Opunit: Sanity Checks for Computing Environments

Samim Mirhosseini(✉) and Chris Parnin(✉)

North Carolina State University, Raleigh, NC 27695, USA
{smirhos,cjparnin}@ncsu.edu

Abstract. Computing environments, including virtual machines and containers, are essential components of modern software engineering infrastructure. Despite emerging tools that support the creation and configuration of computing environments, they are limited in testing and validating the construction of these environments. Furthermore, professionals and students new to these concepts, lack feedback on their construction efforts. In this paper, we argue that the design of environment testing tools should fundamentally support asserting essential properties, such as reachability and availability, in order to maximize usability and utility. We present OPUNIT, an environment testing tool that supports assertion of these properties. We describe properties students failed to check when testing computing environments, which guided the design of OPUNIT. Finally, we share our early experiences with using OPUNIT in the classroom to support education and training in configuration of computing environments.

Keywords: Configuration management · Environment verification · Testing · DevOps training

1 Introduction

Software developers no longer simply build software in isolation: They now are expected to continuously deploy fixes and experimental features to production environments serving millions of customers. Making such ultra-fast and automatic changes to production means that testing and verifying the design and implementation of computing environments is increasingly important. However, based on the 2018 State of DevOps Report [6], only 36% of participants have capacity for dedicated testing of computing environments in their companies, making environment construction easy to get wrong. For example, GitLab lost 300 GB of customer data after accidentally deleting their production database [5]. Even worse, they could not restore the data because they discovered their backup procedure had been failing due to a mismatch in versions between the dump utility (`pg_dump 9.2`) and their database (`PostgreSQL 9.6`).

Unfortunately, the skills required to construct and test these computing environments supporting continuous deployment requires expertise and training that

J.-M. Bruel et al. (Eds.): DEVOPS 2019, LNCS 12055, pp. 167–180, 2020.
https://doi.org/10.1007/978-3-030-39306-9_12

is even more rare and highly sought than data science skills.[1] For example, Mozilla's Kim Moir says she "recently looked at the undergrad classes required to graduate with a computer science degree from a major university, and [she] was struck by [a lack of] practice on deploying code. In most computer science programs, there is little emphasis on infrastructure" [3]. Similarly, Google's Boris Debic claims that "Release engineering is not taught; it's often not even mentioned in courses where it should be mentioned" [3]. For this reason, Facebook's Chuck Rossi considers hiring release engineers "is like finding unicorns."

In this study, we used our experiences and observations from five years of teaching over 400 students the concepts and tools related to continuous deployment in a university course [1]. Consider one assignment, where students were installing and configuring an open-source chat server called Mattermost[2], which works much like Slack[3]. The computing environment requires several components: a database, system dependencies, the Mattermost server itself, and several configuration files (`systemd` services, `mysql.cnf`, and `config.json` for Mattermost). In configuring this environment, many things could go wrong. For example, a simple typo or malformed JSON in a configuration file could result in an non-functioning environment, but with little hints as to why. To diagnose this problem, students might need to check a variety of system components using a myriad of tools and shell utilities in which they have little experience, such as `mysql` shell, `systemctl`, `journalctl`, `cat`, `grep`, and `jq`. In response, we would have to ask a series of questions: "*did you check your mysql credentials*," "*did you check your connection string is correct*," "*did you run jsonlint on your configuration file*." Other times, strange behaviors would result from incidental factors, which we would only resolve after asking, "*did you check dns*," "*did you check your VM's memory size*." Overall, this experience of asking students to perform various sanity checks eventually helped, but resulted in a frustrating and problematic learning environment for students.

To make matters worse, no single tool can support this process meaning students must simultaneously learn many. For example, `ps`, `top`, `ss`, `cURL`, `netcat`, `free`, `lsof`, `who`, `last`, `dmsg`, `history`, `vmstat`, `dstat`, `iostat`, `htop`, `find` and more. In this paper, we argue for two ways to help with the mentioned shortcomings: (1) Train software engineers to be able to recognize desirable properties of a computing environment, (2) Provide them a simple means for evaluating these properties. To this end, we formalized these checks in a simple environment verification tool, OPUNIT. We categorized common student mistakes and issues into violations of properties that computing environments should have. These properties can be verified to easily point out the cause of common issues related to environment setup. Categories of the properties which we include are *availability*, *reachability*, *identifiable*, and *capability*. They respectively indicate whether an environment provides expected services, can access specified resources, has certain items (files, software, etc.), and supports required operations.

[1] http://stackoverflow.com/insights/survey/2017#salary.
[2] https://mattermost.com/.
[3] https://slack.com/.

Finally, we share our early experiences with OPUNIT as a training aid in a DevOps course. First, we used OPUNIT to verify the student's initial local computing environments to ensure they contained appropriate tools and capabilities for the course. Next, we used OPUNIT in workshops and homework assignments to provide formative feedback on their progress (Fig. 1). Then, we administered a usability and feedback survey. Students indicated OPUNIT has increased their confidence about their work because they could ensure they have completed tasks correctly by running the tests. They also showed continued interest in using the tool for other courses and future assignments.

To summarize, our contributions are:

- Environment properties that are root causes for the most common issues students experience.
- OPUNIT, a tool for environment verification, inspired by the common properties in student issues.
- A survey about OPUNIT to suggest its effectiveness as a training tool.

2 Properties

Based on our experience with students in software engineering courses and a specialized DevOps course, we categorized common student mistakes. Then we identified four main properties of a computing environment which can be checked to point out these mistakes. In this section we explain the properties that we identified, an example of student issues related to those properties as well as a verification method that can help point out the issue, and finally application of those properties.

2.1 Availability

Environment functionality depends on availability of services that were set up in previous steps of environment construction. One common issue that students face occurs when they write a whole configuration script without intermediate testing. As a result, they often experience errors which they incorrectly ascribe to the last step they worked on. In reality, the errors often lie in one of the earlier steps. By supporting the ability to check *availability* of services, students can better test their configuration scripts incrementally, allowing them to establish stepping stones of progress.

Example Problem. *Expected services are not available in the environment, because they have not been started:* The goal is to run automated GUI tests for a web application using `Selenium`[4]. Students run the tests, but the server was not able to start successfully before the tests executed. As a result, all of the GUI tests fail as none of web application pages can be served. They often think

[4] https://docs.seleniumhq.org/.

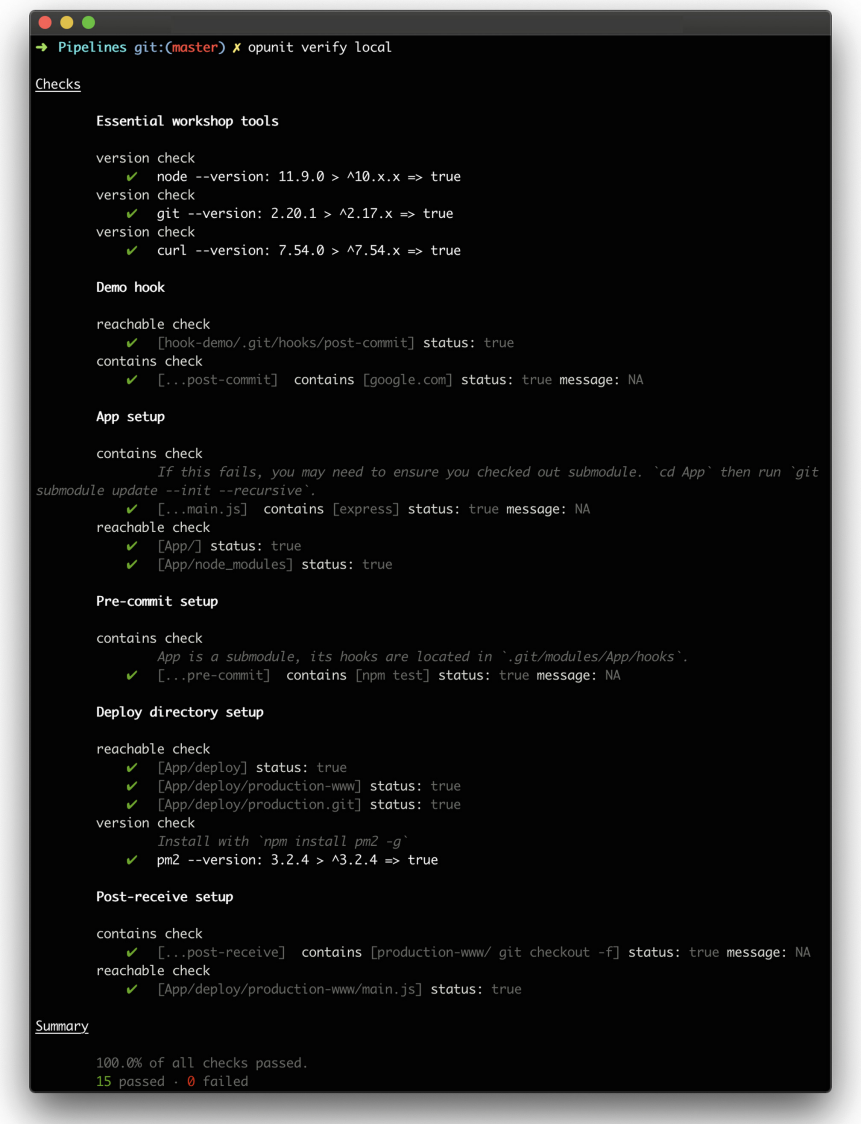

Fig. 1. OPUNIT's verify command to test pipelines workshop

this is because of not running `Selenium` tests correctly, or the tests are really failing. More careful inspection of logs is required to find the reason for the test failures.

The failed server start up can have many different causes. For example, a `bind exeception` could occurs if there is another server running on the same port and often happens due to other instances of the same application still running in the

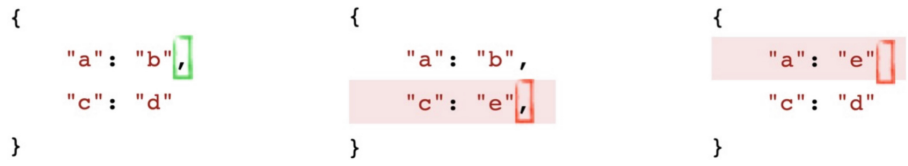

Fig. 2. Examples of students' broken JSON files shown in red color (Color figure online)

background. Another cause can be a broken formatting in configuration files, like an extra "," at the end of a JSON configuration file. This was a common failure because students used string replacements instead of using a utility like `jq`, and created a broken JSON format as shown in Fig. 2 in red color.

Example Verification. If the configuration management scripts was tested incrementally, student would have been able to send a simple HTTP request using `cURL` utility to test if the server is started and can respond to requests.

Application. This property helps with ensuring availability of services before running a task. For example, it can be implemented as a simple HTTP request to a web server, to see if it is available. This idea has been implemented in Google Borg's tasks [7]. Each task implemented an internal health check end-point, and this allowed Borg to send an HTTP request to this end-point to do a health check on each task. Automating the steps for checking availability property allows the user to do a quick health check without having to learn `cURL` utility or other more complicated tools.

2.2 Reachability

Another common issue among students is unexpected software failures as a result of an *unreachable* resource. We might not be able to access a resource because of various reasons such as a missing/wrong configuration file, wrong file permissions, and bad firewall rules. Checking reachability of these resources can help find the reason for the failures. In other words, after discovering an *unavailable* service, checking *reachability* of its related resources can help find the root cause for this unavailability.

Example Problem. *Database is not reachable in the environment because credentials has not been updated in a configuration file:* The goal is to start a web application that requires database access. This application uses a configuration file to store database credentials. Forgetting to update and correctly ensure appropriate access rights to configuration files is a common mistake among students. The application may start without explicit errors, and the UI pages may even be rendered, but the pages will be missing information. Finding the problem will require more careful inspection of the logs from this web application.

Example Verification. Existence of database configuration file should be checked using `ls -l <config_file>`, and this file's permissions should be accessible by the application. So students need to at least understand Unix file permissions and know what parameters they need to use with `ls`. While the check itself may be relatively simple, students may not be well-attuned to pay attention to details such as mismatches in group permissions of a file. Automating these steps will also require experience with tools such as `grep`. And finally, if the permission needs to be changed, students also need to understand how to use the `chmod` command.

Application. Reachability issues in industry, especially in microservices, is even more crucial: "*Reachability is definitely an important thing, security group changes that make downstreams unreachable in a microservice architecture can be dangerous.*"[5] Automated verification of reachability of the resources will prevent reachability issues. It can be implemented as a series of requests to all the needed resources, and triggered after each change to know when reachability is affected.

2.3 Identifiable

Another common property that causes confusion for students is related to the version of installed software, wrong content in configuration files, and such *identifiable* properties or items in the environment. We called these types of properties identifiable because of their relation to one of the core components in traditional configuration management, "identification".

Example Problem. *Unexpected behavior when wrong version of a dependency is installed:* One of the most common observed issues with setting up an environment for running a specific software occurs when incompatible versions of dependencies are installed. For example, if the software required `MySQL v5.7`, it may not work as expected if version v8 is installed. *GitHub does not link commit authors to their profile on GitHub:* Another example of *identifiable* property is when students forget to create git configuration file, `.gitconfig`, and as a result their git commits are not linked to any GitHub account.

Example Verification. Most utilities use a `-v or --version` option to print their current version; this can be used to check if the installed version is the same as the expected version. Also, the content of the configuration files can be checked by opening the files and manually checking for expected changes. Manually checking these properties may not seem very difficult, but automating the steps for checking them will require experience with Unix utilities such as `cat`, `grep`, `awk`, and more.

[5] Personal correspondence from industry.

Application. One of primary objectives in real-world configuration management is to install tools and systems, and fine-grain details. Many of these details can be categorized as *identifiable* properties. As we explained earlier, a serious case of not testing this property happened at GitLab in 2017. GitLab's version of dump utility (`pg_dump 9.2`) was not compatible with the version of their database (`PostgreSQL 9.6`) which resulted in failure in the backup process and unrecoverable loss of 300 GB of customer data. A simple verification of the versions could prevent such incidents. Automating the steps needed for checking the mentioned *identifiable* properties of environment will enable the user have more confident about their environment setup without having to learn how to write a testing script.

2.4 Capability

Capability property is about ensuring that the system has sufficient resources to support required operations. *Capability* of the environment is typically related to the hardware, which is another important property that can effect how applications run. A few examples of this property are number of CPU cores, amount of RAM, free disk space, and virtualization support.

Example Problem. One of the workshops in our software engineering and DevOps courses focuses on provisioning virtual machines. 64-bit virtual machines require having a CPU which supports virtualization (VT-x on Intel and AMD-V on AMD CPUs). Most modern CPUs and laptops support virtualization but many manufacturers disable this feature by default. So, when students try to create a virtual machine, they receive a complicated error messages which is hard for them to understand.

Example Verification. On Windows, virtualization status can be checked in Windows Task Manager. On Linux virtualization support can be checked by inspecting CPU flags and looking for `vmx` and `svm` flags. Modern Apple computers (macOS) have virtualization enabled by default.

Application. One of the most common issues with setting up a system for building java programs, such as a Jenkins[6] executor, was memory limitations. Students would provision instances with 1 GB of RAM, and would experience a variety exotic errors, none of which made it clear insufficient memory was the root cause. By introducing a capability check for RAM, we can reduce the likelihood that students experience these issues.

[6] https://jenkins.io/.

3 Opunit

Inspired by the properties that we identified, we developed an environment testing automation tool, OPUNIT. Figure 1 shows an example of the test results on a DevOps workshop which was about constructing a delivery pipeline using git hooks. This workshop was completed inside a virtual machine. In this study, we are specially concerned with the needed verification in the initial phase of environment creation, rather than monitoring the application for changes.

The goal of OPUNIT is to be a simple tool for verifying the construction of a computing environment by asserting the properties we introduced. Often, multiple properties must be verified and checked in order to understand the cause of a misconfiguration.

3.1 Using Opunit

OPUNIT uses a YAML configuration file, `opunit.yml`, to define the verification steps. Listing 1 shows an example `opunit.yml` file. The verification steps are defined under `checks` property. In this example, OPUNIT will be using `node --version` command to verify version of `node` is in semver[7] range `^10.x.x`. OPUNIT tests can be started with `opunit verify` command. OPUNIT searches the default paths for an `opunit.yml` file and runs the provided checks against the target environment.

```
- group:
  description: "Check node.js support"
  checks:
  - version:
      cmd: node --version
      range: ^10.x.x
```

Listing 1: An example `opunit.yml` file with a simple check for having the appropriate version of nodejs installed.

3.2 Checks

OPUNIT uses automated scripts, *checks*, to implement how each property needs to be checked. To verify the Availability property, OPUNIT uses a check called "availability" which runs a command on target environment followed by a HTTP request to do a health check. A simple example of `version` check is shown in Listing 1. This check has two parameters, the command that needs to be executed to get the version, and a semver range that the version should be in. OPUNIT has *checks* to verify all the mentioned properties in Sect. 2 and each require different parameters which are explained in more details in OPUNIT documentation[8].

[7] https://semver.org/.
[8] https://github.com/ottomatica/opunit.

In summary the supported checks are `availability` to check if a service can be started successfully, `reachability` to check reachability of specified resources, `contains` to check content of specified files, `version` to check version of the specified tool and comparing it with the provided semver range, `service` to check status of installed Linux services, `timezone` to check timezone of the environment, `cores` to check number of available CPU cores, `virt` to check if virtualization is supported, and `disk` and `memory` to check the memory size available disk space.

3.3 Environments

The target Environment that OPUNIT verifies can be the local machine, a remote server, a virtual machine or a container. OPUNIT also supports all three common operating systems, macOS, Windows, and Linux. Supporting various types of environments and operating systems allowed us to use OPUNIT in classroom.

The environment type in some cases can be automatically inferred based on the existence of other configuration files in the project, or the arguments passed to the `verify` command. For example if there is a `Vagrantfile` in the project, OPUNIT will try to connect to that Vagrant virtual machine. Or, if OPUNIT is executed with `opunit verify root@example.com:2222` command, then OPUNIT will use ssh to connect to the target environment. OPUNIT has a few more advanced inference rules in the tool's documentation which we don't discuss in this paper.

3.4 Report

After OPUNIT verification checks are executed, the results are printed in the terminal window. Figure 1 shows an example of this report. The green check (✔) indicates that a check was passed, while the red x (✖) indicates that a check failed. The report is very verbose and includes both expected and actual values for each check. Each check can also include a description defined in `opunit.yml` file. The descriptions for the *checks* proved very useful for learning in workshops as we discuss in the next sections.

4 Experiences

To better understand the impact of using OPUNIT in the classroom, we integrated the tool in our DevOps course. In this section we discuss the experiences of students using OPUNIT, as well as feedback we received from them.

4.1 Supporting Initial Course Setup

In the first week, students are required to prepare their local development environment for the rest of semester. `opunit profile CSC-DevOps/profile:519.yml`[9]

[9] https://github.com/CSC-DevOps/profile/blob/master/519.yml.

verifys their development environment's configuration. `profile` is an `opunit.yml` file hosted in a GitHub repository.

The resulting output is shown in Fig. 3. Notice that one of the checks under "Editor Support", fails to validate. This check looks for syntax highlighting being enabled for `vim`. This check fails because the `.vimrc` file is not present on the machine.

Fig. 3. Result of running an OPUNIT profile

4.2 Using Opunit for Workshops

We added OPUNIT checks in a workshop about pipelines by providing students an `opunit.yml` file. In this workshop students learn how to use git hooks to run static analysis checks before committing their code and then triggering deployment of an application on `git push`. We provided them an interactive way of knowing what they need to complete for the workshop. Each OPUNIT check had a description that help with understanding the corresponding task. When students start the workshop, all the OPUNIT checks fail and as they complete the workshop, they see OPUNIT checks start passing.

YAML snippet in Listing 2 shows the `opunit.yml` file used in the workshop. In this snippet three types of checks are shown, `contains` check, `reachable` check and `version` check. `contains` check verifies that students have updated the `pre-commit` hook to run `npm test` command, `reachable` check verifies students created needed directories, and `version` check verifies they installed a version of `pm2` package in the range `^3.2.4`.

```
1  - group:
2      description: "Pre-commit setup"
3      checks:
4        - contains:
5            comment: App is a submodule, its hooks are located in
             ↪  `.git/modules/App/hooks`.
6            string: npm test
7            file: .git/modules/App/hooks/pre-commit
8
9  - group:
10     description: "Deploy directory setup"
11     checks:
12      - reachable:
13          - deploy
14          - deploy/production-www
15          - deploy/production.git
16      - version:
17          comment: Install with `npm install pm2 -g`
18          cmd: pm2 --version
19          range: ^3.2.4
```

Listing 2: Part of the `opunit.yml` file used in the pipelines workshop

4.3 Student Feedback

After students used OPUNIT for supporting their development environment setup and for completing a workshop, we sent them a feedback form with open-ended responses and an usability survey [2] to collect data about their experience with the tool. We used this feedback to find possible issues and determine if OPUNIT was effective in supporting students.

Feedback. In the feedback form, we asked students to explain how their experience with OPUNIT was comparing to the other assignments that they completed without using OPUNIT. The responses showed that using OPUNIT made it very easy for the students to know if they completed all the necessary tasks or they missed something. In many instances students explained how OPUNIT saved them a lot of time by showing them descriptive errors about what mistakes they made in doing a task. Students explained that they had higher level of confidence when they completed the workshop that took advantage of OPUNIT. Finally, students also showed interest in using OPUNIT in their future assignments and even in other courses.

Usability. On the usability survey, we asked students ten multiple choice questions as shown in the Likert chart in Table 1. Summary of the survey responses confirms the findings of our general feedback form, about *higher level of confidence* and *interest in using the tool in the furure*. Additionally, student responses showed OPUNIT was easy to learn without spending too much time. Most of the students also think they are likely to be able to use OPUNIT in their future projects, without needing assistant from us.

OPUNIT has been effective in classroom and provided good support for training configuration of computing environments. Very few students had difficulty in running the tool. They mostly liked seeing the green check marks after completing each task and indicated this increased their confidence. Students even showed interest in using OPUNIT in future assignments and other courses. We think this is the right direction for OPUNIT, however there are limitations which we try to resolve, and improvements which we plan to add. We discuss these limitations and future directions in next sections in more details.

Based on our observation, we believe one of the reasons for why students are often confused and have a hard time when debugging environments is that they fail to **read** and **understand** the error messages. In many cases that students asked us for help in debugging, we noticed the errors explicitly and clearly indicates the problem. However, students either did not carefully read the error messages, or the did not understand it. An example of such error message is "`Permission 0644 for /Users/ubuntu/id_rsa are too open.`" which makes SSH ignore a key. As mentioned by an StackExchange user who asked a similar question[10], a reason for not reading the error messages and logs can be frustration.

> *I failed to read the output due to a combination of frustration, disillusionment and pessimism*

As mentioned by students in our general survey, OPUNIT improved students' confidence. If the written `opunit.yml` file includes description for the possible causes of the failures, it can especially be helpful for student who missed the error message details as we mentioned earlier.

> *It's all about confidence and I think that opunit gives me such confidence.*

[10] https://superuser.com/questions/1159790/chocolatey-python-am-i-doing-it-wrong?rq=1#comment1672782_1159793.

Table 1. Follow-up survey responses

		Likert Responses[1]					
	% Agree	SD	D	N	A	SA	Distribution[2]
I thought opunit was easy to use.	92%	0	0	2	15	11	
I think that I would like to use opunit in my future projects.	89%	0	1	2	11	14	
I found the various features in opunit were well integrated.	88%	0	0	3	19	4	
I would imaging that most people would learn to use opunit very quickly.	85%	0	1	3	11	13	
I felt very confident using opunit.	64%	0	2	8	9	9	
I needed to learn many things before I could get going with opunit sanity checks.	28%	6	10	4	6	2	
I think that I would need assistance using opunit in my future projects.	14%	7	9	8	3	1	
I thought there were too much inconsistency in the opunit tool.	3%	8	13	6	1	0	
I found opunit very cumbersome/awkward to use.	3%	15	10	2	1	0	
I found opunit unnecessarily complex.	0%	12	15	1	0	0	

[1] Likert responses: Strongly Disagree (SD), Disagree (D), Neutral (N), Agree (A), Strongly Agree (SA). [2] Net stacked distribution removes the Neutral option and shows the skew between positive (more useful) and negative (less useful) responses. ■ Strongly Disagree, ■ Disagree, ■ Agree; ■ Strongly Agree.

5 Future Directions

OPUNIT is a new tool and it's important to realize its limitations. One limitation is the type of checks that OPUNIT supports. Although OPUNIT checks cover many common properties that we identified, there could be more properties which we have not considered. Furthermore, current OPUNIT checks can be extended to support more fine-grain verification. To address this, we accept pull requests and feature requests for the tool, and we are actively adding more checks as we find the need for them.

After seeing promising effectiveness in the current version of OPUNIT, we think adding a CI system integration is an appropriate next step. Using OPUNIT in a CI system will allow developers and students automatically get feedback about the changes they make on every git commit. Another possible future direction for OPUNIT are adding monitoring capabilities and combining our idea of checks with chaos engineering principles [4]. This will allow developers easily measure resilience of the environment and configuration in turbulent conditions.

Additionally we plan to extend our interviews with the professionals to find other properties that are checked in industry and improve the list of supported checks in OPUNIT. The new OPUNIT checks that we have identified and plan

to implement are integration with different services. For example, support for verifying write access of a GitHub token, or verifying needed rules in AWS[11] EC2 security groups. Finally, as we mentioned earlier, the currently supported checks still can be improved by better fine-grain verification.

6 Conclusion

This paper describes the design of an environment testing tool, OPUNIT, guided by experiences and observations obtained after five years of teaching the concepts and tools related to continuous deployment. Our experience in a DevOps course showed that our tool was effective and this could be a step in the right direction, however there is more work to be done.

Acknowledgement. This material is based in part upon work supported by the National Science Foundation under grant number 1814798.

References

1. DevOps 519. https://github.com/CSC-DevOps/Course/#devops-csc-519
2. Opunit Survey. https://forms.gle/uhBYmtftdsfj5TxP8
3. Adams, B., Bellomo, S., Bird, C., Marshall-Keim, T., Khomh, F., Moir, K.: The practice and future of release engineering: a roundtable with three release engineers. IEEE Softw. **32**(2), 42–49 (2015). https://doi.org/10.1109/MS.2015.52
4. Basiri, A., Jones, N., Blohowiak, A., Hochstein, L., Rosenthal, C.: Chaos Engineering. O'Reilly Media, Inc., Newton (2017)
5. GitLab: Postmortem of database outage of January 31. https://about.gitlab.com/2017/02/10/postmortem-of-database-outage-of-january-31/
6. Puppet: 2018 state of DevOps report. https://puppet.com/resources/whitepaper/state-of-devops-report/
7. Verma, A., Pedrosa, L., Korupolu, M.R., Oppenheimer, D., Tune, E., Wilkes, J.: Large-scale cluster management at Google with Borg. In: Proceedings of the European Conference on Computer Systems (EuroSys), Bordeaux, France, p. 18 (2015)

[11] https://aws.amazon.com/.