containers, which we distribute across multiple hosts with the open-source container orchestrator Nomad. While Nomad manages the lifetime of containers, we decided to abstract from its Application Programming Interface (API) by creating an own executor middleware called Poseidon, which further manages the execution environments. For example, it enforces time constraints for containers (i.e., by limiting the



Web-based Auto-Grader CodeOcean
posts files and execution requests
communicates via WebSocket
Executor Middleware Poseidon
relies on
Nomad Orchestrator
manages
Node 1
Docker Containers
Node n
uses
PostgreSQL
manages:
• jobs / execution environments
• pre-warmed containers
• time and resource constraints

Fig. 1. The current architecture of our auto-grader CodeOcean, consisting of the front-facing web application, a custom execution middleware named Poseidon and a Nomad cluster (cf. [5]). Learners' code is executed in one of the language-specific Docker containers and output is streamed to the learners' browser through a WebSocket connection.

execution to a few seconds) and configures the containers as desired. Furthermore, Poseidon manages a pool of so-called pre-warmed containers for each execution environment, which consists of idling containers not yet assigned to any user. As soon as a new learner requests a code execution, a pre-warmed container is used to avoid delays that would otherwise occur when waiting for a new container to start.

Besides tackling the scalability through the chosen architecture, our approach still poses some requirements on the deployment of CodeOcean, Poseidon and the Nomad orchestrator as well: The availability of (many) resources. In our production environment, the Nomad cluster consists of four agents executing Docker containers and three smaller server nodes managing and orchestrating the workload, together with one server instance each for CodeOcean, a PostgreSQL database and Poseidon. In total, we allocated 240 GB RAM and 104 vCPU cores to these ten virtual machines (VMs), allowing the system to be used in a MOOC with more than 17,000 active learners (more than 40,000 enrollments) successfully. While those resources are only required for high-demand periods, we also need to maintain a "baseline" of all servers throughout the year for other courses and for learners accessing the courses in self-paced mode. Practically, we are rarely scaling our infrastructure and rather keep the system running unchanged, thus blocking the allocated resources and incurring (maintenance) costs. By evaluating serverless functions, as introduced in the next Section III, we aim to reduce the permanently-blocked resources. Potentially, this might save costs or reduce the environmental impact (since hardware resources can be better shared among various tasks).

III. Related Work

Our work is mainly based on previous research in two areas: The execution of learners' code in the context of programming education (see Section III-A) and the recent advancements of serverless functions (as introduced in Section III-B).

A. Current Execution Approaches for Auto-Graders

Learning a programming language greatly benefits from the actual programming assignments [3], as just answering questions in multiple-choice quizzes is not enough [6]. However, supporting learners to have access to a local execution environment on their system is unfeasible for large-scale MOOCs, since it would be too time-consuming for the teaching team and also poses another hurdle to begin learning [3]. Therefore, many course providers decided to provide setup-free environments to their learners, either as part of a regular learning offer or to provide automated feedback about the correctness of a solution. We are aware of three different execution models that were chosen for auto-graders so far, and argue that all models have different advantages and disadvantages.

1) Browser-based Executions: From a provider's perspective, the code written by learners can be considered "untrusted", meaning that it could be malicious or otherwise have unattended side effects. Therefore, one approach to execute code is by running it in the learners' browser, i.e., on the client side. While this approach works smoothly for JavaScript as a browser-native language [7], other programming languages require dedicated handling. For example, the so-called Online-IDE designed for Java programs includes a custom parser for Java code and transpiles it to JavaScript [8]. Unfortunately, this approach also limits the use of (external) libraries and has further limitations, such as missing support for exception handling [8].

Another approach to this problem was taken by Sharrock *et al.*, who adapted a Linux VM to work in the browser with their project called WebLinux [9]. Within the Linux instance, the authors provided access to a command line and a compiler tool chain for learners, allowing them to get started with the C programming language. While their solution works offline without being connected to the internet (once the page finished loading), the performance of the tool varies by the device used. For resource-constrained devices, such as older smartphones or tablets, this might have an impact on the user experience.

2) Dedicated Server-side Executions: The requirements on client devices can be further lowered by executing learners' code on a backend server, for example as operated by the MOOC provider. While this allows the provider to have more control over the execution environment and further eases adding arbitrary libraries, it also increases the attack vector for malicious code. Hence, previous research has focussed on securing the code executions. For some languages, such as Java already running in the Java Virtual Machine (JVM), Strickroth employed the built-in Java Security Manager to define restrictions for the execution of user code [10].

Similarly, the auto-grader Praktomat originally written for evaluating Java code, used the Java Security Manager for Java executions, but also added support for other programming languages through Docker containers [11]. According to Flauzac *et al.*, the usage of container technologies, including Docker containers, efficiently isolates an application from the host without the overhead of a traditional virtual machine [12]. Presumably, this is why many of the generic code execution platforms employ container technologies to run arbitrary code, including CodeRunner [13], codeboard.io or the ranna code runner.

3) Other Server-based Execution Approaches: Since the execution of some newly developed source code is not just unique to educational settings with an auto-grader, further, more general approaches exist. One of them is the use of a continuous integration (CI) pipeline, usually attached to a version control system such as Git, which executes pre-defined commands for a set of source files. In an educational context, a CI pipeline is mainly used to assess learners' code, but it does not allow learners to run their code interactively, which is a hard requirement for us. Instead, learners submit their code to the version control system. Depending on the configuration, the code might either be evaluated shortly after with learners having access to the feedback [14] or the code might only be evaluated once the submission deadline passed, serving as a grading hint for the teaching team [15].

B. Serverless Functions

Except for the browser-based execution, the majority of code execution modes introduced in the previous section requires a server with the corresponding resources allocated for the respective demand. In this regard, the browser-based execution model is different, since a server is only used to deliver the web page and relevant resources, but not for the actual code execution. Therefore, some authors refer to this approach as being serverless [7]. While this definition is true from a technical perspective, we refrain from using the term *serverless* for browser-based executions in this paper, but rather reserve it for use in the context of serverless functions.

Serverless functions, also referred to as Function-as-a-Service (FaaS), describe a cloud computing model where events trigger the execution of comparatively small and self-contained programs, the so-called functions [16]. In this sense, serverless functions can be seen as an evolution of microservices and containers, further splitting those components in even smaller units [17]. In contrast to regular server applications, serverless functions are only executed when triggered by an event and are rather short-living (some even restricted to a total of 15 minutes) [16]. Due to the small nature of serverless functions, they can be scaled easily to match the current demand [16]. Serverless functions premiered in 2014 on Amazon Web Services (AWS) with the launch of AWS Lambda, and are available on all major cloud platforms and for on-premise installation by today [17]. Similar to traditional cloud computing resources, customers are billed in relation to execution duration and the resources allocated for the serverless function.

IV. Concept

Our vision is to enable code executions through serverless functions in our auto-grader and thereby improve the scalability for high-demand periods. Following, we could minimize the overall resources allocated, and rather use the automatic scaling of serverless functions. In the long-term, we might even drop our Nomad cluster, which uses a majority of all resources, and completely rely on the cloud provider's infrastructure.

To achieve our vision and include serverless functions seamlessly, we plan to extend Poseidon as our existing executor middleware. We envision that only Poseidon is aware of the different execution models and abstracts from the specific details for CodeOcean as our auto-grader. This also includes another implicit requirement: The environments provided through our Nomad cluster and the cloud-based serverless functions should behave similarly, allowing all existing exercises to be used with both environments interchangeably. This not only covers how programs are executed and which libraries are available, but also refers to the ability of both environments to stream output generated by the learners' program in real time.

V. Implementation

Adding support for serverless functions to our existing architecture (as introduced in Section II) was mainly performed by customizing the executor middleware Poseidon. From the very beginning, Poseidon had a dedicated Nomad Manager, being responsible for the communication with the Nomad cluster. Similarly to the existing Nomad Manager, we added a second Manager dedicated for the interaction with serverless functions, in our case the AWS Manager. Further, we decided to implement the *chain of responsibility* pattern to decide which manager (and thereby which environment) should be responsible for handling an execution request. With a *chain of responsibility*, a request is passed in a pre-defined order from one manager to another until one of the managers is finally able to handle the request. In our case (and for the sake of our evaluation), we decided to prioritize the AWS environment, but the order can be swapped easily (i.e., to use the AWS environment only as a fallback for high-demand periods).

For AWS Lambda, we developed a small Java program, consisting of two classes and less than 250 lines of code. This program is designed to run as a serverless function and accepts an execution request with the respective code to execute. With our code being deployed to AWS Lambda, we also had to configure an API Gateway at AWS, responsible for routing external WebSocket requests (originating from Poseidon) to a newly invoked function instance. In contrast to Nomad, where preparing a runner with the desired files and actually executing the code are two separate requests, we combined them for the AWS Lambda environment to comply with the event-driven design principles of serverless functions. Figure 2 depicts the resulting architecture of our auto-grader, supporting Nomad and AWS environments simultaneously.

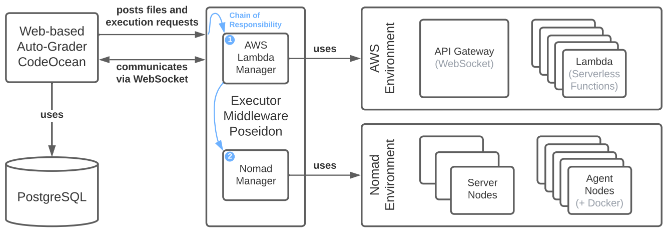


Fig. 2. The resulting architecture of our auto-grader CodeOcean. In comparison to Figure 1, the front-facing web application and the Nomad environment remain unchanged. New is the AWS environment, consisting of an API gateway listening to incoming WebSocket requests as well as the use of AWS Lambda responsible for executing learners' code. Now, Poseidon implements a *chain of responsibility* (highlighted in blue) to decide which environment is used.

VI. Evaluation

In order to assess the feasibility of serverless features in the context of auto-graders and to test our architecture as realistically as possible, we decided to perform a hands-on evaluation in one of our programming MOOCs. Together with the technical insights gained during the implementation, this study forms the baseline for answering our research questions.

A. Methodology

For our evaluation, we chose one of our regular MOOCs introducing novices to object-oriented programming in Java. In addition to videos and subsequent multiple-choice quizzes, the course included a total of 65 practical programming exercises offered through CodeOcean. Those exercises allowed learners to apply their newly acquired knowledge and receive automated feedback to fix potential issues. As introduced in Section I, learners were able to execute their code or request feedback at any time and as often as they wished. A graded certificate was rewarded to learners achieving at least 50% of all available points through weekly homework assignments (40%) and all 65 programming exercises (60%). While the course was originally offered in four weeks, the iteration used for our evaluation was stretched to a total of two months to accommodate the limited time budged available in a school context.

As part of the aforementioned Java course, we conducted a Randomized Control Trial (RCT) with all learners solving any programming exercise. Through a round-robin principle, users were automatically assigned to one of the two execution environments (Nomad with Docker containers or serverless functions with AWS Lambda). Using this RCT, we compared key metrics of code executions (such as the time taken for each program invocation) between both environments. Since we were particularly interested in high-demand periods, we focused our evaluation on the top 5% of all hours with the most code executions. We also identified low-demand periods, i.e., the 5% of hours with the fewest code executions. Furthermore, we asked all learners of the course to participate in a voluntary survey about CodeOcean. As we targeted a group of beginners that just started to learn Java, we did not include any technical details in the survey, but rather asked about their satisfaction with different components on a five-point Likert scale.

B. Results

By course end, about 1,600 of the 2,200 enrolled learners had attempted to solve a programming exercise and thus were participating in our RCT. We measured a total of 370,000 executions, of which 47% were performed using the AWS environment. Regardless of the execution environment used, we did not experience any technical issues influencing our experiment. Figure 3 visualizes the median time required for each execution (referred to as the *execution time* in the following) over the course period of two months. Especially during the second half of the course period with a lower demand of executions, we observed that the AWS environment was slower than the Nomad environment and also showed some spikes with a considerably slow code execution. Analyzing the entire course period, we identified a statistically significant difference between code executions on AWS and Nomad, with a weak correlation (Spearman's $\rho = 0.22$, $p < 0.05$).

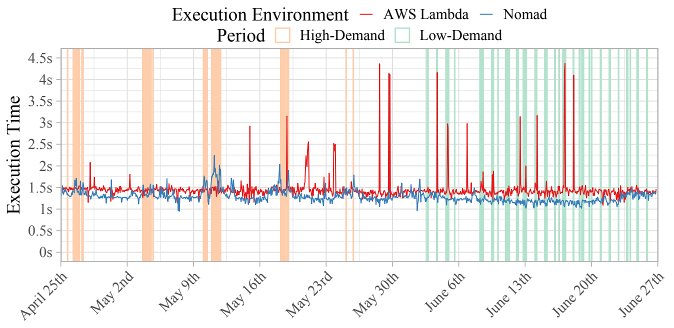


Fig. 3. The median time for each code execution throughout the course. Each line represents one of the execution environments, the colored background indicates high- and low-demand periods. Interestingly, the AWS Lambda environment irregularly showed an increased execution time of up to 4.5 seconds, which we did not observe with the Nomad environment.

When analyzing the execution time in high-demand periods, we made another observation: While the median execution time for the entire course was generally better with Nomad (1.27*s*) than with AWS Lambda (1.42*s*), AWS performed slightly better in high-demand periods (AWS 1.46*s* vs. Nomad 1.48*s*). As shown in Figures 4 and 5, the variance of our Nomad environment appears to be higher, especially in those high-demand periods. However, we were not able to confirm this trend statistically, as our data does not reveal a significant difference between executions on both environments for high-demand periods ($p = 0.19$). Also, we noticed that the median execution time on Nomad increased by 17% in high-demand periods (compared to the entire course period), whereas AWS was considerable more constant with an increase of only 3%.

In order to gain a comprehensive understanding of serverless functions in the context of an auto-grader, we also asked our learners about their subjective experience with the code executions. In total, we received 270 responses to our survey and associated those with the execution environment provided to them. Figure 6 shows the survey responses to which we

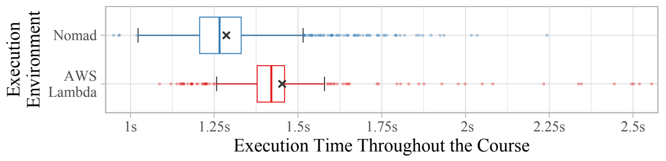


Fig. 4. Execution time for all exercises on both environments. The colored lines mark the median, the black cross shows the average. The Nomad environment is usually faster; AWS shows more (slow) outliers.

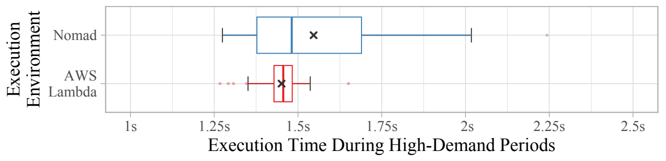


Fig. 5. Similar to Figure 4, but focussing on high-demand periods (the busiest 5% of the course period). In comparison to Figure 4, the variance of the AWS environment decreased, and fewer outliers are visible.

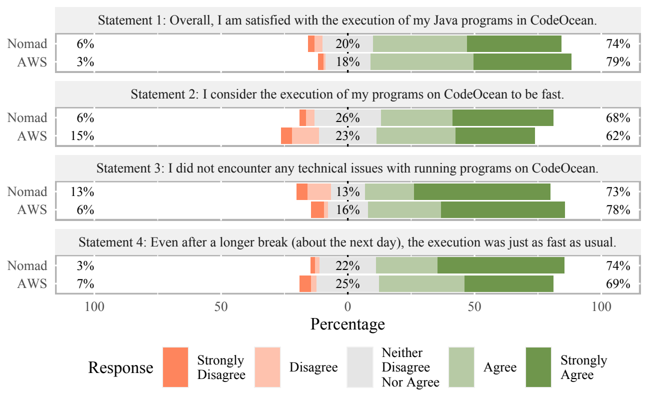


Fig. 6. An excerpt of 270 responses received to our voluntary survey, separated by the execution environment provided to learners.

refer in this paper. Overall, about 74% of all Nomad users and about 79% of all AWS users were satisfied with the code execution (Figure 6, Statement 1). The performed Mann–Whitney U test revealed no significant difference between the two groups ($U = 8853$, $p = 0.50$). Specifically asked about the duration of their code executions (Statement 2), Nomad (68% satisfaction) performed insignificant better than AWS Lambda (62% satisfaction, $U = 7552$, $p = 0.09$). Interestingly, more AWS users agreed to the statement that they did not experience technical issues than Nomad users (AWS 78% vs. Nomad 73%, Statement 3, $U = 8223$, $p = 0.93$), aside from the fact that our monitoring did not reveal any server-side issues, neither with AWS nor with Nomad. We also asked learners whether they noticed any difference between consecutive executions and those happening after a longer period of inactivity, such as those performed after taking a break. While a majority did not notice any difference, about 7% of AWS users and 3% of Nomad users reported such a difference (see Statement 4). Here, the Mann–Whitney U test revealed significant differences ($U = 6793$, $p = 0.04$) with a small effect size ($r = 0.13$).

C. Discussion

The evaluation allowed us to gain first insights about the feasibility of serverless functions in the context of an auto-grader. Considering the entire course period, the performance of AWS Lambda was not as good as the one of Nomad, and accordingly learners rated AWS lower. Especially in low-demand periods and after longer times of inactivity, AWS Lambda did not yield satisfying results. We also attribute the spikes in the execution time (as seen in Figure 3) to those low-demand periods. Hence, the accumulation of spikes in the second half of the course is not surprising to us, with more learners dropping out. Simultaneously, the performance of our Nomad environment was best in the final course weeks, most likely also caused by the lower number of learners.

Furthermore, we see that executions in our Nomad environment became slower with an increasing usage, whereas the performance of AWS was more stable. The behavior we observed for AWS Lambda is in line with findings from Baldini *et al.*, who describe pre-warming strategies: Extended phases of inactivity will cause the cloud provider to reduce the number of functions deployed for a customer [17]. For the Nomad environment, we implemented our own pre-warming mechanism with a static pool size (see Section II), thus preventing those issues in low-demand periods. Therefore, the survey results awarding Nomad with a slightly better performance after an extended phase of inactivity seem plausible to us.

In high-demand periods, the automatic scaling performed by AWS turned out to be advantageous. In those situations, the variance of the execution time was considerably lower with AWS Lambda than with Nomad, and the median time was almost unchanged in comparison to the entire course period. Potentially, it is this steadiness, which might cause learners to associate less technical issues to the code execution with AWS than Nomad. Overall, the constant performance of AWS Lambda is especially useful in high-demand periods, which can be used to balance peak loads and to provide learners with a stable execution environment at all times.

VII. Future Work

Through our work, we were not only able to test the general idea, but also gained first insights into the isolation of code executions on AWS Lambda. While AWS uses VMs to protect against malicious code as a provider, subsequent serverless functions might run in the same context, partially sharing some temporary files. Since we were unable to control which context is used by AWS, we cannot prevent the code execution of one user to access an arbitrary file previously created by another user. This limitation, usually being beneficial as another cache layer, is well documented by AWS [18], but is unacceptable for our use case. Therefore, we want to compare different providers of serverless functions in the future, focussing on function reuse and isolation capabilities. Additionally, we want to compare different pricing options (i.e., regarding the main memory or the processor architecture) and reduce the environmental impact.

Further, we want to expand our evaluation to cover more programming languages, take the learners' internet latency into account, and test the system in larger courses. The one chosen with about 1,600 active learners was a magnitude smaller than other courses we managed (with more than 17,000 active learners) and only focussed on object-oriented programming with Java. In some other courses, for example those with R and Python, we have other types of exercises featuring interactive Turtle graphics or allowing learners to download artifacts (e.g., figures created by their code run). Ideally, we would also transition to a language-agnostic function or a customized base image to accept incoming requests from our executor middleware Poseidon, making it easier to add support for additional programming languages.

VIII. Conclusion

Programming education in Massive Open Online Courses (MOOCs) benefits from hands-on exercises, allowing learners to apply their newly acquired knowledge. Since supporting learners individually on setting up a local development environment is not manageable by teaching teams, a web-based programming environment is desired [3]. One of those setup-free environments is our auto-grader CodeOcean, allowing learners to execute code and receive automated feedback. In this paper, we evaluated so-called serverless functions (offered by cloud computing providers) in an introductory Java course as an alternative to the current container-based execution.

Impact of serverless functions on code executions (RQ1): Generally speaking, serverless functions showed a higher median execution time compared to the container-based approach. However, this changed during high-demand periods, where our systems were slowing down, while the serverless functions continued to run with an unchanged performance. This makes the serverless functions ideal for peak workloads.

Learners' experience (RQ2): In a voluntary survey, learners rated both execution environments comparably, revealing only minor differences. Confirming our measurements, learners identified a slower execution after longer periods of inactivity with serverless functions (due to less pre-warmed executions), but also showed a higher satisfaction and less technical issues (potentially caused the more uniformed execution times).

Open questions and future work (RQ3): With the current implementation, we focused on Java exercises, leaving more advanced use cases (such as interactive Turtle graphics or downloadable artifacts) to future work. Also, we identified a challenge regarding the isolation of executions performed by different users that requires further investigation.

Overall, our evaluation suggests that employing serverless functions in the context of an auto-grader is feasible. It has shown to be especially beneficial in high-demand periods, outperforming our traditional container-based execution mode. Thereby, we contribute to scale online courses beyond their current (technical) limitations and empower an increasing number of learners to participate in programming MOOCs.

References

[1] A. Robins, J. Rountree, and N. Rountree, "Learning and Teaching Programming: A Review and Discussion," *Computer Science Education*, no. 2, 2003.

[2] H. T. Tran, H. H. Dang, K. N. Do, T. D. Tran, and Vu Nguyen, "An interactive Web-based IDE towards teaching and learning in programming courses," in *IEEE International Conference on Teaching, Assessment and Learning for Engineering (TALE)*, Bali, Indonesia, 2013.

[3] T. Staubitz, H. Klement, J. Renz, R. Teusner, and C. Meinel, "Towards practical programming exercises and automated assessment in Massive Open Online Courses," in *IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*, Zhuhai, China, 2015.

[4] T. Staubitz, H. Klement, R. Teusner, J. Renz, and C. Meinel, "CodeOcean - A versatile platform for practical programming excercises in online environments," in *2016 IEEE Global Engineering Education Conference (EDUCON)*, Abu Dhabi, 2016.

[5] S. Serth, D. Köhler, L. Marschke, F. Auringer, K. Hanff, J.-E. Hellenberg, T. Kantusch, M. Paß, and C. Meinel, "Improving the Scalability and Security of Execution Environments for Auto-Graders in the Context of MOOCs," in *Proceedings of the Fifth Workshop "Automatische Bewertung von Programmieraufgaben" (ABP 2021)*, Virtual Event, Germany, 2021.

[6] P. Blayney and M. Freeman, "Automated formative feedback and summative assessment using individualised spreadsheet assignments," *Australasian Journal of Educational Technology*, no. 2, 2004.

[7] V. Karavirta and P. Ihantola, "Serverless automatic assessment of Javascript exercises," in *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*, Ankara, Turkey, 2010.

[8] M. Pabst. "[Online-IDE] LernJ vs. Java: Unterschiede." (2023), [Online]. Available: https://www.learnj.de/doku.php?id=unterschiede_zu_java:start.

[9] R. Sharrock, L. Angrave, and E. Hamonic, "WebLinux: a scalable in-browser and client-side Linux and IDE," in *Proceedings of the Fifth Annual ACM Conference on Learning at Scale*, London United Kingdom, 2018.

[10] S. Strickroth, "Security Considerations for Java Graders," in *Workshop "Automatische Bewertung von Programmieraufgaben" (ABP 2019)*, Essen, Germany, 2019.

[11] J. Breitner, M. Hecker, and G. Snelting, "Der Grader Praktomat," in *Automatisierte Bewertung in der Programmierausbildung*, 6 vols., Münster, Germany, 2016.

[12] O. Flauzac, F. Mauhourat, and F. Nolot, "A review of native container security for running applications," presented at the The 17th International Conference on Mobile Systems and Pervasive Computing (MobiSPC 2020), Leuven, Belgium, 2020.

[13] R. Lobb and J. Harlow, "Coderunner: a tool for assessing computer programming skills," *ACM Inroads*, 2016.

[14] H. Bai, "GoAutoBash: Golang-based multi-thread automatic pull-execute framework with GitHub webhooks and queuing strategy," in *International Conference on Automation Control, Algorithm, and Intelligent Bionics (ACAIB 2022)*, Qingdao, China, 2022.

[15] M. Hofbauer, C. Bachhuber, C. Kuhn, and E. Steinbach, "Teaching software engineering as programming over time," in *Proceedings of the 4th International Workshop on Software Engineering Education for the Next Generation*, Pittsburgh Pennsylvania, 2022.

[16] J. Nupponen and D. Taibi, "Serverless: What it Is, What to Do and What Not to Do," in *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, Salvador, Brazil, 2020.

[17] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, "Serverless Computing: Current Trends and Open Problems," in *Research Advances in Cloud Computing*, Singapore, 2017.

[18] AWS. "Understanding the Lambda execution environment," AWS Lambda Operator Guide. (2023), [Online]. Available: https://docs.aws.amazon.com/lambda/latest/operatorguide/execution-environment.html.