Improving the Scalability and Security of Execution Environments for Auto-Graders in the Context of MOOCs

Sebastian Serth[1], Daniel Köhler[1], Leonard Marschke[1], Felix Auringer[1], Konrad Hanff[1], Jan-Eric Hellenberg[1], Tobias Kantusch[1], Maximilian Paß[1], Christoph Meinel[1]

Abstract:

Learning a programming language requires learners to write code themselves, execute their programs interactively, and receive feedback about the correctness of their code. Many approaches with so-called auto-graders exist to grade students' submissions and provide feedback for them automatically. University classes with hundreds of students or Massive Open Online Courses (MOOCs) with thousands of learners often use these systems. Assessing the submissions usually includes executing the students' source code and thus implies requirements on the scalability and security of the systems. In this paper, we evaluate different execution environments and orchestration solutions for auto-graders. We compare the most promising open-source tools regarding their usability in a scalable environment required for MOOCs. According to our evaluation, Nomad, in conjunction with Docker, fulfills most requirements. We derive implications for the productive use of Nomad for an auto-grader in MOOCs.

Keywords: Auto-Grader; Scalability; MOOC; Programming; Security; Execution

1 Introduction

The acquisition of fundamental digital knowledge is often an essential prerequisite for confident and self-determined participation in today's world. Gaining programming skills, in particular, is usually seen as an integral part of this knowledge. Learners can only acquire these skills through hands-on practice. While a teacher can personally instruct individuals from small groups, frequent manual feedback becomes laborious for larger learning groups and is impossible for Massive Open Online Courses (MOOCs) with thousands of learners. Hence, automated feedback, based on unit tests provided by the course instructors, is required in these self-paced learning scenarios.

So-called auto-graders automatically assess code submissions and often additionally provide a development environment to the learners. These online tools do not require any local software besides a web browser, which allows learners to get started quickly. Those tools execute the learners' code to provide the desired functionality and grade submissions. As outlined in Sect. 2, most auto-graders run the potentially untrusted code on their servers,

[1] Hasso Plattner Institute, University of Potsdam, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany, {sebastian.serth, daniel.koehler, leonard.marschke, christoph.meinel}@hpi.de, {felix.auringer, konrad.hanff, jan-eric.hellenberg, tobias.kantusch, maximilian.pass}@student.hpi.de

doi:10.18420/abp2021-1

imposing security and scalability questions for the provider. To improve the auto-grader that we currently use in our MOOCs, we performed a requirement analysis (cf. Sect. 3) and compared existing industry-standard execution environments in Sect. 4. We conclude our paper with an outlook on future work (Sect. 5) and summarize our findings in Sect. 6.

1.1 Current Architecture of our Auto-Grader CodeOcean

Nowadays, our auto-grader CodeOcean [St16] is used by several MOOC platforms and as part of teaching activities at our faculty. It hosts more than 650 exercises and typically runs around 2.5 million code executions per month during the duration of a MOOC. At this scale, performance and security aspects are important factors to consider. As shown in Fig. 1, the system consists of two major parts: the CodeOcean application providing the frontend and handling of submissions as well as a custom micro-service called DockerContainerPool managing pre-warmed, idling containers to improve the response time for learners.

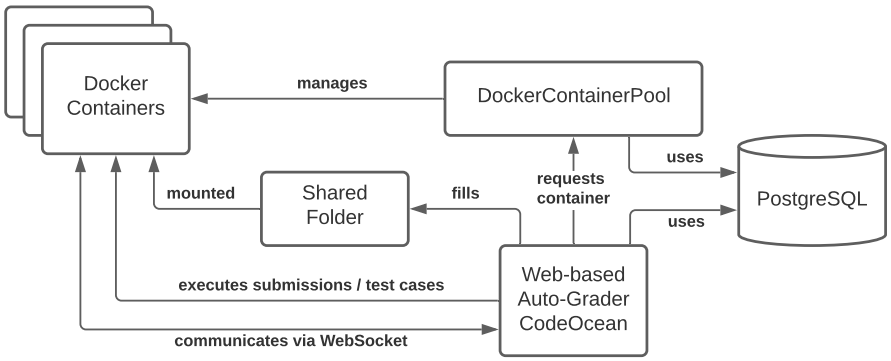

Fig. 1: The current architecture of our system consisting of two custom services for the code execution. The micro-service DockerContainerPool manages and pre-warms Docker containers while the CodeOcean application provides all user-facing interactions.

Even though the chosen architecture extracts the handling of Docker containers into a dedicated micro-service, the main application CodeOcean is tightly coupled with the DockerContainerPool. Moreover, the shared folder imposes additional security challenges for production use and makes scaling more difficult. Hence, a modernized solution is desired as a foundation for future development and educational research (cf. Sect. 5).

2 Background and Related Work

Automatically assessing code submissions using auto-graders and the implications of executing learners' code on a server have been the subject of several research projects. Two different approaches are distinguished for assessing code submissions: static and

dynamic code analysis [Ga13]. A trusted tool is used during the static analysis to inspect the submission without executing the analyzed code. For dynamic assessment, the code is run while the behavior is observed and compared with pre-defined expectations. In the context of auto-grader solutions, Garmann, as well as Strickroth, have described security implications and considerations when executing learner's Java code [Ga13, St19].

To isolate the code execution, web-based applications such as codeboard.io, the ranna code runner[2] or the grader Praktomat often rely on Docker containers [BHS16]. Docker is a containerization technique based on isolation concepts from the Linux kernel not requiring a virtual machine (VM) [An15]. Docker, as well as competing software, uses the containerd runtime[3] providing a standard interface to exchange the underlying container technologies. By introducing Firecracker as a so-called MicroVM with minimal hardware virtualization, Agache et al. increased the isolation of containers from the host intending to achieve comparable performance to containerized deployments [Ag20]. Similarly, Kata Containers provide full-featured VMs with support for container specifications and interfaces [RT19].

Scaling multiple containers across various hosts is done by so-called orchestrators to increase performance. Kubernetes and Nomad are two open-source tools scheduling and managing container clusters on multiple nodes[4]. Orchestrators allow horizontal scaling and to run jobs on-demand and are therefore relevant technologies for large-scale auto-graders.

For learning environments, even further approaches can overcome the resource and security constraints on the provider's server. Sharrock et al. demonstrated the technical feasibility of loading a sufficient Linux environment in the users' browser for use in a MOOC [SAH18]. However, to comply with our current architecture approach as well as to fulfill our interest in gathering data from the executions for further research, we aim to use approaches which allow us to keep the sovereignty about the execution with our system.

3 Requirement Analysis

Based on our existing auto-grader and experiences during past courses, we identified various requirements with the user-centered Design Thinking approach [MLP11]. Following this approach, we conducted qualitative, semi-structured interviews with a dozen representatives from identified target groups (learners, teachers, and administrators). Additionally, we inspected usage data of CodeOcean ranging back to first courses in 2015. Consequently, we identified the following three general requirements that any solution has to offer:

(1) Interactivity: Our auto-grader provides feedback for test runs and executes the learners' code in real-time. Therefore, offering a synchronous channel between the user and the execution shall allow a new command to start in less than one second.

[2] Source code of the ranna code runner on GitHub: `https://github.com/ranna-go/ranna`

[3] Homepage of the containerd project: `https://containerd.io`

[4] Kubernetes homepage: `https://kubernetes.io`, Nomad homepage: `https://www.nomadproject.io`

(2) Scalability: In the past, our auto-grader handled up to 120 execution requests per second with a mean execution time of fewer than ten seconds. While the system hits these peak usages only during a MOOC, these numbers provide a rough estimation for scalability requirements.

(3) Flexibility: Specifying runtime dependencies for containers should be as simple as with the current Docker-based solution and hence be compliant to the containerd ecosystem.

In addition to providing isolated processes and fulfilling these requirements, administrators urged the need to limit resources for code executions on their servers. Specifically, the execution time, memory consumption, and CPU should be limitable to provide a stable service for all users. As only a few exercises require network access, it should be generally restricted and only selectively enabled. Network access includes access to other hosts or forwarding specific ports to the learner. Optionally, allowing additional network orchestration with multiple nodes (e. g., for providing hands-on firewall exercises) would be a plus.

4 Evaluation of Code Execution Environments

Considering the requirements described in the previous Sect. 3, we practically evaluated containerization technologies and orchestrators (as described in Sect. 2) for the use by our auto-grader. We checked the extent to which each system meets our requirements and how extensive the effort for initial integration and following regular maintenance would be.

4.1 Execution Environments

As Docker, Firecracker, and Kata Containers share a similar feature set, we chose those technologies for our main comparison. Except for specifying an execution time limit, all three tools meet the technical demands for the required interactivity (1) once running. Docker and Firecracker also include a REST API easing the integration with a web-based auto-grader and providing a constant input and output stream to the containers.

When comparing the scalability (2), significant differences between the systems stand out. We measured some key metrics when using a Python container commonly used by our auto-grader on our production-grade system and under comparable, real-world conditions. For our experiment setup, we recorded the time for a container to complete the initialization and enter the idle waiting state (e. g., by subsequently executing the `sleep` command). Furthermore, we calculated the memory consumption of these idling containers. As shown in Tab. 1, Firecracker has the lowest memory footprint and the fastest startup time, followed by Docker. In repeating measurements with 10 and 100 concurrent container requests, we observed a linear increase in RAM usage for both. Kata Containers, in turn, had a significant increase in memory consumption, and we were unable to create 100 instances at once.

	1 Container		10 Containers		100 Containers	
	Time	RAM Usage	Time	RAM Usage	Time	RAM Usage
Docker	0.7s	47 MB	2.6s	333 MB	20.3s	3,244 MB
Firecracker	0.2s	35 MB	0.3s	375 MB	2.3s	3,900 MB
Kata Containers	3.2s	150 MB	7.0s	1,500 MB	Resource Limit Hit	

Tab. 1: The startup time for 1, 10, and 100 requested containers with different containerization technologies and the overall memory consumption when all containers are idling.

We assume that the underlying isolation technologies cause the observed differences. The setup bases Kata Containers on VMs requiring about 150 MB of RAM each. We conclude that they don't meet our second requirement, scalability, primarily due to the relatively slow startup speed. Similarly to Firecracker, they provide more robust isolation than Docker, which auto-graders would benefit from.

The third requirement, flexibility, refers to creating a new container environment for different programming languages. Docker (and Kata Containers) rely on a single so-called `Dockerfile` describing steps to build container images automatically. Each new container request uses these container images for the initialization process. Firecracker with the microVM concept requires two files: A bootable Linux kernel binary and a file system image. Even though different containers can reuse the Linux kernel and creating the required file system image can be automated, we argue that this process is more complex than the steps required for Docker. Combined with the low-level API of Firecracker, it makes using the tool more difficult even though it provides more robust isolation compared to Docker. Hence, for now, we decided against using it in our research-driven environment due to the increased setup and maintenance effort as well as the missing native support for `Dockerfiles`.

4.2 Execution Orchestrators

Similar to containerization technologies, we evaluated Kubernetes and Nomad for the desired use case in CodeOcean. Our main goal by using one of those orchestrators is to distribute the expected workload of 120 container requests per second to multiple machines with minimal manual intervention. Since the orchestrators abstract from the underlying execution technology, they also need to support the desired requirements (cf. Sect. 3). The two open-source tools evaluated both support the features mentioned above.

Therefore, we were most interested in the performance of both tools and the maintenance effort for researchers involved in the system management. For the performance evaluation, we repeated the experiment described in the previous Sect. 4.1 and focused on the startup time as the relevant metric. We deployed each tool to the same infrastructure for our assessment, with two nodes hosting Docker containers and one dedicated control plane managing the two systems. Tab. 2 shows the increase in startup times based on the number of requested containers for Nomad and Kubernetes. Interestingly, it took Kubernetes almost

eight times as long as Nomad to launch 100 containers. Even when using a managed cluster of the Google Cloud Platform with considerable more total resources, the startup times were not significantly lower or even worse as in our environment.

	Time for 1 container	Time for 10 containers	Time for 100 containers
Nomad	2.5s using 1 node	4.0s using 2 nodes	10.0s using 2 nodes
Kubernetes	3.0s using 1 node	8.0s using 2 nodes	78.0s using 2 nodes

Tab. 2: The startup time for 1, 10, and 100 requested containers using two popular container orchestrators and the number of nodes used for executing these containers.

Initially, we expected Kubernetes to outperform Nomad as it uses containerd directly rather than relying on Docker using containerd (which Nomad does). However, a Site Reliability Engineer daily working with Kubernetes explained that we were most likely observing the difference in scheduling algorithms between the two solutions.

Throughout the evaluation of Kubernetes and Nomad, we also observed the complexity of the tools during installation and configuration. Due to the vast amount of additional features which are mostly not required for our use-case, we identified Kubernetes to be more complex to configure. Since both tools show comparable performance otherwise, we decided for Nomad based on the differences in the startup time and the more straightforward configuration. The decision to use Nomad also means that we have to use Docker for now, as adapters to other containerization solutions are not yet production-ready.

4.3 Derived Implications for the Production System

Revisiting our current architecture (cf. Sect. 1.1) and the evaluation results, we created a proof of concept using Nomad and Docker containers. The architecture shown in Fig. 2 abstracts the handling of containers from our auto-grader CodeOcean and leverages this to a newly introduced middleware and the Nomad orchestrator. Most importantly, files are no longer provided to a shared folder mounted in the Docker container but instead copied to the container through Nomad. Additionally, the executor middleware abstracts from the concrete Nomad interface by providing a simplified representation of available execution environments. This detached architecture improves the system's scalability and increases the security of the containers through more robust isolation.

Hence, we plan to develop the executor middleware as a replacement for the current DockerContainerPool service. Based on the experiences we gained through our proof of concept, this middleware will primarily translate requests between the auto-grader and Nomad by abstracting the underlying container and orchestration technologies. Our goal is that the new service manages the pre-warming of containers, i. e., building and launching suitable containers for a single user. By providing separate environments for different users, their submissions cannot interfere with one another. Furthermore, the new service should enforce the time and resource constraints rather than leaving this to our auto-grader.

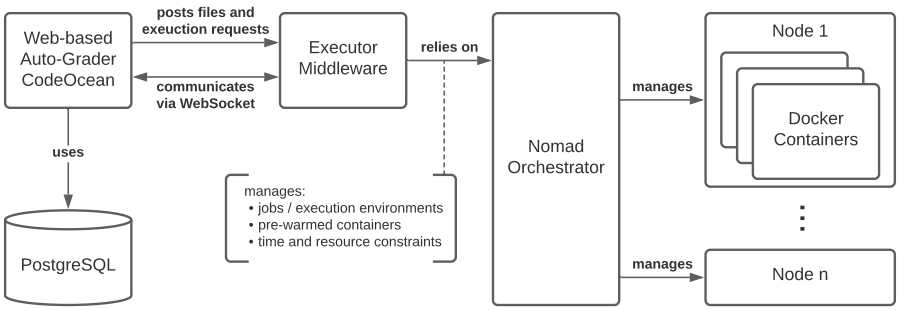

Fig. 2: The target architecture of CodeOcean consisting of the auto-grader, a custom execution middleware and a Nomad cluster. The main responsibility of the execution middleware is the management of desired states in Nomad and an abstraction of using pre-warmed containers.

5 Future Work

The proposed architecture and the insights collected from our evaluation are a foundation for future developments. The abstraction of executions allows more diverse environments as requested by various teaching teams in the past. Further, we want to investigate how course instructors can use CodeOcean in courses requiring specific hardware (such as a GPU or a Raspberry Pi). Besides that, we want to extend the auto-grading capabilities for networking and internet security courses by providing students access to network scenarios. A starting point for subsequent research could be integrating the interactive web page supplied by CodeOcean or remote shell access to one of the containers.

With the implementation of the described design and by achieving better scalability, we further provide the preconditions for longer-lasting executions. For example, we would like to offer advanced programming courses introducing learners to a debugger and give the required tools within our web-based programming environment CodeOcean. Therefore, we will add support for remote debugging of executions within a container and evaluate its impact on learners. On a technical level, we want to assess different pre-warming strategies and measure their effectiveness for handling varying workloads during a MOOC. Additionally, we plan further to compare other programming languages and their security concepts to develop our auto-grader and the new executor middleware.

6 Conclusion

In this paper, we approach enhancing our web-based, interactive auto-grader for MOOCs to enable improved security and higher scalability. From qualitative interviews with learners, teachers, and administrators, we identified a list of three main requirements: (1) interactivity, (2) scalability, and (3) flexibility. Considering these requirements, we evaluated several containerization and orchestration technologies to provide secure execution environments.

According to our analysis, Docker and Firecracker are the most promising container solutions and significantly more efficient than Kata Containers. When comparing Kubernetes and Nomad in conjunction with Docker containers for improved scalability across multiple nodes, we identified Nomad as the more performant solution for an auto-grader.

Finally, we propose a generalized architecture using Nomad and Docker for providing isolated and scalable execution environments in the context of CodeOcean. A newly introduced micro-service abstracts the management of containers and offers a simplified interface for the auto-grader. The resulting architecture provides a solid foundation for current and future use cases, thereby fostering the programming education in MOOCs.

Bibliography

[Ag20] Agache, A.; Brooker, M.; Florescu, A.; Iordache, A.; Liguori, A.; Neugebauer, R.; Piwonka, P.; Popa, D.-M.: Firecracker: Lightweight Virtualization for Serverless Applications. In: 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20). USENIX Association, Santa Clara, CA, USA, p. 17, February 2020.

[An15] Anderson, C.: Docker [Software Engineering]. IEEE Software, 32(3):102–c3, May 2015.

[BHS16] Breitner, J.; Hecker, M.; Snelting, G.: Der Grader Praktomat. In: Automatisierte Bewertung in der Programmierausbildung. volume Digitale Medien in der Hochschullehre, Waxmann Verlag, Münster, Germany, pp. 159–172, July 2016.

[Ga13] Garmann, R.: Sicherheitsimplikationen beim Einsatz von Test Doubles zur automatisierten Bewertung studentischer Java-Programme mit Graja und mockito. In: Workshop "Automatische Bewertung von Programmieraufgaben" (ABP 2013). volume 1067 of CEUR workshop proceedings, CEUR-WS.org, Hannover, Germany, p. 6, October 2013.

[MLP11] Meinel, C.; Leifer, L.; Plattner, H., eds. Design Thinking. Understanding Innovation 1. Springer, Berlin, Heidelberg, 2011.

[RT19] Randazzo, A.; Tinnirello, I.: Kata Containers: An Emerging Architecture for Enabling MEC Services in Fast and Secure Way. In: 2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS). IEEE, Granada, Spain, pp. 209–214, October 2019.

[SAH18] Sharrock, R.; Angrave, L.; Hamonic, E.: WebLinux: A Scalable in-Browser and Client-Side Linux and IDE. In: Proceedings of the Fifth Annual ACM Conference on Learning at Scale. ACM, London United Kingdom, pp. 1–2, June 2018.

[St16] Staubitz, T.; Klement, H.; Teusner, R.; Renz, J.; Meinel, C.: CodeOcean - A Versatile Platform for Practical Programming Excercises in Online Environments. In: 2016 IEEE EDUCON. IEEE, Abu Dhabi, pp. 314–323, April 2016.

[St19] Strickroth, S.: Security Considerations for Java Graders. In: Workshop "Automatische Bewertung von Programmieraufgaben" (ABP 2019). Gesellschaft für Informatik e.V. (GI), Essen, Germany, pp. 35–43, October 2019.