Comparing Jailed Sandboxes vs Containers Within an Autograding System

Matthew Peveler
Rensselaer Polytechnic Institute
pevelm@rpi.edu

Evan Maicus
Rensselaer Polytechnic Institute
maicue@rpi.edu

Barbara Cutler
Rensselaer Polytechnic Institute
cutler@cs.rpi.edu

ABSTRACT

With the continued growth of enrollment within computer science courses, it has become an increasing necessity to utilize autograding systems. These systems have historically graded assignments through either a *jailed sandbox* environment or within a *virtual machine* (VM). For a VM, each submission is given its own instantiation of a guest operating system and virtual hardware that runs atop the host system, preventing anything that runs within the VM communicating with any other VM or the host. However, using these VMs are costly in terms of system resources, making it less than ideal for running student submissions given reasonable, limited resources. Jailed sandboxes, on the other hand, run on the host itself, thus taking up minimal resources, and utilize a security model that restricts the process to specified directories on the system. However, due to running on the host machine, the approach suffers as new courses utilize autograding and bring their own set of potentially conflicting requirements for programming languages and system packages. Over the past several years, *containers* have seen growing popularity in usage within the software engineering industry as well as within autograding systems. Containers provide similar benefits of isolation as a VM while maintaining similar resource cost to running within a jailed sandbox environment. We present the implementation of both a jailed sandbox and container-based autograder, compare the running time and memory usage of the two implementations, and discuss the overall resource usage.

CCS CONCEPTS

• **Social and professional topics** → *Computer science education*; *Software engineering education*; • **Applied computing** → *Interactive learning environments*; *Learning management systems*;

KEYWORDS

Containers, Jailed Sandbox, Autograding, Docker

ACM Reference Format:
Matthew Peveler, Evan Maicus, and Barbara Cutler. 2019. Comparing Jailed Sandboxes vs Containers, Within an Autograding System. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19), February 27-March 2, 2019, Minneapolis, MN, USA.* ACM, New York, NY, USA, Article 4, 7 pages. https://doi.org/10.1145/3287324.3287507

SIGCSE '19, February 27-March 2, 2019, Minneapolis, MN, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5890-3/19/02.
https://doi.org/10.1145/3287324.3287507

1 INTRODUCTION

The demand for computing professionals across industries continues to grow. In conjunction, universities are seeing a growth in enrollment in CS courses and are having to teach and grade larger classes [3]. To handle this surge, many are turning to autograding systems, whether they be commercial or homegrown, to provide automated feedback to students on their assignments [4]. Increasingly, these systems are being seen as an absolute necessity for instructors as it no longer feasible to provide adequate manual feedback on each homework submission from a student [23]. In addition to this, instructors may wish to use a variety of programming languages and system resources to teach their courses. A study of CS1 courses in the UK found that 13 different programming languages were used [17]. Additionally, given that many of the languages identified on that list release new versions at least annually (or in the case of Javascript, a major revision biannually), ensuring the instructors have a specific version within an autograding system requires a flexible design, especially as instructors within the same department may disagree about the specific version of a language they wish to use for their courses.

We consider the following requirements necessary for a robust and flexible autograding system. These aspects are independent of the details of any assignment or grading evaluation criteria.

- A student should be able to submit an assignment multiple times, and should receive prompt automated testing feedback for each attempt.
- A student should not be able to view the submission files or autograding results of any other student on the server.
- A student's submission should be subjected to some limitations on allowed runtime, memory usage, size of generated artifacts, etc. to prevent runaway processes.
- A student may not affect the host system.
- A student may not utilize resources not explicitly granted to them as part of the autograding process (such as accessing the Internet).

Autograding systems have historically been implemented using either a jailed sandbox, which is fast and light on resources, but hard to isolate specific versions of dependencies per assignment, or a virtual machine/external server, which is slower, but allows for a full separation of dependencies per assignment per VM, assuming a unique VM per assignment configuration. Relatively recently, containers have seen a surge in popularity within the computing industry as a middle ground between jailed sandboxes and virtual machines, drawing upon each environments' strengths while avoiding their weaknesses. These traits, therefore, suggest that containers are potentially ideal for autograding. Within this paper, we examine and draw comparisons between the jailed sandbox environment versus utilization of containers. For the latter, we utilize the Docker [5] platform, which has greatly popularized usage of containers in both industry as well as for autograding. Through the course of the paper, we use Docker interchangeably with containers, but recognize that containers predates Docker, and that there are alternatives, such as Core OS rkt[19], for creating and managing containers. Virtual Machines and external servers (such as the Amazon EC2 platform [2]) are omitted from consideration. Utilization of VMs for proper isolation proves far too heavy in terms of used system resources as well as boot times, which often takes over a minute. While we could leave a VM booted continuously to avoid the boot time, we would then have to load the required dependencies for multiple courses/assignments such that it would then have the same issues as the jailed sandbox. Additionally, a booted VM generally requires upwards of 512MB of RAM which it will consume so long as it lives so that it can perform tasks that make sense within a desktop/server environment, but not necessarily for autograding. External servers are rejected due to the high associated cost of running them throughout the semester and/or explicit fees per submission per unit of autograding time. Furthermore, external servers face many of the same problems as VMs in addition to connection delays between the submission server and the external testing server.

In this paper, we provide a comparison of a jailed sandbox environment versus using containers for autograding. In Section 3, we introduce Submitty [21], our open-source course management platform for collecting, automated testing, and automated grading of programming assignments. In Section 4, we describe our jailed sandbox autograding implementation and the pros and cons of this approach. In Section 5, we discuss our implementation of autograding via Docker and its strengths and weaknesses. In Section 6, we measure the performance of these two approaches, covering resources used on the machine as well as time spent grading an assignment. In Section 7, we conclude our paper and discuss promising future lines of research for this work.

2 RELATED WORK

Many different autograding platforms have been created focusing on different aspects of the process. The first comprehensive surveys were done of such systems and their history in 2005 [1, 6]. Ihantola et al. point out in their survey of autograders that even though many of the developed prototypes are not open sourced or released [11]. However, the systems do have common functionalities. Additionally, prior work has been done to document many of the potential attack vectors a student may use in their submission [7] that a robust autograding environment must be able to handle.

Furthermore, while prior work discusses autograders, what they should do, and how they should operate [24], they largely gloss over the exact specifics of the environment within which they operate and provide little context for why they decided on a given environment. Additionally, beyond work done by Špaček et al. on how one might structure an autograder to utilize Docker [20], there is very little literature on using Docker for autograding, or the considerations one must take in using it. To our knowledge, there does not exist a work that provides a comparison of resource usage and throughput of a jailed sandbox vs. Docker for an autograding environment.

3 OUR AUTOGRADER IMPLEMENTATION

Within our university, we have implemented our own open source autograding system, Submitty [21]. Submitty has been in use in the Computer Science Department at Rensselaer Polytechnic Institute since 2014, and during the last year, was used by a dozen courses per semester with more than 2000 enrolled students combined. For our deployed server installation we use Ubuntu 18.04 and the packages that come with it, but Submitty also supports for Ubuntu 16.04 and Debian 7. Within this system, we have support for about 8 programming languages (e.g. Python, C++, Java, C, Scheme) used in our courses, as well as an equal number of instrumentation packages to performance static analysis, monitor memory usage, run test suites, etc.

To create a new assignment, an instructor writes a JSON specification file that describes any necessary compilation steps for the assignment, the testcases to be run, and how to validate the output from the code, most commonly using some variation of Myer's Diff [18] (e.g. doing a strict diff, allowing for lines to appear in any order, ignoring whitespace differences) or through usage of an automated test suite, like JUnit [12]. Students can re-submit the assignment multiple times before the submission deadline, each time receiving feedback on the success or failure of the *non-hidden* automated tests. Note: We suggest that instructors include additional *hidden* test cases to ensure that students do not simple hard-code solutions to the published test suite. Once the submission deadline is passed, student work may be further assessed manually by an instructor or TAs via an web interface that shows a rubric, as well as the output of each individual testcase, as well as the student's submitted code.

The instructor specifies how they wish students to submit to the server, either utilizing a drag-and-drop interface, or by using Git [9] version control wherein the server will clone the student's repository. The autograding pipeline is modular and allows for selection and customization of alternative grading methods (e.g., jailed sandbox vs. Docker) from within the JSON configuration. The Submitty system configuration allows specification of the overall number of concurrent autograders (based on server hardware specifications). We utilize this system and its associated autograding pipeline to analyze these two environments to compare and contrast the performance of the two implementations.

4 AUTOGRADING VIA JAILED SANDBOXES

A popular approach to handling autograding is to utilize a jailed sandbox environment, which is conceptually straightforward. In a jailed sandbox, the autograding process is run directly on the host machine utilizing whatever services/packages have been installed there. To achieve the secured sandbox so as to achieve our previously stated requirements, we have sufficient *untrusted* users. These users only have access to files within a specific workspace directory with access restricted to a single untrusted user. The autograding process for a student submission is then roughly as follows:

(1) Copy a student's submission into the untrusted user workspace
(2) Compile the student's submission as that untrusted user

(3) Run the testcases for the student's submission as that untrusted user
(4) Validate the results from the testcases
(5) Move the final results out of the untrusted user workspace

To perform this process, a user with a higher permission level does the first action of copying in the required files. The higher level user then executes the compilation, testing, and validation script as the reduced-privilege untrusted user. To further enhance security we could also use `chroot` to more robustly restrict the workspace to the working directory and to ensure that the untrusted user cannot access any files on the system with mistakenly set permissions and removing any environment settings for the system so as to prevent the untrusted user from accessing them. To restrict the actions and resources available to the student code, the untrusted user is limited to a whitelist of allowed system calls using the Linux `seccomp` library [22].

To protect the host system from programs that may unintentionally, accidentally, or maliciously hog CPU or RAM, open excessively-many files, attempt to write infinite length files, or "fork-bomb" the system with too many threads or processes, RLimits are put in place on the executing process. Anytime a process attempts to violate one of these limits, a related signal is sent to the process (such as SIGXCPU for using too much of the CPU), which is then translated internally to a SIGKILL on execution of the student's code.

While the jailed sandbox is in some ways straightforward and efficient, it also has significant drawbacks, principally in what resources remain available to the student code. Because student code is being run directly on the host machine, any program or package that's installed on the machine could potentially be run by the student. Students may have access to programs that an instructor did not want a student to use and did not even realize is installed on the host. While an instructor can use a `seccomp` to prevent a student from running additional processes from their code, if that filter is not used, then a student would potentially `exec` any program installed on the machine. However, while it is possible to handle this via a more complex virtualization of the filesystem, this adds additional complication for setting up an assignment for an instructor and for a system administrator to ensure correctness. Furthermore, many programming languages, such as Python, make use of modules. These modules are often times installed globally on the system such that any user may use them, which may complicate things when one class expressly wants students to use a module that another class directly forbids. Some of these languages feature methods of creating "virtual environments" that only have access to specific modules installed into the environment, utilization of them is language specific and complicated to fold into a jailed sandbox, while still maintaining an ease of use to instructors.

5 AUTOGRADING VIA DOCKER

An alternative to autograding via a jailed sandbox is to use containers. While much of the underlying technology of containers (such as cgroups and namespaces) have been around for over a decade, they have received a lot of recent attention due to Docker, which has allowed a lower barrier of entry to using containers. Containers provide a lightweight alternative to virtual machines, while still providing isolation of running processes from the host machine. This is accomplished by providing virtualization at the operating system level, which allows use of resources of the host system and its kernel. Containers share the base components of their runtime with the host as well as other containers which leads to a smaller footprint in running the containers. This is in contrast to VMs which perform their virtualization at the hardware level and prevent sharing of resources with other VMs. To make use of containers, much like for a VM, one needs to create an *image* that is used to create independent, isolated containers. We can easily create specialized container images targeting a specific language or specific packages without impacting other courses or existing assignments. This allows instructors to create unique environments for their courses, with whatever languages or packages they might want without affecting the host system or the containers prepared for other courses or assignments. Additionally, through the use of containers, it becomes possible to automatically test and grade networked or distributed programs that utilize isolated networks wherein the containers communicate to each other [13, 16].

The autograding process for Docker adds some additional steps over the jailed sandbox environment:

(1) Copy a student's submission into untrusted user workspace
(2) Create a container, mounting the untrusted user workspace to that container
(3) Execute the compilation step for an assignment
(4) Execute each testcase for the assignment
(5) Execute validation over the test cases results
(6) Destroy the container
(7) Move the results from the untrusted user workspace

We still utilize our implementation of a jailed sandbox inside of the container for added security and per testcase resource restrictions. Docker can be configured to similarly limit the CPU and memory a container uses. While this does effectively cover a similar effect as some of the RLimits from before, we still require RLimits to limit the size of files created by runaway processes.

5.1 Security Considerations

By default, utilization of a container provides isolation of automated testing of a student's submission from other students as well as the host machine. Additionally, by default, a container is not allowed to communicate with the outside Internet. In addition to these container-specific details, we also utilize parts of the security model built around the sandbox environment. This means that within our container, we utilize seccomp to specify what system calls are allowed to be made in execution of their programs. However, it should be noted here that given that one could run any OS within a container, not just one matching the host, a common pitfall of seccomp is to not consider how a different OS may require different filter flags to achieve the same result within seccomp [8].

However, unlike in the sandbox environment, the default user that runs a process within a Docker container is root. Additionally, the folder that was mounted into the container in step (2) above has the same owner/group as the host system. This means that it's not enough to simply not run as root, but rather we must run as the same untrusted user inside the container that owns that workspace. While it's possible to exec a command on container creation to create the necessary user, Docker also allows one to pass in a specific group ID (GID) and user ID (UID) to execute all commands within a container under. These IDs need not exist on the container, and so long as you do not attempt to get the name of the user/group (such as via whoami), this works seamlessly, and removes the risk of running programs within the container as root.

5.2 Docker Images

A necessary step of autograding via Docker containers is the creation of Docker images. For VM images, a user generally installs an OS, installs some packages, and then creates an archive out of the resulting image or uses provisioning software (e.g. Chef, Salt, Puppet) with its own format to create an image. These steps are either manually labor-intensive in the case of the former or require knowledge of the specialized format in the latter. In contrast, Docker images are specified by creating a *Dockerfile*. This file begins with a base image (major linux distros such as Ubuntu and Debian are available as base images) is specified to build from and then a series of *RUN* commands to be executed on that base image using the default shell of that distro (e.g. bash). Docker then handles getting the base image and running each command, returning a built image, without requiring the user to manually type the commands inside the VM. However, to produces the smallest possible Docker images, one must condense run commands as much as possible and not add unnecessary packages. Listing 1 presents a sample Dockerfile built from the Debian base image.

```
FROM debian:stretch-slim

RUN apt-get update \
    && apt-get -y --no-install-recommends install \
       grep \
       libseccomp-dev \
       libseccomp2 \
  && rm -rf /var/lib/apt/lists/*

CMD ["/bin/bash"]
```

Listing 1: Example Dockerfile

To keep the size of Debian and Ubuntu images small, we use the *no-install-recommends* flag, we remove the cache of available packages created by the *apt-get update* command, and we execute these commands in a single RUN line. The choice of base image is significant as well. Table 1 presents the base size of four popular Docker images.

Image	Size
alpine:3.8	4.41MB
ubuntu:18.04	84.1MB
debian:stretch-slim	55.3MB
debian:stretch	101MB

Table 1: Popular Docker images and their sizes

While it would seem that one would want to always use Alpine [15] thanks to its very small footprint, it can be more difficult to add the necessary programs and packages. For example, Alpine does not come with the `glibc` and it is not easy to build packages like Clang6 without prior Alpine experience. Thus, we selected the *debian:stretch-slim* image thanks to its more familiar interface, general ease in building any package, relatively small size, and similarity to our host OS Ubuntu (to avoid problems with seccomp).

Submitty allow instructors to specify their own complete Dockerfile, or use our simple command-line interface to mix and match Dockerfile components (such as Python 3.6, Clang 6, Java8, etc.) and specify additional system packages and language modules.Instructors specify the appropriate Docker image within the assignment configuration JSON. Additionally, these images can be distributed to students for use during offline development and testing.

6 PERFORMANCE ANALYSIS

While containers are billed as being lightweight compared to virtual machines, they are not as lightweight as a jailed sandbox, due to the additional OS virtualization layer. To analyze these costs, we identified several busy periods of student submissions from a prior term. Even the busiest time periods did not overwhelm the CPU or memory resources of our server hardware (which is ensured by our conservative limit of the machine to grading at most ten assignments in parallel). We selected the busiest four hour period during the Fall 2017 that semester – a time period with ~540 unique submissions in our CS1 and CS2 courses. The period selected occurred the night of their common assignment deadline and happened on the 9th week of the semester when both classes are tackling more compute-intensive assignments. The submissions gave us a mix of perfect full-credit submissions, submissions earning partial credit, submissions that failed to compile or run at all, and submissions that entered into infinite loops. We then took this time period and replayed it (and repeated the time period) with different time speedups over an hour to artificially stress the hardware and test the scalability. We also increased the number of allowed concurrent autograding processes. We repeated each simulation twice, running it first under the jailed sandbox setup, and then running it using Docker containers. Fo the Docker test, we prepared a single "master" image with both Python and C++, and additional utilities used by CS2, such as Valgrind, as well as some extraneous packages, with a total image size of 2.1 GB. The system hardware we utilized for these tests was a Dell Poweredge R520 with an Intel Xeon ES-2470 (20 Cores, 40 Threads) CPU and 32 GB of RAM.

During the execution of the time period we measured the following statistics:

- Number of active graders every 0.1 seconds
- CPU utilization percentage every 0.1 seconds
- Memory usage every 0.1 seconds
- Time taken for a submission to be executed and graded
- Time spent by a submission waiting in the queue for grading to begin
- Time spent creating a container (for Docker)
- Time spent destroying a container (for Docker)

These measures were then averaged over every minute within the one hour. For time spent, for any minute that did not have any processes either finish grading or finish waiting, we took the average of its nearest neighbors that did have those values. We then plotted the system performance metrics against each other in terms of percentage used versus the overall amount available and then

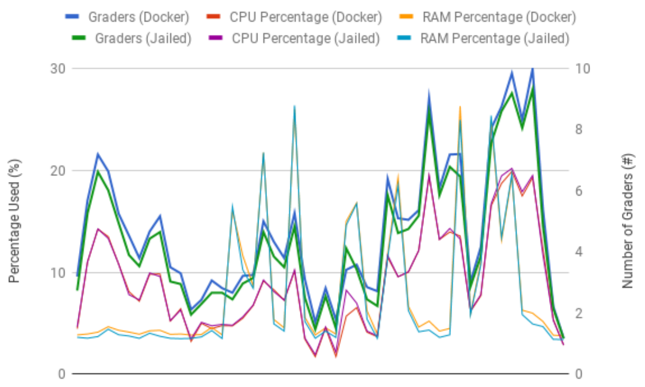

Figure 1: Resource usage for 4X simulation

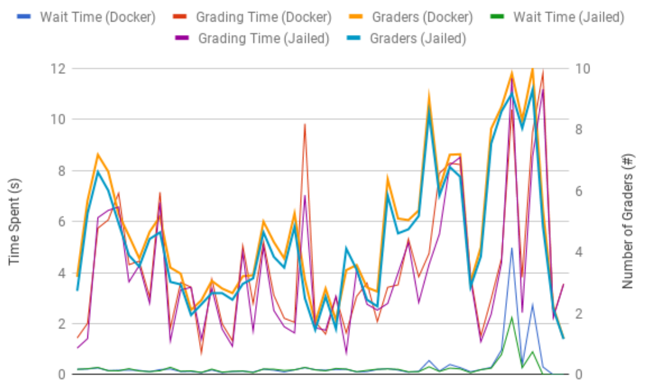

Figure 2: Average time spent for 4X simulation

created a plot of the average time spent grading and average time spent waiting.

6.1 Results

For our first considered simulation, we took the four hour block and replayed it once within an hour block (4X the recorded load). Within that block, we allowed for 10 concurrent graders to operate. The results of this simulation are shown in Figures 1 and 2. The CPU used for Docker versus the jailed system are roughly equivalent. The RAM, while higher for Docker, had on average less than a percent difference. The time used for grading is also roughly comparable, with an average of 0.6 seconds more spent for Docker versus the jailed sandbox and then an almost equal time spent waiting for each submission throughout. However, we see Docker has two periods where the wait time deviates from the jailed sandbox wait time (the first around 3 seconds and the second around 2 seconds). Just before the assignment deadline (at the end of the 4 hour window) the machine has reached our conservative limit of concurrent autograding processes. This happens because for each container, we spent around ~1.4 seconds for creating the container, and about ~1 second for destroying the container, making each grader last an

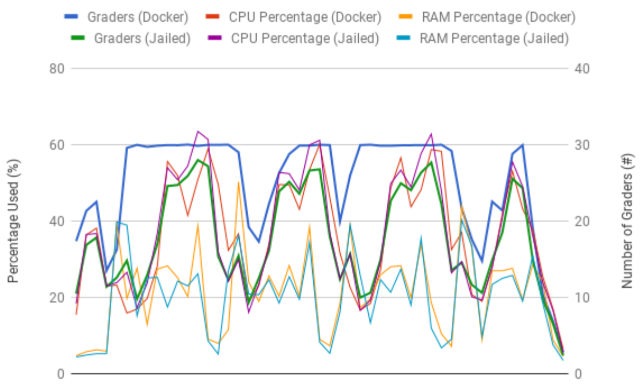

Figure 3: Resource usage for 16X simulation

extra ~2.4 seconds with Docker versus the jailed sandbox. While the production server could handle this load without an increase in wait time, the slight overhead of Docker did result in a small wait time for the equivalent simulation replay.

For our second presented simulation, we took the same four hour block and condensed it to 15 minutes, and then replayed that 4x to make up the hour (effectively a 16X of realtime recorded load). We also bumped up the number of allowed concurrent graders to be 28 to partially handle the simulated increase in submissions per hour. Note that the number of concurrent graders is still less than the number of threaded cores on the machine (40), which should not unduly stress the machine for the single-threaded student assignments in our CS1 and CS2 courses. The results of this simulation are shown in Figures 3 and 4. Here, we see that Docker ends up utilizing all of the concurrent graders for a large fraction of the simulation hour. This ends up causing Docker to start utilizing more RAM (with a peak difference of about 15%) and CPU (with a peak difference of about 5%). The jailed sandbox environment on the other hand never needed to use all of the graders at one time. More importantly, we see a far more drastic change in the amount of time taken to grade an assignment and the wait time per submission. Docker, once maxed out, with the additional time cost of creating and destroying containers, cascades through the system to subsequent submissions, giving larger fluctuations in the amount of time spent waiting for an assignment to start grading, which at the worst of times added almost an additional 25 seconds of waiting, and on average was above 10 additional seconds. By contrast, the jailed sandbox ran more efficiently, maintaining a wait time that was close to zero seconds spent waiting per submission. While under the full load of the concurrent graders, we saw that the time taken on average to create a container to be 7.4 seconds and destroying a container to be 3.3 seconds.

6.2 Discussion

From these results, we can conclude that while containers do have a cost related to them in terms of resources, the primary cost is paid in terms of time taken with container creation and destruction. Even when not all graders were being utilized on a machine, each student on average experienced an additional 2 seconds slowdown

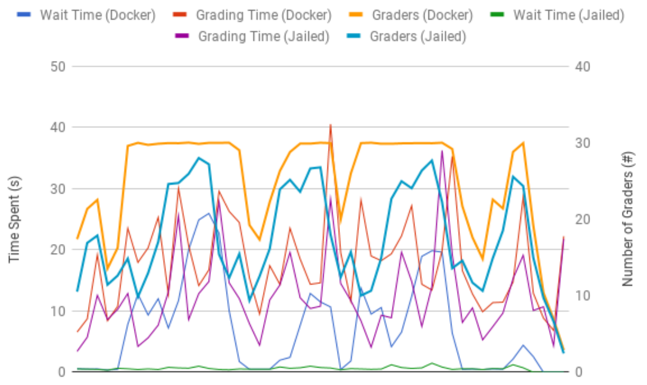

Figure 4: Average time spent for 16X simulation

per submission. However, once the server was under a full load, that time rose to an average of almost 11 seconds per submission. While not a drastic change in time taken, any time increase to a student is disliked, especially during that the busiest and most stressful period before the assignment deadline.

The spikes in the grading time in the data largely corresponded to students who had submitted programs that had infinite loops that would therefore run for several minutes until finally being killed off by the system via the RLimit. While this affected the average grading time for an assignment in the time slice where it occurred, more importantly, it meant that a grader was taken out of circulation for an abnormally long time which ended up causing a cascade effect on other graders that were potentially similarly hogged and which ended up causing more severe fluctuations in wait time and grade time for submissions.

Finally, the image that we was used for these test was not specialized per class or per assignment, rather bringing in all packages that might be needed for multiple courses, resulting in a relative large image size. By creating more specialized images per course and per assignment, we may see that the amount of time taken in creation and destruction of a container decreases as well, while secondarily also not requiring as much space on the host environment's hard drive to store the container.

We did attempt to test Docker underneath additional concurrent graders (up to 40) for the 16X simulated test case; however, the system was not stable. Grading processes would crash when container creation failed showing a definite limitation of using Docker for autograding. When we equivalently bumped up the number of concurrent graders for the jailed sandbox environment beyond the number of threads on the machine, the system and grading processes ran slower, but remained stable and operational. More development will be needed to tune the system and robustly handle near-capacity autograding load in containers.

7 CONCLUSION

Given the results of our performance testing, switching from the jailed sandbox to Docker containers for autograding is quite promising. In this work, we discussed the configuration, strengths, and weaknesses of both implementations. We analyzed the CPU and RAM utilization and timing statistics over the lifetime of the autograding processes for submitted assignments. Through this, we found relatively negligible differences in performance between the two environments, except when the machine was under a significant load at which point containers started to perform much worse in terms of time taken for container creation and destruction. Overall, we believe that the strengths of containers far outweigh the weaknesses, with the caution that care should be taken to tune the system configuration for large loads and further investigate issues impacting of system robustness.

In this work, we utilize a simplistic view of the container life cycle as it is applied to autograding. As shown in Section 5, within a single autograder process, a container is created, the student's code is run and evaluated, and then the container is destroyed. However, as shown in Section 6, most of the time cost of using a container is in the construction and destruction portions of that life cycle. Looking forward, we would like to investigate ways to mitigate these costs so that to a student, the time cost of using a container vs a jailed sandbox is relatively equal. We first consider the startup cost.

To improve this, we take inspiration from other sources (e.g. web servers, databases) and how they handle fluctuations in load. With these services, it is common to define a "pool" of available workers for a given process, oftentimes defining a minimum number of workers in the pool as well as the maximum size that it can grow to. Similarly, for the containers, we can work to maintain a container pool such that for a new submission, it can take a container from the pool instead of having to create the containers on-demand for all submissions. However, unlike the aforementioned examples, to maintain the flexibility that container usage affords us, we must further investigate algorithms to handle which container images we need to fill our pool with. As a first step in this direction, we can maintain a count of images used within some time frame, and then fill our pool with the most used images within that frame, with the expectation that the next submission will draw from the same distribution. Further work around this would be to investigate and utilize forms of machine learning and analysis to allow us to tune this pool on demand.

Opposite the start up time of containers, to improve the destruction time, we can offset this process to be done outside of the core process of autograding of a submission. To do this, we can create a container garbage collector that cleans up containers asynchronously at some fixed interval. Implementations of these strategies, while allowing for a better throughput has to be balanced out with the potential increase in number of containers that may exist at one time, especially in regards to a GC for recycling old containers. As stated, in Section 6, we found that once we had too many running containers at once, Docker would start to error out in attempting to create more.

REFERENCES

[1] Kirsti M Ala-Mutka. 2005. A Survey of Automated Assessment Approaches for Programming Assignments. *Computer Science Education* 15, 2 (2005), 83–102. https://doi.org/10.1080/08993400500150747 arXiv:https://doi.org/10.1080/08993400500150747
[2] Amazon. 2006-2018. Amazon EC2. https://aws.amazon.com/ec2/
[3] Computing Research Association. 2017. *Generation CS: CS Undergraduate Enrollments Surge Since 2006.* Technical Report.

[4] John DeNero, Sumukh Sridhara, Manuel Pérez-Quiñones, Aatish Nayak, and Ben Leong. 2017. Beyond Autograding: Advances in Student Feedback Platforms. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17)*. ACM, New York, NY, USA, 651–652. https://doi.org/10.1145/3017680.3017686

[5] Docker. 2013-2018. https://www.docker.com/

[6] Christopher Douce, David Livingstone, and James Orwell. 2005. Automatic Test-based Assessment of Programming: A Review. *J. Educ. Resour. Comput.* 5, 3, Article 4 (Sept. 2005). https://doi.org/10.1145/1163405.1163409

[7] Michal Forisek. 2007. Security of Programming Contest Systems.

[8] Tal Garfinkel. 2003. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[9] Git. 2005-2018. https://git-scm.com/

[10] Georgiana Haldeman, Andrew Tjang, Monica Babeş-Vroman, Stephen Bartos, Jay Shah, Danielle Yucht, and Thu D. Nguyen. 2018. Providing Meaningful Feedback for Autograding of Programming Assignments. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. ACM, New York, NY, USA, 278–283. https://doi.org/10.1145/3159450.3159502

[11] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of Recent Systems for Automatic Assessment of Programming Assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research (Koli Calling '10)*. ACM, New York, NY, USA, 86–93. https://doi.org/10.1145/1930464.1930480

[12] JUnit. 2018. https://junit.org/junit5/

[13] Federico Kereki. 2015. Concerning Containers' Connections: On Docker Networking. *Linux J.* 2015, 254, Article 2 (June 2015). http://dl.acm.org/citation.cfm?id=2807678.2807680

[14] Stephan Krusche and Andreas Seitz. 2018. ArTEMiS: An Automatic Assessment Management System for Interactive Learning. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. ACM, New York, NY, USA, 284–289. https://doi.org/10.1145/3159450.3159602

[15] Alpine Linux. 2018. https://alpinelinux.org/

[16] Evan Maicus, Matthew Peveler, Stacy Patterson, and Barbara Cutler. 2019. Autograding Distributed Algorithms in Networked Containers. In *Proceedings of the 2019 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '19)*. ACM, New York, NY, USA.

[17] Ellen Murphy, Tom Crick, and James H. Davenport. 2017. An Analysis of Introductory Programming Courses at UK Universities. *Programming Journal* 1 (2017), 18.

[18] Eugene W. Myers. 1986. An O(ND) difference algorithm and its variations. *Algorithmica* 1, 1 (01 Nov 1986), 251–266. https://doi.org/10.1007/BF01840446

[19] Core OS. 2014-2018. rkt. https://coreos.com/rkt/

[20] František Špaček, Radomír Sohlich, and Tomáš Dulík. 2015. Docker as Platform for Assignments Evaluation. *Procedia Engineering* 100 (2015), 1665–1671.

[21] Submitty. 2014-2018. http://www.submitty.org/

[22] Wikipedia. 2018. Seccomp. https://en.wikipedia.org/w/index.php?title=Seccomp&oldid=853577449 [Online; accessed 27-August-2018].

[23] Chris Wilcox. 2015. The Role of Automation in Undergraduate Computer Science Education. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)*. ACM, New York, NY, USA, 90–95. https://doi.org/10.1145/2676723.2677226

[24] Chris Wilcox. 2016. Testing Strategies for the Automated Grading of Student Programs. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16)*. ACM, New York, NY, USA, 437–442. https://doi.org/10.1145/2839509.2844616